NTNU

Norwegian University of
Science and Technology

Ethereum Smart Contracts: Security Vulnerabilities and Security Tools

Ardit Dika

Applied Computer Science
Submission date: December 2017
Supervisor: Mariusz Nowostawski, IDI

Norwegian University of Science and Technology
Department of Computer Science

Preface

This project is a Master's thesis conducted during the autumn term at NTNU, Gjøvik. The author was initially introduced to Ethereum and blockchain technology during a lecture in 'Advanced Course in Mobile Technology' by his supervisor Mariusz Nowostavski.

The architecture of Ethereum requires developers to have a novel engineering mindset. It is therefore important for non-security experts to become familiar with the process of developing secure smart contracts and equip themselves with the knowledge of mitigating security vulnerabilities. More generally, it has become increasingly essential for computer scientists and developers to be 'security aware' and consider security implications throughout the process of their development. Therefore, this thesis targets developers of smart contract applications and blockchain enthusiasts within computer science research.

A background in Computer Science is preferable for the understanding of this thesis. However, no prior knowledge of blockchain or Ethereum is necessary because the key terminologies used are explained in detail.

15-12-2017

Acknowledgment

I would like to express my utmost gratitude to my supervisor Mariusz Nowostavski for his continuous and constructive feedback as well as his expert advice. I truly appreciate your extensive support throughout the process of this dissertation. I would also like to thank Rune Hjelsvold for his advice on academic research which genuinely helped improve the quality of this work.

Furthermore, I thank the SmartDec team for sharing an early beta-version of their tool SmartCheck with us – this was a great addition to the thesis.

I want to sincerely thank my girlfriend Albesa for her loving and endless support throughout my studies as well as her valuable feedback on the thesis.

And finally, I genuinely thank my family for their unconditional support.

Faleminderit shumë, Babi & Mami!

A.D.

Abstract

Ethereum represents the second generation of blockchain technology by providing an open and global computing platform which allows the exchange of cryptocurrency (Ether) and the development of self-verifying smart contract applications. Smart contracts present a foundation for possessing digital assets and a variety of decentralized applications within the blockchain area. Ethereum and smart contracts are public, distributed and immutable, as such, they are prone to vulnerabilities sourcing from simple coding mistakes of developers.

Motivated by the security breaches and recurring financial losses in smart contracts, we aim to advance the field of security in smart contract programming. The main objective is to aid smart contract developers by providing a taxonomy of all known security issues and by inspecting the security code analysis tools used to identify those vulnerabilities. Based on previous research as well as attacks on Ethereum smart contracts, we propose an updated taxonomy which categorizes all known vulnerabilities within their architectural and severity level. Our second proposed taxonomy is a novel categorization of security tools on Ethereum.

Furthermore, we conduct the investigation of security code analysis tools on Ethereum by assessing their effectiveness and accuracy. In particular, we analyze four security tools, namely, Oyente, Securify, Remix, and SmartCheck. The results indicate that there are overall inconsistencies between the tools on different security properties. SmartCheck outperformed the other tools in terms of effectiveness, whereas Oyente performed the best in terms of accuracy. Furthermore, based on the limitations we identified, we propose future improvements within the user interfaces, interpretation of results, and additional vulnerability checks.

Contents

List of Figures

List of Tables

1 Introduction

The section below provides a brief introduction to the three main terms necessary for the comprehension of this chapter and the overall thesis; blockchain, Ethereum, and smart contracts. However, if you are already familiar with these terms, you can jump to Section 1.1.

Innovative technological advancements are introduced regularly with the purpose of finding novel and better approaches of implementing systems and software products more effectively. One example of such a technology is **blockchain**, which has the capacity of being implemented in many use cases such as internet interaction systems, public services, Internet of Things (IoT), and financial systems. Blockchain technology and cryptocurrencies have experienced a steady increase of attention from academia and the industry [1]. Simply said, blockchain represents a fully-distributed public ledger and a peer-to-peer platform which makes use of cryptography to securely host applications, transfer digital currency/messages and store data [2]. It was initially introduced by Satoshi Nakamoto[1] in 2008 and implemented one year later in a peer-to-peer electronic cash system named Bitcoin [3]. After that, many new and novel ideas have been proposed and implemented, which will be mentioned later throughout this thesis.

The second most popular blockchain platform as of November 2017, based on the current cryptocurrency market capitalization[2], is **Ethereum** - which is simultaneously the main focus of this work. Throughout our research work we have encountered different definitions of Ethereum. However, for an initial and simplified introduction we will use the definition of Vitalik Buterin (the inventor of Ethereum) taken from a panel discussion. He explains Ethereum as a general purpose blockchain, meaning that the blockchain is able to understand a general purpose programming language. This allows developers to build a variety of applications like the ones mentioned in the beginning, all in one blockchain, instead of building a separate blockchain for each use case or application[3]. This is the key component that differentiates Ethereum from Bitcoin. In addition to serving as a decentralized peer-to-peer platform to exchange cryptocurrency (Ether[4]),

[1] Anonymous (invented) individual or group of people.

[2] Cryptocurrency Market Capitalization: https://coinmarketcap.com/

[3] Video: "Technologies that will decentralize the world." https://techcrunch.com/video/decentralizing-everything-with-ethereums-vitalik-buterin/59c01b739e451049f87f8c18/

[4] Ether: "The crypto-fuel for the Ethereum network." Link: https://ethereum.org/ether

Ethereum also introduces, for the first time, the idea of a Turing-complete blockchain. Turing-completeness programming language in a blockchain-based platform creates the opportunity for implementing a handful of decentralized applications in financial and non-financial areas.

These decentralized applications in Ethereum and in other blockchain-based platforms that support Turing-completeness, are referred to as **smart contracts**. As mentioned previously, this is the main component that makes Ethereum differ not only from Bitcoin but also from other blockchain-based platforms[5] because of its originality/novelty and also as a first mover in this field. However, the concept of *smart contracts* itself was first introduced by Nick Szabo in 1997, where he describes them with a real-life canonical/primitive example of the humble vending machine [4]. The rest of this chapter and the background chapter specifically, will cover a more detailed explanation of smart contracts since they are of a prime importance to this thesis.

1.1 The Scope

Ethereum is considered a relatively new and highly experimental platform, both because of the time when it was introduced (July 2015[6]), as well as its ability to create distributed applications with a Turing-complete programming language running in a decentralized, peer-to-peer platform like blockchain. Therefore, this area of interest has gained a lot of attention from academia the last two years. Being a new and highly experimental platform, it consists of many ongoing issues and challenges. The most prominent one is the security aspect of the platform itself and also the security issues within smart contract applications. These issues are discussed in a systematic mapping study by Alharby and Moorsel, which identify current research topics and open challenges; the results show that two thirds of the papers focus on identifying and tackling smart contract issues [5]. Simply said, the primary topic which is covered in this project is Ethereum's security aspect.

Respectively, this thesis investigates the security vulnerabilities of smart contract applications on Ethereum, and analyzes the security code analysis tools used to identify vulnerabilities and bugs in smart contracts. To the best of our knowledge and based on our thorough research of the relevant literature, we were not able to find any research which focuses on the analysis of Ethereum's security tools. Whereas, there are many related research works regarding the security vulnerabilities of smart contracts, which are further explained in detail in Section 3.

This thesis takes two approaches; the first one is focused on a current literature

[5]There are other blockchain platforms that support smart contracts implementation, but they are either not-popular or non-Turing complete.

[6]Link: https://blog.ethereum.org/2015/07/30/ethereum-launches/

review and provides an updated and modified taxonomy of the security vulnerabilities based on their classification and severity level, whereas the second approach is based on an experiment with the current existing/available security tools on vulnerable and audited smart contracts, by assessing their accuracy and effectiveness.

1.2 Keywords

Blockchain, Ethereum, Ether, smart contracts, security vulnerabilities, security tools, Solidity, Ethereum Virtual Machine (EVM), attacks, security audits, incidents, scams.

1.3 Problem Description

In addition to expressing business logic and handling different, sometimes heavy computational tasks, nowadays (based on the Ether price[7]), smart contracts also present a foundation for possessing costly digital assets (e.g. cryptocurrencies, tokens). This means that there are currently financial and semi-financial smart contracts which are worth thousands and millions of dollars. Mainly because of this reason, smart contracts and the Ethereum platform are continuously a target for adversaries and manipulators. As a result, one of the main potential research areas (based on the Ethereum community [6]) is within the **security drawbacks of high-level programming languages**. The research community predominantly proposes further research and development work in formal verification, techniques for analyzing smart contracts, and defensive programming.

Furthermore, what Ethereum and other popular blockchain platforms have in common, is the **publicly visible data**. This is a result of having a decentralized peer-to-peer network and distributed ledger among thousands of nodes[8]. Hence, Ethereum is referred to as *The World Computer*[9], by its development team. Regardless of the positive impact and many benefits that this approach has on Ethereum and generally in any other public distributed blockchain, it presents a serious challenge on implementing specific use cases of smart contracts, considering the fact that the complete source code of an application is publicly visible from anyone in the network.

Another key characteristic of Ethereum's Virtual Machine (EVM) specifically, is that **once you deploy your smart contract** in the blockchain, **you cannot modify or alter it**. This characteristic can both be seen as advantageous and disadvantageous. The advantage is that it represents a trustworthy platform where the developers cannot modify the smart contract once they have deployed it, with the sole purpose of gaining illegal profit and misleading the users. This leads to the

[7] $305 as of the time of writing: https://ethereumprice.org/

[8] Ethereum as of May, 2017 has nearly 25,000 nodes. Link: http://www.trustnodes.com/2017/05/31/ethereum-now-three-times-nodes-bitcoin

[9] Ethereum: the World Computer: https://www.youtube.com/watch?v=j23HnORQXvs

notion of *immutability* which is a big discussion topic for Ethereum, and it is further explained in Section 2.2. Its disadvantage is its novel development approach with which developers are not yet familiar, and thus ignoring this issue leads to unprecedented vulnerabilities and makes developers reluctant to adopting this novel engineering mindset.

Due to the above mentioned issues, **a significant number of smart contracts are considered to be vulnerable**. In 2016, a symbolic execution analysis tool (Oyente) was developed by Luu et al. [7], which analyzed all smart contracts in the Ethereum blockchain in order to identify several potential vulnerabilities[10]. Their results state that at that time, 45% of 19,366 smart contracts in total were vulnerable with at least one security issue [7]. To conclude, regardless of Ethereum's popularity and its promising opportunities, its security issues and challenges pose a real threat for the effective continuity and the further increase of its adoption and popularity.

1.4 Justification, Motivation and Benefits

The overall issues and challenges as well as the ongoing attacks and incidents on the Ethereum platform and smart contracts create a solid justification for this thesis. The dearth of research analyzing the current security tools on Ethereum adds a significant value and additionally justifies the experiment conducted. Currently there are several tools assessing smart contract vulnerabilities. However, only one so far, assesses and explains how they address the issue of accuracy and results precision [7]. Also, considering the fact that new vulnerabilities are exploited all the time, the current taxonomies get quickly outdated. This creates a gap and room for further research on the potential improvement of security tools and security vulnerability taxonomies which is the research focus of this work.

Developing self-verifying and self-executable applications, such as smart contract applications which run in a decentralized (not controlled by anyone specifically) platform, represents a revolutionary approach of development and application interaction. The opportunity to preserve this feature and the notion of immutability and trust in Ethereum, serves as the main motivation for this thesis. In addition, the ability to develop securely without being a security or blockchain expert, strengthens the motivation for the conducted experiment. Finally, the finding that as of May 2016, 45% of smart contracts are vulnerable [7], significantly drives the motivation of our work.

Thorough research and a comprehensive experiment on security vulnerabilities and code analysis tools, as presented in this thesis, gives a better insight on this

[10]Vulnerabilities that Oyente is able to identify: transaction-ordering dependence, timestamp dependence, mishandled exceptions and reentrancy.

topic and creates extensive future work opportunities. Generally the stakeholders to benefit from this thesis are Ethereum users/developers altogether.

1.5 Research Questions

As it was mentioned before, this thesis addresses two separate but closely related approaches of Ethereum's security. The first one provides updated and modified taxonomies in relation to the security vulnerabilities and security code analysis tools (their classification and severity level), based on a thorough research work. The second one provides close insight and a detailed perception of the current available security tools used to identify vulnerable smart contracts, and is based on extensive research and a practical experiment. Therefore, we aim to answer two main questions with a few sub-questions for each.

- **Research Question 1:** *What kind of security vulnerabilities arise in smart contract programming?*
    - **Research Sub-Question 1:** *How can we categorize these vulnerabilities?*
    - **Research Sub-Question 2:** *Why do these mistakes occur? What are the factors that raise these vulnerabilities?*
    - **Research Sub-Question 3:** *How can we mitigate these vulnerabilities and promote a new secure engineering mindset for programming smart contracts?*
- **Research Question 2:** *What are the limitations of the current security tools used to identify vulnerabilities in smart contract applications on Ethereum?*
    - **Research Sub-Question 1:** *What is the current state of the security tools in regards to their accuracy, effectiveness and consistency?*
    - **Research Sub-Question 2:** *What are the possible future improvements that could be done, in regards to security code analysis tools?*

1.6 Contributions

The initial contribution of this thesis is the proposed up-to-date and modified taxonomy of vulnerabilities, their architectural classification, in conjunction with their severity level. Additionally, two novel taxonomies are proposed. The first one aids us on classifying already developed security code analysis tools based on the methodology they use to identify the vulnerabilities, the user interface, and the technique used to analyze smart contracts (bytecode and/or high-level programming language). This gives an overall insight on the current state of the tools. The second one is a matrix based on the already developed security tools versus the vulnerabilities they cover. Doing so, we investigate the absent vulnerabilities which are not covered generally by the tools.

The second contribution consists of an experiment on several security tools to assess their accuracy, effectiveness, and consistency. This generates results, such as false positive and false negative rates and an overall discussion on how effective these tools are in analyzing the smart contracts from the complete data set collected. Throughout our experiment we have been in contact with the development teams of two, out of four security tools, and they were informed for any experienced issue or error with the tools. Also, one of the development teams (SmartDec[11]) were very eager to participate in such unbiased research, and therefore, gave us early closed beta-version access to their yet unreleased security tool, in order to get our feedback. Once we were done with the experiment, we assembled the results particularly for SmartCheck and sent a constructive feedback to the SmartDec team.

The main contribution of this thesis is considered the advancement and improvement of the security aspect of Ethereum through a research work and an experimental project. The main stakeholders of the first approach in this thesis are smart contract developers in Ethereum, via the updated and modified security taxonomies provided, which could be used as a knowledge base for programming secure smart contracts, without being a security expert. The main stakeholders of the second approach are security tool developers and companies, via the proposed possible improvements within security tools in general. Also, smart contract developers become informed of the current state of security tools and can better understand whether they should completely rely on them.

The Ethereum community is generally positively affected by this work due to the promotion of a more secure and trustworthy environment. Last but not least, this project forms a solid foundation for further contributions in this area.

1.7 Thesis Outline

In order to properly navigate from a broader Ethereum security perspective to a more narrowed-down scope of security vulnerabilities in smart contracts, we provide a theoretical background of blockchain, Ethereum, and smart contracts. Additionally, we give an overview of the current related research work in security vulnerabilities, attacks/incidents, and available preventive security methodologies. Chapter 4 describes the methodology and justifications for the proposed taxonomies and the conducted experiment. Chapter 5 presents the two proposed taxonomies and one generated matrix. Chapter 6 describes parts of the experiment and presents the obtained results. Chapter 7 provides the answers to the research questions and a general discussion with a list of limitations. Lastly, Chapter 8 provides a summary of the thesis and a list of possible future work opportunities.

[11] Link: https://smartcontracts.smartdec.net/

2 Theoretical Background

The following sections contain a more detailed introduction of blockchain, Ethereum and smart contracts, which are of a prime importance for understanding specific security vulnerabilities and the overall results obtained from our experiment. However, if you are already extensively familiar with these terms, you may skip and jump to Chapter 3.

2.1 Blockchain

A blockchain is a shared ledger that records all transactions that have ever occurred and all the data exchanged since its creation [5]. With other words, as it is defined in [8], blockchain maintains a continuously-growing list of records, called **blocks**. Each block contains a number of transactions and it is chained to the previous block created, up to the first, *genesis* block. **Mining** represents the process of adding a block and verifying the validity of transactions (prevent double spending) through a Proof of Work (PoW) or other consensus protocols, such as Proof of Stake (PoS)[1]. Considering this design, blockchains are considered to be firmly repellent on altering the previous data stored in the blockchain. In addition, this ledger is replicated among many **nodes** in the network, which makes the blockchain technology decentralized and trustworthy in the sense that participants can send transactions securely without the need of a third party. It is believed that eliminating the third party control leads to lower processing fees, disposal of single point of failure (SPoF) issues and increased security of transactions [1].

To summarize, based on a survey by Zheng et al. [1], blockchain has the following **key characteristics**:

- *Decentralization* - in the sense of a peer-to-peer platform (no third party control)
- *Persistency* - in relation to the inability of deleting or altering (rollback) the transactions once they are recorded in the ledger
- *Anonimity* - based on the asymmetric cryptography and cryptographic hashing of blockchain data (transactions, digital cryptographic keys, etc)
- *Auditability* - of the global truth, in the sense of public availability of the blockchain data in order to verify and trace all previous transactions

[1] The PoS consensus algorithm is work in progress which will result in significantly faster block chains and also more transactions per second. [9]

Blockchains are closely related to the existence of cryptocurrencies based on their initial establishment which introduced the first secure generation of cryptocurrencies [5]. However, a blockchain can be created and function properly without the need of a cryptocurrency [10]. Therefore, nowadays we have different blockchain application domains and a variety of **blockchain taxonomies** [1, 10, 11]. A survey on blockchain application domains identifies five categories; **finance, security, IoT, public service** and **reputation systems** [1]. However, some of these application domains suffer from some serious challenges, also stated in [1], such as, **scalability, vulnerability, deficiency of existing consensus protocols, tendency for centralization**, and **privacy leakage**.

The most highlighted **blockchain taxonomy**, is based on the general **accessibility** of the blockchain network [10]. This categorization consists of **private** or permissioned network, and **public** or permission-less network. The most popular blockchain platforms, including here Bitcoin and Ethereum, are public and therefore anyone can serve as a node and transact and/or mine. On the other hand, private networks are mostly used by stakeholders/companies who wish to operate in a controlled, regulated environment, and distributed environment but still private, and as such, the blockchain applies specific filters to who can transact or mine on that network [10].

To conclude, there are many opportunities and use cases for blockchain technology to be applied in a variety of applications. Nevertheless, Ethereum presents the second generation of blockchain platforms, as the most popular one on building complex distributed applications beyond the cryptocurrency idea [5]. Therefore, the next section gives an overview of the Ethereum platform and also the rest of this thesis is only focused on Ethereum.

2.2 Ethereum

The main component that differentiates Ethereum from Bitcoin and other cryptocurrency-based platforms is that Ethereum does not only serve as a payment system but also as a computing platform[2]. Ethereum is referred to as *The World Computer*[3] and as the *future of internet*[4] powered by blockchain technology. These claims come as a result of its novel idea of distributed computational processing of applications, with full transparency governance and without any third party interference. It is also referred to as the "*next generation smart contract and decentralized application platform*" in Buterin's white paper which introduces Ethereum [12]. He introduces the main components and characteristics of Ethereum, starting from

[2]Link: https://www.coindesk.com/whats-big-idea-behind-ethereums-world-computer/

[3]The World Computer: https://www.youtube.com/watch?v=j23HnORQXvs

[4]Ethereum as the future of internet: https://svds.com/ethereum-the-rise-of-the-world-computer/

its overall architectural design, state, mining process, transactional method, continuing with its application on token systems, financial derivatives, decentralized autonomous organizations (DAOs) and finalizing with a collection of concerns and challenges [12].

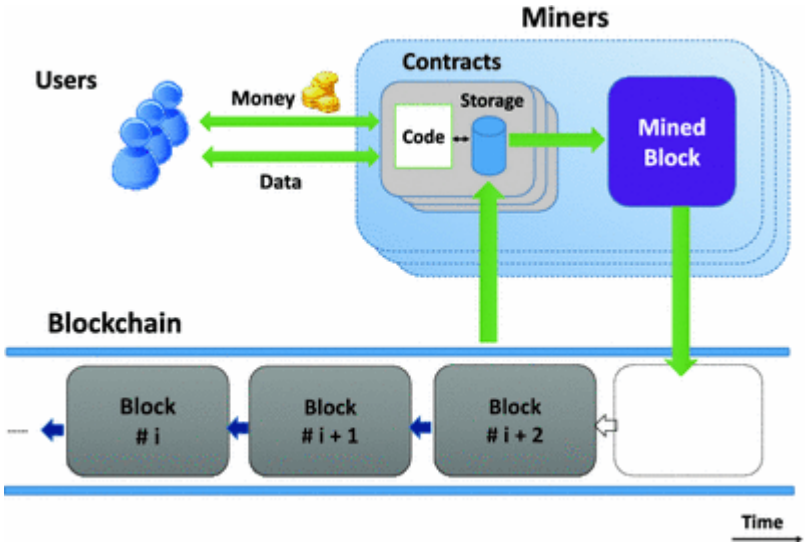

Figure 1: Schematic of a blockchain platform with smart contracts [13]

Figure 1 as illustrated by Delmolino et al. [13] represents a visualized schematic, generally for any decentralized cryptocurrency system with smart contracts, and specifically for Ethereum in this case. As we can see, smart contracts play a big role since they are represented as one of the main components in such a platform. In addition to sending money (in this case Ether), users can also send data/messages. Smart contracts have their own storage and act as an externally owned account in Ethereum. A more detailed explanation of smart contracts and how they operate is presented in Section 2.3. Additionally, blockchain stores the source code of a smart contract. More respectively, the code that is stored in blockchain is Ethereum Virtual Machine (EVM)[5] bytecode[6]. Except for this, the rest of the blockchain architecture is similar to any other cryptocurrency-based platform (non Turing-completeness platform).

Since these smart contract applications sometimes require heavy computational tasks, such as, contacting external services, storing data, mathematical/logical op-

[5]Ethereum Virtual Machine (EVM): "The virtual machine that forms the key part of the execution model for the bytecode (smart contracts' code)" [14] or with other words, the runtime environment for Ethereum contracts.

[6]EVM Bytecode: "The code that EVM can natively execute". [14]

erations, etc, and also since those computational tasks are processed from all the nodes (miners) repeatedly, the need for heavy computational power is inevitable. Therefore, in order to motivate and encourage miners to do all this computational work, and also to have control over DoS attacks, Ethereum introduces the **gas** usage. Each computational task requires an amount of gas in order to be executed. Gas is obtained by Ethereum's cryptocurrency, Ether, and once a block is created, the miner (or group of miners) is/are rewarded with Ether.

As in all blockchain-based systems, the **immutability** of the blockchain state in general, is heavily emphasized. Considering the blockchain design of chained blocks, the possibility of altering a section of the data, is practically impossible. But, there is a possibility of "disabling" a smart contract code, which frees the space in blockchain and the corresponding code is deleted [15]. The EVM instruction is named *SELFDESCTRUCT* and once it is called, the EVM will behave as the contract itself is "*dead*".

2.2.1 Two Ethereum Platforms

A significant attack on one of the biggest smart contracts ever deployed, The DAO, led to notable damage to the idea of immutability and Ethereum in general. Since The DAO attack had an enormous negative impact[7] on Ethereum and has a significant role for this thesis, it is further explained in detail in Section 3.2. However, simply said, a hacker was able to steal $50 million worth of Ether from a complex smart contract. Therefore, the Ethereum team and community decided to take action and to find a solution regarding this issue by initiating a **hard fork** in the Ethereum blockchain. The hard fork means that the new Ethereum chain will not be backward compatible and it will run as a completely separated branch from the old branch. By doing this, in the new branch they reverted the state of the blockchain to the state that it was before the attack and all the DAO tokens and Ether assets were forwarded back to the original owners. However, as we already know, in order to do such an enormous change in the blockchain, you have to gain consensus from more than 51% of the miners/nodes. A number of nodes/miners did not agree on the hard fork as a solution, with the justification that it will go against the main underlying philosophy of Ethereum (immutability and "code is law") [16], and decided to stick to the *old* branch.

As a result, the community was split in now two platforms, the new **Ethereum** branch and the old Ethereum branch now called **Ethereum Classic**. It is worth mentioning that these platforms now function as completely two separated platforms with their own cryptocurrency and community. According to a blog article [16], the main advantage of Ethereum Classic is that it stays true to the philos-

[7]The price of Ether plummeted from $20 to $13 right after the attack. Link: https://www.coindesk.com/understanding-dao-hack-journalists/

ophy of immutability, and the main disadvantages are that most "heavyweights" of Ethereum have moved to the new branch and that the old branch is known to be full of scammers. Whereas, Ethereum (the new branch), except for the disadvantage of not staying true to the notion of immutability, everything else, such as the exponential growth and the constant updates and modifications, are considered to be advantageous. [16]

2.3 Smart Contracts

So far we have continuously mentioned the concept of smart contracts and we briefly outlined what smart contracts are. However, this section will dive deeper into; smart contract application domains, how they operate, and what are the current challenges, solely in relation to the Ethereum platform.

Smart contracts are usually presented as **software applications** as for a more friendly/understandable term, but this term is rather more abstract, since smart contracts are more alike with the concept of **classes in object oriented programming** [2]. As it was mentioned previously, Ethereum was the first platform that introduced Turing-complete language in a blockchain platform, as such, the concept of smart contracts began to emerge in a practical sense. Smart contracts represent **self-autonomous** and **self-verifying** agents stored in the blockchain. They are composed by fields and functions [17]. Once they are deployed in the blockchain they have their unique address which the users/clients can use to interact with and it is referred to as **'contract account'** to differentiate from an **'external account'** which is controlled by public-private keys used by humans [18]. Also, the code that is stored in blockchain after deployment is a low-level stack-based bytecode (EVM code) representative of the high-level programming language (a JavaScript-like language) in which the smart contracts are initially written. As a result, it can be said that since the bytecode is publicly available from the blockchain, smart contracts' behaviour is completely predictable and its code can be inspected by every node in the network [10].

Furthermore, smart contracts are able to hold state, exchange digital assets, take user input, store data, obtain information from external services, and express business logic [19]. Once the smart contract code is deployed, its functions are triggered by messages and/or transactions sent to the smart contract address.

A smart contract should always be **deterministic**, meaning that the same input will always produce the same output [10]. Otherwise, considering that each task is executed on every node in the network, it will create a problematic state on reaching the consensus within the nodes, and therefore will deflect the entire process.

The smart contract that is illustrated in Figure 2 is written in Solidity, which

is the most popular high-level programming language since it is supported by the Ethereum developers and the overall community [20]. Solidity is similar to the JavaScript programming language which after deploying is compiled into EVM bytecode. Considering that the majority of popular smart contracts are written in Solidity, in this thesis we are only focusing on that one. However, the security issues and programming techniques mentioned throughout the thesis are applicable for other blockchain high-level programming languages as well, such as Serpent[8] (Python-like), LLL (based on Lisp), and Viper[9].

In Figure 2 we provide a simple example of a smart contract (a wallet) which is also used by other researchers to illustrate smart contracts [23, 17]. Up to version 0.4.0 there was no need to specify the compiler version on a smart contract. However, smart contracts that use version 0.4.0 and up to 0.5.0 must specify the compiler version[10] to avoid incompatibility. As shown in the example, a smart contract design correlates more with a class in OOP rather than with a software application with multiple classes and hierarchies.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4      mapping (address => uint) public inflow;
5
6      function AWallet(){ owner = msg.sender; }
7
8      function pay(uint amount, address recipient) returns (bool){
9          if (msg.sender != owner ||  msg.value != 0) throw;
10         if (amount > this.balance) return false;
11         outflow[recipient] += amount;
12         if (!recipient.send(amount)) throw;
13         return true;
14     }
15
16     function(){ inflow[msg.sender] += msg.value; }
17 }
```

Figure 2: A simple example of a smart contract [17]

The function **AWallet** in line 6 is the constructor which only runs once when the contract is created, and it assigns the sender as the owner of the contract. Function **pay** is used to send a specific amount of Ether from the contract to a recipient. In line 9 the function checks if the owner is the sender or if some Ether is attached to the invocation and transferred to the contract. The function in line

[8]Deprecated in 2017. Considered to be of a very low quality, untested and overall not safe. [21]
[9]Viper is considered to be the successor to Serpent, and it is under active development. [22]
[10]The compiler version is used as follows: e.g. pragma solidity ^0.4.0;

10 checks if the required amount of Ether is unavailable. And the last check is in line 12 which checks if the *send* succeeded. Otherwise, if one of these checks fails, the contract will throw an exception, the changes will be reverted, and the Ether is returned to the caller. Line 11 updates the outflow registry before transferring Ether to the recipient. Lastly, the function in line 16 will be triggered when the contract is receiving Ether and no other function is invoked. In such a case, the inflow registry will update and also *this.balance* will be automatically updated in any case (sending or receiving). [23, 17]

Each invocation or computation of a smart contract's function demands an execution fee which is defined in terms of gas (gas price) and acquired with Ether. The collected gas serves as an incentive for miners/nodes to run these computational tasks on their computer[11], but also protects the platform from DoS attacks[12].

To conclude, smart contracts also gave rise to the concept of **Decentralized Autonomous Organizations (DAOs)** which represent an advanced version of smart contracts where their behaviour could be altered if certain conditions are met [10].

2.3.1 Application Domains

Since we are presenting smart contracts as powerful agents and very promising applications, this section will cover the basis of possible use cases of SCs and their categorization. There are several attempts to categorize the application domains of smart contracts [24, 12, 9, 5]. The initial categorization is done by Buterin (founder of Ethereum) in [12], based on three top levels:

- **Financial applications**
  - Sub-currencies, financial derivatives, hedging contracts, saving wallets, or full-scale employment contracts
- **Semi-financial applications**[13]
  - Self-enforcing bounties, cloud computing, gambling
- **Non-financial applications**
  - Online voting, decentralized governance, prediction markets

This categorization is general and tries to cover all areas in which smart contract applications could be instantiated.

Moreover, based on an empirical analysis of all currently-deployed smart con-

[11] Miners are rewarded with Ether each time a block is created.

[12] An adversary can attempt to slow down the network by invoking a time-consuming function. In order to succeed, the adversary will need to allocate a large amount of gas (with other words, ether) which makes the attack really expensive and almost impossible.

[13] "Money is involved but there is also a heavy non-monetary side to what is being done". [12]

tracts in Bitcoin[14] and Ethereum, Bartoletti and Pompianu [24] propose a **taxonomy of smart contract application domains** with five categories, including here; **financial, notary, game, wallet**, and **library**. Based on their results, the most prominent (higher number) smart contracts are within the financial category, followed by game category and notary.

Further research on smart contract domain applications is conducted by Alharby and Moorsel in [5]. They categorize the possible use cases of smart contracts in; **Internet of Things** and **smart property**, **music rights management** and **e-commerce**. Clack et al. [25] highlight only the use of smart contracts as **legally-enforceable templates and agreements** based in legal documents. Whereas, other authors focused only on smart contracts for **Internet of Things (IoT)** usage [10]. They identified the opportunities that smart contracts offer for the IoT area and also they emphasized the current challenges and limitations that should be considered before deploying a blockchain network in an IoT setting [10]. Additionally, another interesting piece of work is that by Juels et al. [26] which investigated the use of smart contracts for **criminal activities**, such as money laundering, marketplaces for illicit goods and, ransomware.

We argue that some of the categorizations are too broad and some are too specific (narrowed down), i.e. assessing only one application domain. We are not stating that they should not be used nor that their approach is wrong. However, the application domain of smart contracts is a disputable topic, thus we propose further work on categorizing the already-existing SCs on Ethereum.

Despite the fact that Ethereum and smart contracts can be extensively applied in many areas, they suffer from some serious challenges and limitations as a result of being a relatively new technology and taking a highly experimental approach in general. Therefore, this project will particularly assess the security issues of Ethereum smart contracts.

[14]Bitcoin supports a low-level of smart contract applications as well. However, the programming language used in Bitcoin is not Turing-complete and therefore the overall potential is heavily limited.

3 Related Work

This chapter consists of three sections; security vulnerabilities, smart contract attacks/incidents, and preventive methodologies. Gathered via a thorough research[1] of the relevant work and state of the art, it represents the base knowledge which is used to form the taxonomies proposed in Chapter 5.

3.1 Security Vulnerabilities

In general, this section covers the related work on research papers and other articles that analyze/identify and provide an overview of security vulnerabilities in smart contracts. Moreover, it is explaining each vulnerability, in order to have a better understanding of the attacks and incidents mentioned in Section 3.2.

Despite the fact that Ethereum and the concept of smart contract is relatively new, there is sufficient work on the security aspect, which is also stated by Alharby and Moorsel in [5], where two thirds of the papers examined, focus on tackling smart contract issues. Some authors are focused more in providing a taxonomy and a state of the art, in general for security vulnerabilities [23, 7, 5, 13, 27]. Whereas, some others are focused more in specific vulnerabilities and smart contract challenges, such as privacy [9, 28]. There are also papers focused only in one vulnerability and try to develop or propose a solution, for example, in timed commitments [29] and smart contract altering possibilities [30].

3.1.1 General Research on Vulnerabilities

This section provides an overview of the research done generally on smart contract vulnerabilities. Even though at this point, some of the vulnerabilities might appear abstract for the reader, later on, in this section we explain them in detail. Thus, the point of this subsection is to gain an overview of how vulnerabilities are currently classified/grouped in general.

For our proposed taxonomy, we chose to partially[2] base our taxonomy on that provided by Atzei et al. [23], it being the most prominent one[3]. Table 1 illustrates the taxonomy proposed in [23]. The levels they chose to represent the vulnerabilities are: Solidity, EVM, and Blockchain. In addition, for most cases of vulnerabilities they provide an attack example, and the numbers in 'Attacks' column, represent the

[1] The methodology on how we conducted the literature search is described in Section 4.1.

[2] Meaning that we chose to use the same categorization based on three levels, in addition to the severity level, without including the attacks category.

[3] Section 4.2 in Methodology chapter justifies the selection of this and other taxonomies.

sections where the attacks are discussed in their paper. *'Solidity'* vulnerabilities are also applicable for other high-level programming languages in Ethereum. This taxonomy seems to properly classify all vulnerabilities based on their level, since a newly discovered vulnerability definitely falls in one of these categories. Also, it represents a well-defined base knowledge for future work studies which can have a more narrowed-down research.

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	4.1
	Gasless send	4.2
	Exception disorders	4.2, 4.5
	Type casts	-
	Reentrancy	4.1
	Keeping secrets	4.3
EVM	Immutable bugs	4.4, 4.5
	Ether lost in trasfer	-
	Stack size limit	4.5
Blockchain	Unpredictable state	4.5, 4.6
	Generating randomness	-
	Time constraints	4.5

Table 1: Taxonomy of vulnerabilities in Ethereum smart contracts [23]

Another taxonomy is provided by Alharby and Moorsel [5] based on a systematic mapping study of current research topics related to smart contracts. In their study, they discovered four key smart contract issues; **codifying issues, security issues, privacy issues**, and **performance issues**. In addition to categorizing all vulnerabilities they identified, based on one of the four aforementioned groups, they also provide an extra category on *proposed solutions* with references to the corresponding solution-proposals papers. Some solutions are proposed via the aid of security analysis tools which are explained in detail in the next Section, 3.3. By **codifying issues** they refer to the challenges/mistakes that are related with the development of smart contracts. **Security issues** mean bugs or vulnerabilities, and by **privacy issues** they refer to the issues related to unintentional information disclosure to the public. Lastly, **performance issues** are related to the challenges that affect the ability of blockchain to scale. [5]

Based on this taxonomy, we argue that sometimes it is difficult to distinguish the aforementioned categories and fit one particular vulnerability correctly in one class. Meaning that, many *codifying issues* caused by developers lead to security issues and privacy issues as well. For example, one of their security issues is 'mishandled exception', which we argue that is raised more as a codifying issue rather than a security issue of the platform itself. Also, the 're-entrancy vulnerability' identified

as a security issue by [5], appears as a result of developers' mistakes (codifying issues) on improperly handling external calls. Therefore, we argue that this taxonomy design appears to be somehow representative for classifying papers, but not for a general taxonomy of vulnerabilities in smart contracts.

Other research papers and articles refer to security vulnerabilities in general, without any categorization, such as, in [7], where they discussed several severe vulnerabilities. In [27], where Buterin with the community's help created a crowdsourced list of the major bugs with smart contracts, and in [13] through a university course for smart contract programming, they exposed numerous common pitfalls and vulnerabilities.

3.1.2 Research on Specific Vulnerabilities

In addition to the general research on vulnerabilities, there is also research work and articles focusing in one vulnerability. The first one is research work focusing on the **privacy preserving** issue of smart contracts. As it is discussed later on, we will see that the privacy issue is a crucial aspect of smart contract security. It represents a real challenge for developers to keep critical functions/methods secret, apply cryptography, and avoid disclosing data that should not have been public in the first place. A research on 'replacing paper contracts with Ethereum smart contracts' finds out what kind of criteria Ethereum needs to fulfill to be properly applied on replacing paper contracts [9]. They conclude that due to a large privacy setback it is not *yet* recommended to replace legally-enforceable agreements with smart contract applications [9]. This is as a result of the private information these papers (agreements) hold and the damage that could be done if they become public or if the blockchain does not work as intended on preserving privacy. According to [5], *lack of transactional privacy* and *lack of data feeds privacy* are two issues correlated with the privacy preserving category.

A similar research on the issue of **privacy-preserving** is conducted by Kosba et al. [28]. They highlighted the significant importance of privacy in smart contract applications generally in blockchain technologies, not only in Ethereum. For a solution to this issue, they proposed a decentralized smart contract system, Hawk, which does not store financial transactions in the blockchain and saves the developers on implementing any cryptographic functionality [28]. Juels et al. [26], also investigated the leakage of confidential information and theft of cryptographic keys for smart contracts used in criminal activities.

One of the most prominent vulnerabilities of Ethereum is considered to be the **timestamp dependency**. Therefore, Boneh and Naor [29], introduce and construct *timed commitment schemes* which are proposed as a solution for this vulnerability. Their proposed solution could be applied when two mutually suspicious parties

wish to exchange signatures on a contract [29].

Another issue that is being tackled as a single vulnerability is the **gas-costly pattern**, more specifically, the under-optimized smart contracts that consume more gas than necessary. A research investigation in this regard is done by Chen et al. [31], in which they identified 7 gas costly patterns and grouped them into two categories. They also developed a tool, named Gasper, focused only on identifying gas-costly patterns by analyzing the smart contracts' bytecode [31]. Their results indicate that over 80% of 4240 smart contracts analyzed, suffer from one of the gas-costly patterns [31].

Additional specific research on a particular issue is done on the challenge of communicating with **external services** (Oracles[4]). Zhang et al. [32], presented an authenticated data feed system called Town Crier, which enables smart contracts to consume data from outside the blockchain while preserving confidentiality with encrypted parameters.

To conclude, there is a significant number of research on specific issues in addition to the general (overview) research on smart contract vulnerabilities. However, the solutions proposed usually require for blockchain upgrades, meaning that all the nodes have to upgrade their version in order to solve a particular issue, or they are proposed as a separate solution/platform on top of blockchain. This makes it challenging in regards to obeying all nodes in the network to upgrade, what we briefly mentioned before, the issue with the hard fork, there will always be network participants not agreeing on something. Therefore, we can argue that a more effective way of identifying vulnerabilities and preventing vulnerable smart contracts being deployed in Ethereum, is through security code analysis tools, which are thoroughly examined in Section 3.3.

3.1.3 Vulnerabilities Explained

This section provides a brief explanation for each one of the security vulnerabilities that are mentioned throughout this thesis. Some vulnerabilities based on their naming convention and the already gained knowledge for blockchain and Ethereum are self-explanatory. Therefore, those vulnerabilities are excluded from the list and the main focus of this section is (mostly) within the severe vulnerabilities.

Reentrancy is considered to be the most severe vulnerability for a smart contract, based on the biggest attack ever made[5]. The reentrancy vulnerability is explained as follows:

> "Any interaction from a contract (A) with another contract (B) and any transfer

[4]Oracle: A reliable connection between Web APIs and smart contracts, since smart contracts cannot fetch external data on their own. Link: `http://www.oraclize.it/`

[5]TheDAO - Explained in section 3.2

> of Ether, hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed." [33]

Doing so, contract (B) can retrieve multiple refunds and empty the balance of contract (A). The use of *checks-effects-interactions*[6] is recommended as a solution to avoid this vulnerability.

Tx.origin usage is strictly prohibited to use for authorization. Using it, will result in draining all contract funds, because the *tx.origin* sets the attacker's address as the owner of the contract. As such, the attacker obtains full access to the smart contract funds.

Callstack depth makes an external call to fail because it exceeds the maximum call stack of 1024 [33]. As a result, the call will fail, and if the exception is not properly handled by the contract, the attacker can force the contract to produce an output or result which suits them.

Timestamp dependence presents a common vulnerability favoring a malicious miner. If a contract is using, e.g. *now, StartTime, EndTime* based on the time stamp of the block, that means that the miner can manipulate the timestamp for a few seconds by changing the output to be for his favor [34]. However, this vulnerability is severe only if used in critical components of a contract.

Transaction-ordering dependence refers to the idea that the user can never be sure of the order of transactions. An example can be a smart contract which offers a reward for a solved puzzle. Once a user solves the puzzle and submits the transaction, at the same moment the smart contract owner can reduce (or completely remove) the reward. There is a probability that the transaction that reduces or removes the reward is processed first. In this case, the owner gets an answer for the puzzle, and the solver (user) does not get the reward.

The use of **external calls** is considered to be by default risky [34], because adversaries can execute malicious code in that external contract. Therefore, it is recommended to possibly avoid external calls ("calls to the unknown") in general or treat those calls as potential risk and take precautions, such as, use *send* instead of *call_value()*, favor *pull* over *push* for external calls, and handle errors (check the return value) [34].

Unchecked-send bug is part of the **exception disorders** or **mishandled exceptions**. From some authors is referred to as "*send* instead of *transfer*, because '*transfer*' automatically checks for the return value, whereas using '*send*' you have to manually check for the return value, and throw an exception if the send fails. Not doing so, can lead to an attacker executing malicious code into the contract and drain its balance. Overall, the consequences are similar to the **reentrancy** and

[6] First subtract the value from the contracts' balance then send the Ether, and check for the return value.

call to the unknown vulnerability, as are the precautions as well.

DoS is explained by SmartCheck[7] as follows:

> "A conditional statement *(if , for , while)* should not depend on an external call: the callee may permanently fail *(throw or revert)*, preventing the caller from completing the execution." [35]

An attacker can cause inconvenience by supplying the contract with data that is expensive to process, thereby preventing others to interact with it. This vulnerability is closely related to the **external calls** vulnerability and to prevent this from happening, we need to handle properly any *throw* exceptions from external calls, and also, avoid looping behaviour.

Blockhash usage similarly to the block timestamp, it is not recommended to be used on crucial components, for the same reason as with the timestamp dependency, because the miners, to some degree, can manipulate it and change the output to their favor.

Gasless send makes a transaction to fail if not enough gas is provided for a specific call. The maximum gas limit on the network can vary over time based on the transaction fees[8]. It is important to throw an exception if a failure based on the gas consumption happens. Also, it is important to develop functions that do not require too much gas, not only for the purpose of failing because not enough gas is provided, but also for the sole purpose of mitigating the expensive payment fees.

Other vulnerabilities include; **immutable bugs** (e.g. wrong constructor name) which refer to a bug or a code mistake which cannot be altered after deployment/discovery, **the use of untrustworthy data feeds**, **failure to keep secrets** or in other words failure to apply cryptography and as a result expose crucial functions or values, the **challenge to generate randomness**, **style guide violation**, etc.

Generally, many of these vulnerabilities have to do with the novel approach of building smart contract applications in a public blockchain. Since, so far, developers are used to a more centralized way of developing applications, in which they did not have to worry about many of these issues, because the centralized system provides certain guarantees that a public blockchain cannot[9].

To conclude, it is important, for this thesis, to have a general understanding regarding these issues, and not a detailed conception on how they happen and how can they be prevented. Because, almost every attack used a different approach on exploiting these vulnerabilities.

[7] SmartCheck: https://smartcontracts.smartdec.net/
[8] Link:http://www.kingoftheether.com/contract-safety-checklist.html
[9] Link: https://goo.gl/KBjVbH

3.2 Smart Contract Attacks and Incidents

This section provides an overview of the most prominent attacks that have happened since the existence of Ethereum, in conjunction with the vulnerability that triggered the attack, and the overall damage that was caused.

TheDAO

TheDAO is briefly mentioned before in Section 2.2 because of the impact that it had on Ethereum. TheDAO attack is the reason why Ethereum had the hard fork and is split into two platforms[10]. TheDAO is an abbreviation for "The Decentralized Autonomous Organization" and is represented as a complex smart contract which was considered to have revolutionized Ethereum forever [16]. TheDAO was not owned by anyone, and it previously worked as follows:

- A group of people developed the smart contract(s) that will run the organization.
- An initial funding period, where you could buy DAO Tokens (ICO[11]) that represent ownership (the right to vote).
- Once the funding is over, developers make smart contract proposals, and the users that own DAO tokens vote on approving and funding these proposals.

TheDAO was able to raise a record of $150 million in Ether, and its popularity quickly emerged. Anyway, TheDAO had the option for token holders to opt-out from the organization and get back the money they invested, if for example, they did not endorse a smart contract that was going to get funded. This was done through a *split function*, which ones is triggered, the user would automatically opt-out from the organization and get their money back[12]. The user could also choose to create a *child DAO* and anyone else that did not agree on funding (endorsing) a specific proposal, could join the child DAO created. This design made TheDAO popular by offering **flexibility, full control** and **complete transparency**. However, this design was the reason that a hacker exploited a **reentrancy** vulnerability in the "split DAO" function. The attacker was able to steal one third of DAO's funds (3.6 million Ether) which was considered to be around $60 million at that time[13]. The consequences that this attack had on Ethereum were unprecedented. The price of Ether dropped from $20 to $13 and the actions taken *demolished* (harmed) the notion of immutability.

[10] Ethereum and Ethereum Classic.
[11] ICO: Initial Coin Offering
[12] After splitting off from the DAO you had to hold on to your Ether for 28 days before you could spend them. [16]
[13] As of November, 2017, that price would be around $1.3 billion.

King of the Ether Throne (KotET)

KotET is an example of a contract account that experienced a type-of-incident rather than an attack. The normal operation of KotET, as explained on their website is as follows:

> - Suppose the current claim price for the throne is 10 Ether. You want to be King/Queen, so you send 10 Ether to the contract.
> - The contract sends your 10 Ether (less a 1% commission) to the previous King/Queen, as a "compensation payment". And the contract makes you the new King/Queen of the Ether Throne. The new claim price for the throne goes up by 50%, to 15 Ether in this case.
> - If an usurper comes along who is willing to pay 15 Ether, they depose you and become King/Queen, and you receive their payment of 15 Ether as your "compensation payment". [36]

The incident happened once KotET tried to send a reward in a contract-based address. KotET allocated a small amount of gas for this transaction, and as a result the transaction failed. Additionally, it means that the exception was not handled properly by the contract, and therefore the contract proceeded in making a new "King" of the game, even though the compensation was not sent to the previous one. **Gasless send** and **mishandled exception** were the vulnerabilities that caused this incident.

GovernMental

GovernMental refers to a smart contract educational game which simulates the finances of a government. It is also refered as Ponzi GovernMental (Ponzi scheme). It consists of four rules, which are explained in [37]. And because they represent a common real-life ponzi scheme, all rules are not presented here. However, the game was played by "lending" the government money (Ether), and they promised to pay it back (+10% interest), if the system continuously runs (receives Ether). This smart contract suffered from an *'unintentional'* mistake which led to an incident where the jackpot payout of 1100 Ether was stuck because the transaction which needed to reward (pay-out) the jackpot, required too much gas[14]. However, there are discussions about whether this mistake was intentional which is difficult to conclude because the money is stuck and technically, no one benefits from it being stuck in an account[15]. It can be said that GovernMental overall, suffered from **gas costly pattern**, **mishandled exception**, and also from the **unpredictable state** vulnerability.

[14] 5057945 amount of gas, whereas the current maximum gas amount for a transaction was 4712388 gas.

[15] Reddit Discussion: https://goo.gl/cbiCG4

EtherPot

EtherPot is a decentralized, autonomous and provably fair lottery system. This smart contract suffered from the **unchecked-send call**, in which a call silently failed because the code itself did not handle the exception properly. Also, it suffered from the **blockhash usage** vulnerability, because it used the blockhash of the blocks to determine the winner of the lottery. So far, we only know that this contract is not in use anymore and we have no knowledge of the damage that was caused or the attack that occurred. Based on a community discussion on their Github repository[16], it may have been that the developers realized the bug before the attack. The only source is their website which is partially broken[17] and an analysis which discovered this vulnerable smart contract[18].

SmartBillions

Another similar example to EtherPot is a very recent[19] smart contract application. SmartBillions presents a fully decentralized and transparent lottery system. The development team was so confident of its security, that they challenged everyone in the network to compromise the smart contract by adding a reward of 1500 Ether ($450,000 on that time) for anyone who would be able to hack the smart contract [38]. This challenge quickly backfired, as an attack on the contract happened and the attacker was able to drain 400 Ether from the total reward of 1500, before the owners of the contract pulled out the remaining of the funds [38]. The contract used **blockhash** in one of their lottery functions to determine if the user wins or loses. The attacker successfully managed to manipulate the blockhash of that function twice, and force the result in his favor.

TheRun

TheRun, a lottery based smart contract, suffers from the **timestamp dependence** vulnerability. The contract, as discovered by [39], uses the blockchain timestamp to generate random numbers and based on that to reward a jackpot. As we can see, the contract suffers not only from the timestamp dependence but also from the challenge of **generating randomness** in blockchain. As we have already discussed, an adversary can manipulate the timestamp and benefit from the result (win the jackpot).

There are many other cases of attacks and incidents which we have used for our experiment and are omitted here to conserve space. However, we can clearly see that a vulnerable smart contract can lead to losses of millions of dollars and other

[16]Link: https://github.com/etherpot/contract/issues/1
[17]Link: https://etherpot.github.io/
[18]Link: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/
[19]October, 2017

obstacles in general for Ethereum. These examples that we mentioned, further emphasize the need for an increased security in general on Ethereum and also highlight the importance of this thesis, respectively the importance of research and development work on security code analysis tools.

3.3 Preventive Methodologies

This section covers the basis on different ways (mostly tools) for preventing from vulnerable smart contracts, with an emphasis on the security tools which will be used for our experiment[20].

3.3.1 Preventive Security Methods

ZeppelinOS is an operating system for smart contract applications developed by Zeppelin Solutions [40]. As referred to by Zeppelin Solutions, ZeppelinOS is "an open-source, distributed platform of tools and services on top of the EVM to develop and manage smart contract applications securely" [40]. Its prime focus is the advancement of smart contracts security. Their system is composed of four components, *kernel, scheduler, marketplace*, and *off-chain tools*. In other words, Zeppelin introduces a novel approach in developing smart contracts by using already developed and secure smart contracts (i.e. libraries). Doing so, presumably will lead to mitigating severe vulnerabilities which are related to programming mistakes. Furthermore, the off-chain component provides numerous tools like debugging, testing, deployment and monitoring. Based on their team, these tools will enhance the development process (better, easier, robust), and generally to a more secure smart contract environment.

SolCover[21] provides code coverage for Solidity testing. Relying on code coverage, SolCover measures and describes the degree of overall testing in a smart contract. Even though, it does not serve as a mechanism to identify specific vulnerabilities, it could be argued that it creates a more secure environment with the philosophy that *more tests = more secure*.

HackThisContract [22] is a crowdsourcing experimental website that encourages developers to test their smart contracts before deployment by uploading it on their website. Other developers, with their own techniques, will try and exploit possible vulnerabilities. Additionally, they provide a list of vulnerable smart contract examples which the developers should not follow. Overall, with the sole purpose of deploying secure smart contracts and mitigate (eliminate) severe issues in a pre-deployment phase.

[20] Section 4.3 provides a detailed explanation of the methodology used to select the tools for our experiment.
[21] Link: https://github.com/sc-forks/solidity-coverage
[22] Link: http://hackthiscontract.io/

Security audits are considered to be the most effective way of identifying vulnerabilities in a pre-deployment phase. Experienced blockchain developers, and specialized teams, carefully investigate the smart contract manually and automatically to identify possible vulnerabilities, and make sure that it follows best programming practices. Despite the fact that it might be the most secure method for preventing deployment of vulnerable smart contracts, it is not considered to be popular because of the high price range that security audit firms have[23]. Currently there are many firms that do smart contract security audits, including here, Zeppelin, Solidified[24], SmartDec[25], and DejaVu[26]. Basically, they review a smart contract code, and provide a report with the issues/vulnerabilities they found, based on their severity level and the potential risk. In addition, they provide overall recommendations to increase security. Since for our experiment we analyze audited smart contracts, they are furthermore explained in Chapter 4.

Other preventive methodologies include staying up-to-date with Ethereum upgrades and especially with the attacks that happen over time, since they may discover a new vulnerability. Also, it is of a vital importance to follow a list of recommendations for secure smart contracts once you start developing, such as the extensive list by ConsenSys [41].

The above-mentioned preventive methodologies present a variety of means for a more secure smart contract. However, some of them provide low-level of a guarantee (crowd-sourcing), and some are quite expensive (security audits). Thus, it is recommended that, if utilized, they should serve as an extra aid to mitigate the deployment of vulnerable SCs and developers should not completely rely on them.

3.3.2 Security Tools

In this section, and for the experiment conducted, we cover only self-contained security tools which could be utilized in a (not only) pre-deployment phase. If these tools work as intended, they could serve as a solid protection against attacks/hacks.

Oyente

Oyente is known to be the first and most popular security analysis tool based on the research conducted and based on the Ethereum community[27]. It was developed by Luu et al. [7] and is one of the few tools presented in a major security conference,

[23] Based on a community discussion in Reddit, a smart contract security audit costs between $20k-$60k. Link: https://www.reddit.com/r/ethdev/comments/6pdgvd/how_much_does_a_smart_contract_audit_cost/

[24] Link: https://solidified.io/

[25] Link: https://smartcontracts.smartdec.net/

[26] Link: http://www.dejavusecurity.com/services/

[27] This tool is referenced by many papers and also throughout our search for security analysis tools, many blockchain developers and people from the Ethereum community (Reddit) suggested Oyente.

Ethereum Devcon[28]. Oyente leverages symbolic execution to find potential security vulnerabilities, including here **transaction-ordering dependence, timestamp dependence, mishandled exceptions** and **reentrancy**. The tool can analyze both Solidity, and the bytecode of a smart contract. In its early stage it could have been used only through a command line interface, later on, they developed a web-based interface which seemed to be more user friendly. It is worth mentioning that it is the only tool that describes its verification method to eliminate false positives [7].

Securify

Securify[29] is a web-based security analysis tool and based on their website they state that it is the first security analysis tool that provides **automation** (to enable everyone to verify smart contracts), **guarantees** (for finding specific vulnerabilities), and **extensibility** (to capture any newly discovered vulnerability). Securify uses formal verification but also relies on static analysis checks. It is in its beta version and the security issues that it covers are: **transaction reordering, recursive calls, insecure coding patterns, unexpected ether flows**, and **use of untrusted input**. However, the recursive calls, unexpected ether flows, and part of the insecure coding patterns checks are locked (require full access) for the time being[30]. Securify, in addition to analyzing the bytecode and Solidity, it can analyze a smart contract through its contract address.

Remix

Remix[31] is a web-based IDE that allows to write Solidity smart contracts, deploy and run them. A debugger and a testing environment (test-blockchain network) are integrated. Additionally, it serves as a security tool by analyzing the Solidity code only, to reduce coding mistakes and identify potential vulnerable coding patterns. Some of the vulnerabilities that it identifies are: **tx.origin usage, timestamp dependence, blockhash usage, gas costly patterns, check effects (reentrancy)**, etc[32]. Remix security analysis rely on formal verification (deductive program verification, theorem provers).

SmartCheck

SmartCheck[33] is also a web-based security code analysis tool provided by SmartDec team[34]. SmartDec is a company focused on security audits, analysis tools and web development. Recently (November, 2017), they released a beta version of their se-

[28] Link: https://www.youtube.com/watch?v=bCvh6ED-cj0
[29] Link: https://securify.ch/
[30] As of October, 2017
[31] Link: https://remix.ethereum.org/
[32] Link: https://remix.readthedocs.io/en/latest/analysis_tab.html
[33] Link: http://tool.smartdec.net
[34] Link: https://smartcontracts.smartdec.net/

curity tool, SmartCheck. It automatically checks for vulnerabilities and bad coding practises. In addition to that, it highlights the vulnerability (e.g. line of code), gives an explanation of the vulnerability, and a possible solution to avoid a particular security issue. Their analysis run only for Solidity code and it is not stated which specific methodology they use to identify the vulnerabilities (e.g. symbolic execution, formal verification, etc.). Each vulnerability discovered is shown in correlation with its severity level. Some of the severe vulnerabilities they identify are: **DoS by external contract, gas costly patterns, locked money, reentrancy, timestamp dependency, tx.origin usage, and unchecked external call**. Additionally, SmartCheck identifies many other vulnerabilities with low severity (warnings), such as, compiler version not fixed, style guide violation, and redundant functions.

F* Framework

"F*" presents a framework for analyzing the runtime safety and the functional correctness of Ethereum smart contracts, outlined by Bhargavan et al. [42]. It relies on formal verification, by translating Solidity or bytecode into F* (a functional programming language) and then identifying potential vulnerabilities, such as, **reentrancy** and **exception disorders** (mishandled exceptions).

Mythril

Mythril is a recently released experimental security analysis tool which is under heavy development[35]. Through a command line interface, it is able to analyze bytecode, and by installing solc (command line compiler) it also analyses Solidity code. Mythril relies on concolic analysis (symbolic execution). So far, it is able to identify a variety of vulnerabilities, such as, **unprotected functions, reentrancy, integer overflow/underflow,** and **tx.origin usage**. Some other severe vulnerability checks are presented as work in progress, such as, timestamp dependence, transaction-ordering dependence, and information exposure[36].

Gasper

Gasper is a security tool developed by Chen et al. [31], which is not released yet. However, from their research paper, we already know that it is focused only on identifying **gas costly programming patterns** in a smart contract through a command line interface. It runs analysis only for the bytecode. Moreover, they have discovered seven gas costly patterns, and grouped them into two categories. Gasper also relies on symbolic execution to cover all reachable code-blocks by disassembling its bytecode using *disasm* (disassembler). So far, they only cover the gas costly patterns from the first category that they have discovered, the rest is work-in-progress. [31]

[35]Link: https://github.com/b-mueller/mythril/
[36]Link: https://github.com/b-mueller/mythril/blob/master/security_checks.md

4 Methodology

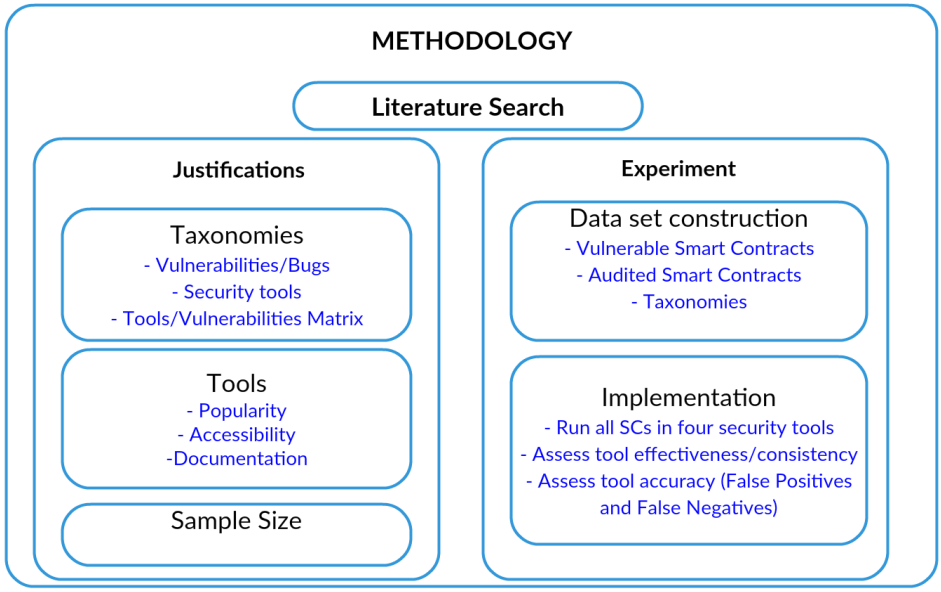

Figure 3: Methodology Overview

Figure 3 illustrates an overview of the methodology process of this work which is fully explained in this chapter. The emphasis is on the literature search, justifications, and the experiment. The chapter begins with the **literature search** with the purpose of explaining how we gathered the literature used in this thesis. We proceed with a justification for the taxonomies by reasoning about the taxonomy designs we propose as well as a justification for the tools by explaining the criteria that we use to select the tools which are further analyzed. Section 4.4, **Data set construction** explains the process of how we gathered the vulnerable and audited smart contracts, as well as the process of how we constructed the proposed taxonomies. Section 4.5, **Implementation** describes the experiment and the data analysis process. Simply said, the experiment consists of assessing the tools based on their:

- **Effectiveness** - check how many smart contract applications the tools were

able to analyze from the whole data set
- **Accuracy** - assessing the correctness of the results they produce, based on false positive and false negative rates
- **Consistency** - assessed in security tools that analyze both bytecode and Solidity, and check if there is any inconsistency[1]

4.1 Literature Search

A thorough literature search is conducted for this thesis. Since Ethereum is a relatively new platform, a comprehensive search is conducted on online websites as well, with a particular focus on blockchain, Ethereum, smart contracts, and their security implications. The research for peer-reviewed related work on this topic is conducted on several online databases, such as, Google Scholar, IEEE, ScienceDirect, ACM DL, and Scopus. Except for specific keywords used from Section 1.2, other filters are not used. Many research papers led to other references and some authors were helpful by providing us with additional related works.

As for the search on online websites, it was done continuously from February up until October, 2017. We stayed up-to-date with any related web-articles or blogs that were assessing smart contract security issues. Additionally, the Ethereum community discussion platforms, such as Reddit[2] were of big help for providing information on the existing security tools and especially immediate information/announcements on smart contract hacks/incidents. Furthermore, once the research was conducted, group chats (Slack channels, Gitter) and e-mails were used to communicate with developers or users for a specific security tool.

The literature search represents an important role for this thesis, because the first approach on proposing taxonomies is heavily influenced by it. Additionally, the second approach and the experiment are initially established based upon the research work as well.

4.2 Justification for Taxonomies

Chapter 5 proposes novel taxonomies which are based on the research we conducted. The first taxonomy 5.1 is an updated and slightly modified version of the one proposed by Atzei et al. [23], illustrated in Table 1. The main reason that we chose this taxonomy design is because of its popularity, meaning that it is referenced by other researches [43, 24, 31], as well as the Ethereum community on Reddit which suggested this one as the most reliable one when asked. Furthermore, as we already discussed in Chapter 3, this taxonomy seems to be more efficient on

[1] For example, if a tool produces some results with the bytecode of a smart contract, and other with the Solidity of the same contract.

[2] Link: https://www.reddit.com/r/ethereum/ is a place for open discussion about the Ethereum software, protocol, distributed applications, and related technologies.

categorizing smart contract issues, compared to other taxonomies, such as the one proposed by Alharby and Moorsel in [5]. The modification we made to the taxonomy is that instead of having an attack example for each vulnerability, such as in Table 1, we have a replacement category of the severity level for each vulnerability, based on the harm they can cause to a smart contract.

The second taxonomy, to the best of our knowledge, is a novel taxonomy. It also represents the current state of Ethereum security code analysis tools that are already up and running. It represents a necessity to proceed with our experiment (security tool analysis), since it allows us to make proper choices for the selection of the tools. The third and last categorization is also a new one in this field. It represents a matrix of the security tools versus the vulnerabilities that they cover. This matrix allows us to make proposals for future possible improvements within the security tools.

To conclude, the general purpose of the proposed taxonomies is for smart contract developers to use them. Using these updated and comprehensive taxonomies, they could familiarize themselves with the currently known vulnerabilities and try to mitigate the same.

Taxonomies are presented and furthermore discussed in Chapter 5.

4.3 Justification for Selected Tools

Our second approach (the experiment), is analyzing several security code analysis tools based on their effectiveness, accuracy and consistency. However, consistency is assessed partially because only two out of four tools analyze both bytecode and Solidity. Throughout our literature and websites search we came up with a list of security tools which are further explained in Section 3.3.2 and overall visualized through a taxonomy that we mentioned in the previous section. This aided us in choosing the tools based on three criteria; **popularity, accessibility** and **documentation**. **Popularity** is assessed through the research work and the universal recognition of those tools from the Ethereum community and especially blockchain developers, with the aid of community forums and chat discussions. **Accessibility** is assessed via the quality of how easy it is to obtain or gain access to a specific tool. Accessibility is also assessed based on the cost of using the tool. If the tool is cost effective or requires some kind of membership/payment, it is not taken into consideration for our experiment. **Documentation** is considered to be an important aspect in our case, since we analyze the security tools and without a proper documentation it is hard to do so. Additional criteria for the tool selection is **maintenance** and **easiness to install**. In our case, we did not have any case with a tool which was no longer maintained or troubles with installation since all security tools selected were web-based. However, we have an exception with one of the

tools, Oyente, in which the web-based tool failed to analyze the whole data-set, and thus, we used the CLI-version of the tool to complete our experiment.

4.4 Dataset Construction

4.4.1 Taxonomies

All three taxonomies proposed are build upon the related work research described in Section 3. The security vulnerabilities taxonomy is initially based on the taxonomy proposed by Atzei et al. [23], illustrated before in Figure 1. Afterwards, any other research paper or web article that was assessing security vulnerabilities/issues/bugs, was taken into account for the development of our taxonomy. In order to strengthen the validity of the work, we decided to solely include peer-reviewed papers and trusted web articles[3]. We narrowed our search down to approximately ten works [7, 13, 27, 5, 43, 44, 17, 24, 34, 41, 45]. Many articles referred to the same security vulnerabilities with different naming conventions. Therefore we analyzed each article and fetched the vulnerabilities they cover for further analysis. Based on the taxonomy categorization by Atzei et al. [23], we append each vulnerability in one of the categories; Blockchain, EVM, or Solidity. Additionally, based on the research and the security tools documentations, we categorize each vulnerability based on their severity level; high (critical/significant), medium, low (useful warnings).

The second proposed taxonomy is for the security code analysis tools. The primary idea to generate an overview taxonomy came from Hildenbrandt et al. [19]. In their paper, they provide a taxonomy of all software quality tools in Ethereum. Their taxonomy, categorizes Ethereum tools into: *Spec., Exec., Client, Verif., Debug, Bugs* and *Gas*[4]. They only found two security tools used to identify bugs, Oyente and F*. Also, in their taxonomy, they categorize Remix (web-based IDE) only as a debugger and a testing environment, and not as a tool for finding common issues. Therefore, in regards to the security tools used to identify bugs in a SC, this taxonomy is arguably vague and incomplete.

4.4.2 Audited Smart Contracts

In order to assess the false positive rate for each tool we needed secure/trusted and tested smart contracts which are considered to be bug-free or at least without any severe security vulnerability. Therefore, we decided to analyze smart contracts which are audited by a trusted company. From the different companies that do security audits, mentioned in 3.3.1, we chose Zeppelin for many reasons. First of all,

[3] Like the ones from Ethereum co-founders, or experienced blockchain developers.

[4] "**Spec.**: Formal specifications of the EVM language, **Exec.**: Executable on concrete tests, **Client**: Full Ethereum client, **Verif.**:Verify EVM program properties, **Bugs**: Finding common issues in SC., **Gas**: analyzing gas complexity." [19]

it is considered to be the most popular security audit firm, based on the audits of many popular smart contracts[5]. Secondly, based on our research, it provides a better documentation than all others. Thirdly and more importantly, once the security audit is completed, it is privately sent to the development team of the audited smart contract. It is then up to the development team to decide whether to publish the report provided on the Zeppelin blog[6]. On this blog, each smart contract security audit gives an overview of the smart contract, its potential problems or vulnerabilities (starting from the most severe ones), provides recommendations to increase security, and general reflections of the state-of-the-art security patterns [46]. Also, SmartDec started to provide blog articles with smart contracts audited from their side, but at the time of writing, they were in their infancy stage and had only three audits in total[7].

We collected 28 audited smart contracts in total from Zeppelin, starting from the one audited first, up until the last one (until October 23, 2017). However, we did a manual check for each security audit to dismiss a smart contract which had one of the following cases:

- Is written in a programming language other than Solidity
- Is identified with severe vulnerabilities from Zeppelin, and not updated afterwards
- Is used for token pre-sale[8]
- Very recent security audit (not updated, nor published)

After doing this data clean-up, we ended up with a total of 21 security audited smart contracts. Table 2 provides a list of the data-set for this category, where the ones highlighted in red are the discarded smart contracts which were not taken into consideration (7 SCs). Additionally, each one has a link to the corresponding source on Zeppelin's blog. Lastly, the source code for each smart contract is collected using EtherScan[9], which is a leading web-based Ethereum blockchain explorer.

4.4.3 Vulnerable Smart Contracts

Vulnerable smart contracts are used to identify the false negative rate within the tools. Additionally, they are used to identify the gaps, i.e. the vulnerabilities which are generally not covered by the tools. Due to a lack of one comprehensive list of Ethereum attacks, we had to rely on several research papers, web articles, and community discussions assessing attacks and bugs in smart contracts, to assemble a list of vulnerable smart contracts [17, 7, 19, 27, 47, 48]. Additionally, several

[5] More than $450 million have been raised by smart contracts that have been audited by Zeppelin.
[6] Link: https://blog.zeppelin.solutions/tagged/security
[7] Link: https://blog.smartdec.net/smart-contracts-security-audits/home
[8] They are mostly temporary smart contracts used to crowd-fund an organization.
[9] Link: https://etherscan.io/

Smart Contract	Source (* = https://blog.zeppelin.solutions)
Hacker Gold (HKG)	*/ethercamps-hacker-gold-hkg-public-code-audit-b7dd3a2fe43b
ArcadeCity (ARC)	*/arcade-city-arc-token-audit-9071fa55a4e8
Golem Network	*/golem-network-token-gnt-audit-edfa4a45bc32
ProjectKudos	*/ethercamps-projectkudos-public-code-audit-179ee0c6672d
EtherCamp's DSTC	*/ethercamps-decentralized-startup-team-public-code-audit-65f4ce8f838d
SuperDAO Promissory	*/draft-superdao-promissory-token-audit-2409e0fe776c
SuperDAO ConstitutionalDNA	*/draft-superdao-promissory-token-audit-2409e0fe776c
ROSCA	*/wetrust-rosca-contract-code-audit-928a536c5dd2
Matchpool GUP	*/matchpool-gup-token-audit-852a70330f2
iEx.ec RLC	*/iex-ec-rlc-token-audit-80abd763709b
Cosmos	*/cosmos-fundraiser-audit-7543a57335a4
Blockchain Capital (BCAP)	*/blockchain-capital-token-audit-68e882d14f0
WingsDAO	*/wingsdao-token-audit-f39f800a1bc1
Moeda	*/moeda-token-audit-ac72944caa6f
Basic Attention	*/basic-attention-token-bat-audit-88bf196df64b
Storj	*/storj-token-audit-32a9af082797
Metal	*/metal-token-audit-d7e4dbf17bcf
Decentraland MANA	*/decentraland-mana-token-audit-ee56a6bca708
Tierion Pre-sale	*/tierion-presale-audit-ec14b91c3140
Serpent Compiler	*/serpent-compiler-audit-3095d1257929
Hubbi	*/hubii-token-audit-227c0adf50ea
Tierion	*/tierion-network-token-audit-163850fd1787
Kin	*/kin-token-audit-121788c06fe
Render	*/render-token-audit-2a078ba6d759
Fuel	*/fuel-token-audit-30cc02f257f5
Enigma	*/enigma-token-audit-91111e0b7f8a
Global Messaging	*/global-messaging-token-audit-865e6a821cd8
Ripio	*/ripio-token-audit-abe43b887664

Table 2: Audited Smart Contracts Collection

sample smart contracts with specific vulnerabilities provided by Luu et al. in [49] are taken into consideration. Doing so, we tried to create a comprehensive list of smart contracts and incorporate as much dissimilar vulnerabilities as we can.

Table 3 provides a list of the data-set for this category.

Smart contract name	**Vulnerability**
TheDao	Re-entrancy
SimpleDao Sample compiler version 0.3.1	Re-entrancy, call to the unkown
SimpleDao Sample compiler version 0.4.2	Re-entrancy, call to the unkown
King of the Ether game (KoET)	Unchecked-send bug, Gasless send, Mishandled exception
KotET Sample compiler version 0.3.1	Gasless send
KotET Sample compiler version 0.4.2	Gasless send
GovernMental (PonziGovernmental)	Unchecked-send bug, Call-stack limit
GovernMental simplified sample 0.3.1	Immutable bugs, exception disorder, call-stack limit, unpredictable state
Rubixi	Immutable bugs, wrong constructor name
FirePonzi	Type casts (intentional scam)
Parity Multisig 1	Unintended function exposure
Parity Multisig 2 - Suicide Function called	Unintended function exposure
Parity Multisig 3- Suicide Function called	Unintended function exposure
GoodFellas	Typo (wrong constructor name)
StackyGame	Typo (wrong constructor name)
DynamicPyramid	Contract that does not refund
GreedPit	Contract that does not refund
NanoPyramid	Contract that does not refund
Tomeka	Contract that does not refund
Double3	Allows the contract owner to withdraw all the funds
TheGame	Allows the contract owner to withdraw all the funds
ProtectTheCastle	Call-stack limit, Withdraw option
RockPaperScissors (RPS)	Public moves
SmartBillions	Blockhash bug
EtherPot	Unchecked-send bug
TheRun	Timestamp dependence
OddsAndEvents Compiler 0.3.1 Sample	Keeping secrets
OddsAndEvents Compiler 0.4.2 Sample	Keeping secrets

Table 3: Vulnerable Smart Contracts Collection

For each smart contract, one or more associated vulnerabilities are presented. Smart contracts which are not real, are labelled with the term "Sample" on their name. Moreover, the ones highlighted in red are not taken into consideration, because i) *Suicide* function is called and their code is no longer available (two SCs cases) or ii) Smart contracts that allow their owners to withdraw the contract funds, are removed because it was considered to be more of a trust issue rather than a bug or vulnerability (two SCs cases). Therefore, out of 28 smart contracts in total, after clean-up we ended up with 24[10]. Lastly, their corresponding source code is also collected through EtherScan.

Generally speaking, this category of SCs consists of attacked SCs, buggy (incidents) SCs and samples. However, as for our experiment, it was only necessary that these smart contracts contain one or more bugs, we do not distinguish between them.

4.4.4 Sample Size Justification

Despite the fact that the data set could be considered as small, it is a solid representation for the current state of Ethereum and its security. Analyzing this kind of data size does not lead us to generalizable results or conclusions, however it provides us with valuable insight on current security tools and attacks/incidents.

To have a more complete data set for the vulnerable smart contracts category, we included several samples (examples) of buggy smart contracts as well. An additional precaution that we took regarding the sample size were the continuous checks on community discussions, and websites with blockchain news for any hack or incident that happened, up until 31st of October, 2017.

Smart Contracts before clean-up		
Vulnerable		**Audited**
28		28
Smart Contracts after clean-up		
	Vulnerable	**Audited**
Solidity	23	21
Bytecode	23	18

Table 4: Dataset Size

Considering Ethereum as a relatively new platform and the shortage on smart contracts adoption in a practical manner, the sample size presents the most of what we could extract from the network. Table 4 illustrates the overall data set for both categories, vulnerable and audited SCs, before and after clean-up.

[10] 23 SCs respectively, since two smart contracts have either bytecode or Solidity available, not both.

4.5 Implementation

We have already discussed how we generated the proposed taxonomy in the previous Section, 4.4, therefore in this section we will not cover them. However, we could just state that the new taxonomy of vulnerabilities that we proposed includes not only critical (high-severity) vulnerabilities, but also medium- and low-severity vulnerabilities such as useful warnings.

For the purpose of reproducibility, this section provides a detailed explanation of how we conducted the experiment, the evaluation method and data analysis process.

4.5.1 Tools Set Up and Examination

The tools which are chosen[11] for the experiment are: **Oyente, Securify, Remix** and **SmartCheck**. All four tools have only a web-based user interface, except Oyente which initially started as a CLI version and after a while implemented the web-version for a more user friendly approach. Now both of the versions are simultaneously maintained and updated. As a result, we had to set up only one tool, which is Oyente CLI and its installation is a relatively straightforward process considering that the tool is cross-platform through a docker container. Based on the Oyente documentation, there are several ways to set up the tool. For simplifying and resource purposes we decided to stick with the docker container version of Oyente CLI. In a Windows 10 machine we set up Oracle VirtualBox and installed Ubuntu 16. After some package installations, we were able to run the Oyente docker container with the following command: *docker pull luongnguyen/oyente && docker run -i -t -v "PATH" luongnguyen/oyente*. Doing so, we were able to analyze the smart contract applications that the Oyente web-version failed to analyze. Having in mind that Oyente CLI was the initial tool, it is considered to be more stable.

Before proceeding to the experiment we carefully inspect and examine the security tools and their usage. Appendix A.1, provides screen-shots of the user interface (UI) for each security tool analyzed. These figures show how these tools look like and how they interpret (visualize) the results. We see that Oyente uses an identical user interface design as Remix. However, the rest of the tools are using different approaches, overall on their UI design and specifically on visualizing the results from a particular analysis. Some tools provide a severity level attached to each vulnerability identified and highlight the line where a vulnerability is present, whereas some others additionally provide an explanation of the vulnerability and possible solutions.

Table 5 provides an overview of the tools used for the experiment, including their website, version used, and the date when the experiment took place. As we

[11] We have already justified the tools selection before in this section.

Security tool	Website	Date/Version
Oyente	oyente.melon.fund oyente.melonport.com oyente.melon.network	31/10/2017 oyente.melon.fund
SmartCheck	tool.smartdec.net	01/11/2017 Release0.0.1a Initial MVP-prototype
Remix	remix.ethereum.org ethereum.github.io	02/11/2017 remix.ethereum.org
Securify	securify.ch	04/11/2017 Beta version

Table 5: Security Tools for the Experiment

can see, two out of four tools have more than one website. The decision on which one to use is taken through a discussion with the tool developers and the community in chat channels. Lastly, we generated four tables for each tool examined (Oyente Table 6, Remix Table 7, Securify Table 8, SmartCheck Table 9), to state which vulnerabilities they cover. Only in Securify, Table 8, not all vulnerabilities stated are unlocked, therefore the ones that we did not have access to are highlighted in red. In order to have full access you have to make a request to the development team, which we did, but we never received an answer.

4.5.2 Experiment

Each tool experiment has been conducted separately. Before analyzing both categories, audited and vulnerable SCs, we generated the runtime binary code (bytecode) for each smart contract by compiling the Solidity source code with Solc compiler[12]. Out of 23 vulnerable SCs and 21 audited SCs, we were not able to compile only 3 audited smart contracts, because the Solidity compiler failed[13]. From four tools in total, only in two, Oyente and Securify, additional experiments for analyzing the bytecode have been conducted. Besides this, Solidity analysis have been conducted in all four tools.

The whole experiment is conducted manually (non-automatic). One of the main reasons why we conducted the whole experiment manually is that all four tools are in their infancy stage and as such they do not provide any APIs or other automatic approach. Also, despite the fact that running this experiment manually presents a labor intensive task, doing so, we were able to capture the tools' behaviour based on their responsiveness and effectiveness. As such, we are able to use those results later on, to propose possible future improvements within the tools.

[12]The method on how we set up 'SolC' compiler is explained in Appendix A.2
[13]Compiler error: *Import errors!*

Oyente	Vulnerabilities
	Callstack Depth Attack
	Timestamp Dependency
	Re-entrancy
	Transaction-ordering dependence
	Assertion failure

Table 6: Oyente Vulnerabilities

Remix	Vulnerabilities
	Unchecked-send bug
	tx.origin usage
	Re-entrancy
	Inline assembly
	Block timestamp
	Low level calls
	Blockhash usage
	Gas costly patterns
	this. on local calls
	Constant functions
	Similar variable names

Table 7: Remix Vulnerabilities

In total, 23 vulnerable and 21 audited smart contracts are analyzed with the four tools. Since each security tool identifies different vulnerabilities, not all vulnerable smart contracts were fit to test with all the tools. However, we decided to analyze all vulnerable smart contracts either way, in order to capture a general analysis on how many vulnerabilities each tool is not supposed (or not able) to identify. This also gives us an insight within the possible future improvements of the security tools. Furthermore, any failure (error) in analyzing a specific smart contract is noted.

4.5.3 Data Analysis and Evaluation Methodology

The data analysis consists of four different assessments:

- Effectiveness
- Accuracy
- Consistency
- Overall assessment based on the manual analysis

Apart from the overall assessment, all other three assessments have a clear data analysis process and an evaluation method. The overall assessment is done throughout the experiments, without any particular focus or plan. During this assessment,

Securify	Vulnerabilities
	Transactions may affect Ether Receiver
	Transactions May Affects Ether Amount
	Gas-dependent Reentrancy
	Reentrancy with constant gas
	Reentrancy method call
	Unchecked Transaction Data Length
	Unhandled Exception
	Use of Origin instruction
	Missing Input Validation
	Locked Ether
	Use of untrusted Inputs

Table 8: Securify Vulnerabilities

SmartCheck	Vulnerabilities
	Strict balance equity
	Byte array
	Transfer forwards all gas
	DoS by external contract
	Token API violation
	Costly loop
	Integer division
	Locked money
	Malicous libraries
	Compiler version not fixed
	Private modifer
	Redundant fallback function
	Re-entrancy
	send instead of transfer
	Style guide violation
	Time-stamp dependence
	tx.origin usage
	Unchecked external call
	Unchecked math
	Unsafe type inference
	Implicit visibility level

Table 9: SmartCheck Vulnerabilities

we have the user in mind – one without any deep knowledge and experience in security.

We asses the usability of the tools in terms of easiness to use, and any potential issue a user could face.

Effectiveness

Generally, the effectiveness of the tools is assessed based on the percentage of the smart contracts in total that the tools were able to analyze. The nature of the data-set consists of different type of smart contracts, including here; secure, vulnerable, old compiler versions, grand scale, small scale, and samples. Additionally, the symbolic execution methodology predominantly used to identify vulnerabilities is rather complex, since it analyzes the code without any known input and also loops through the blockchain to cover all possible behaviours. Therefore, the security tools and the methodology are sometimes prone to errors and failures.

Accuracy

Assessing the effectiveness of the tools does not necessarily show us how accurate the results are. Therefore, it is crucial to assess the accuracy of the results that the tools produce. Accuracy is assessed through the false positive and false negative rates. Initially, this assessment idea came from Zhang et al. [50], in which they evaluate the anti-phishing tools with the same methodology, using 200 verified phishing URLs (in our case vulnerable SCs) and 516 legitimate URLs (in our case audited SCs), to test the performance of 10 popular phishing tools (in our case 4 popular Ethereum security code analysis tools).

The initial idea of the data evaluation in regards to the accuracy was to assume that the audited smart contracts were bug-free, and each vulnerability discovered on them is automatically considered as a false positive. However, considering that a security audit does not guarantee full security/safety, and due to the results we obtained, the need for a manual analysis was inevitable.

Firstly, we ran 21 audited smart contracts in each tool. Based on the results obtained and the severity level of vulnerabilities, we decided to manually analyze only five vulnerabilities[14]. Other vulnerabilities are not considered for manual analysis, either because they cannot be manually analyzed (e.g. gas costly patterns), the security audit firm does not cover them, or they are vulnerabilities with low severity (e.g. useful warnings or style violations). The manual analysis is conducted as follows:

- Check the Zeppelin source of the smart contract in which a vulnerability is identified
- If the vulnerability is also identified by Zeppelin, and the smart contract

[14]Including here: Reentrancy, timestamp dependence, transaction re-ordering, unchecked-send bug, tx.origin usage.

owners have not modified the SC for that specific vulnerability[15] or they have reasoned it – it is removed from the false positives results.
- Additionally, a manual analysis following a list of recommendations for smart contract security [41] is conducted and the line where the vulnerability is identified is checked manually to verify if it is false positive.

The other approach in regards to accuracy is the false negative assessment. This is done through the vulnerable SCs, which we already know that have at least one particular vulnerability. If the tools state that they are able to identify a specific vulnerability, and they fail to do so, it is considered a false negative. This also gives us results of how many vulnerable SCs proceed with undetected security issues because the tools are not supposed to identify them.

To conclude, the results obtained from this experiment have two possibilities of failure:

- **False Positive** when the tool identifies a vulnerability in an audited smart contract, and the manual inspection does not identify it.
- **False Negative** when the security tool does not find a specific vulnerability in a vulnerable SC.

Consistency

Consistency assessment relates with the security tools which are able to analyze both bytecode and Solidity. We check for any inconsistency where a tool gives different results for the bytecode analysis and different ones for the Solidity analysis of the same smart contract. Additionally, we check any other inconsistency which may occur during the experiment. For example, failure to analyze a SC with the first try, failure to distinguish a runtime binary (bytecode) from creation code[16], and any other reliability issue.

[15] As we mentioned before, the SCs which in general are not modified/updated after the audit, are not taken into consideration.

[16] A contract creation code contains the EVM code of the account initialization procedure [14]. It does not contain any logic or smart contract source code except for the constructor.

5 Proposed Taxonomies

This chapter illustrates all three taxonomies that we have mentioned so far. Considering that we covered the taxonomies justification and the data-set construction in the previous chapter, here we are only presenting them, and briefly explaining/discussing them.

5.1 Taxonomy of Vulnerabilities/Bugs

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

Table 10: Taxonomy of Vulnerabilities

By this stage, we can no longer compare this taxonomy (Table 10) to the one provided by Atzei et al. [23] (illustrated in Table 1). This is due to many reasons. First and most importantly, this taxonomy does not only include high-severe vulnerabilities, such as in [23], but also low (non-critical) and useful warnings. Secondly, instead of having an "Attacks" section as in [23], we have a severity level (from 1 to 3) for each vulnerability, which is acquired through the research done and also

based on the attacks that have occurred over time in Ethereum. The only resemblance left is the vulnerability categorization level (blockchain, EVM, and Solidity). However, it can be seen as an extension of the one provided by Atzei et al. [23]. As shown on the table, the least modified category is the EVM level. This comes as a result of Ethereum developers continuously working on upgrading the EVM. For example, the *stack size limit* vulnerability is not present anymore[1], and through our research we could not find any other vulnerability that is EVM specific.

We have previously mentioned that many articles and authors refer to the same vulnerability with different names. Therefore, to avoid confusion, for some vulnerabilities we provide more than one name/explanation. Also, some vulnerabilities have several severity levels, and it is challenging to assign only one severity level, because some vulnerabilities are represented as useful warnings, but once they are utilized in a critical SC component, they could present major security issues.

5.2 Taxonomy of Tools

Security Tool	Method	Bytecode analysis	Solidity analysis	CLI[2]	WUI[3]
Oyente	Symbolic execution	✓	✓	✓	✓
Remix	Formal verification	X	✓	✓	✓
F* Framework	Formal verification	✓	✓	✓	X
Gasper	Symbolic execution	✓	X	N/A	N/A
Securify	Formal verification	✓	✓	X	✓
Simple Analysis[4]	Heuristics	✓	X	✓	X
SmartCheck	N/A	X	✓	X	✓
Imandra Contracts	Formal verification	**N/A - paid access**			
Mythril	Concolic testing (symbolic execution)	✓	✓	✓	X

Table 11: Taxonomy of Tools

Table 11 provides an overview of the generated taxonomy for security code analysis tools. The categorization is based on their similarities, such as, the methodology they use (highlight) to identify security issues (symbolic execution, formal verification), which code analysis they are able to perform (bytecode, Solidity), and their

[1]Modifications to the gas rules have eliminated this issue. Link: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md

[2]CLI: Command Line Interface

[3]WUI: Web-based User Interface

[4]A simple program analysis tool specifically used for detecting *unchecked-send bug*. Link: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/

user interface (CLI, WUI). As it can be seen, for some tools we have partial information, either because the tool is not released yet (Gasper), the methodology is not stated in their documentation (SmartCheck), or the tool requires paid access for additional information and usage (Imandra Contracts).

Compared to the taxonomy provided by Hildenbrandt et al. [19], which covers all Ethereum software quality tools, our taxonomy is only focused on security tools used to identify vulnerabilities/bugs in smart contracts. The security tools stated here use symbolic execution and formal verification as a methodology to identify vulnerabilities. These two methodologies, generally, are used interchangeably and in combination. However, we have stated the methodology which the tools highlight most in their documentation or papers. For example, Mythril, refers to the symbolic execution methodology in a more specific manner, as concolic testing.

5.3 Tools/Vulnerabilities Matrix

Security Tool	ReEntrancy	Timestamp dependency	TOD[5]	Mishandled exceptions	Immutable Bugs	tx.orgin usage	Gas costly patterns	Blockhash usage
Oyente	✓	✓	✓	✓	✓	X	X	X
Remix	✓	✓	X	✓	X	✓	✓	✓
F*	✓	X	X	✓	X	X	X	X
Gasper	X	X	X	X	X	X	✓	X
Securify	✓	X	✓	✓	X	✓	X	X
S. Analysis	X	X	X	✓	X	X	X	X
SmartCheck	✓	✓	✓	✓	✓	✓	✓	X
Imandra	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Mythril	✓	X	X	✓	✓	✓	X	X

Table 12: Tools/Vulnerabilities Matrix

Table 12 provides the generated matrix of the security tools and the most severe vulnerabilities they cover. The total list of vulnerabilities is extensive, where for example, only SmartCheck identifies 21 vulnerabilities in total, including here warnings and other low-risk vulnerabilities. Therefore, based on their importance, not all vulnerabilities are taken into consideration for this matrix. Most security tools (6 out of 8) identify more than one vulnerability, and only two tools identify one vulnerability each, Gasper (gas costly patterns), and Simple analysis (unchecked-send bug). Furthermore, since Imandra requires paid access, we do not have any information on what kind of vulnerabilities it covers.

To simplify the matrix, in the *mishandled exceptions* we cover: *exception disorders*, *unchecked-send bug*, and *gasless send*. Whereas, in the *immutable bugs* category we cover, *type casts* and *integer over- and underflow* as well.

Additionally, *visibility* (function exposure) checks are omitted because they are covered only from Smart-Check. And since the *stack-size limit* is not a vulnerability

[5]TOD: Transaction-ordering dependence

anymore, it is eliminated from the list, even though Oyente still has that vulnerability check.

This matrix (Table 12), and the other two taxonomies presented in this chapter (Table 10, Table 11) are further discussed based on the results obtained from the experiment. Additionally, the answer of the first research question as well as the proposed future improvements of the security tools, heavily rely on these taxonomies.

6 Experiment and Results

This chapter summarizes the experiments conducted and illustrates the results obtained. Firstly, the results for each security tool are separately presented, and afterwards, in Section 6.5, a general overview of the results and comparisons for all four tools is presented, visualized, and briefly explained.

6.1 Oyente

As we already mentioned before, the web version of Oyente is chosen primarily for this experiment. However, during the experiment, we ran into some issues with the web version and therefore we had to utilize the use of Oyente CLI version, to complete our experiment. The general results for Oyente's effectiveness, accuracy and consistency, presented in Section 6.5, are based on the merged results from both experiments, Oyente WUI and Oyente CLI.

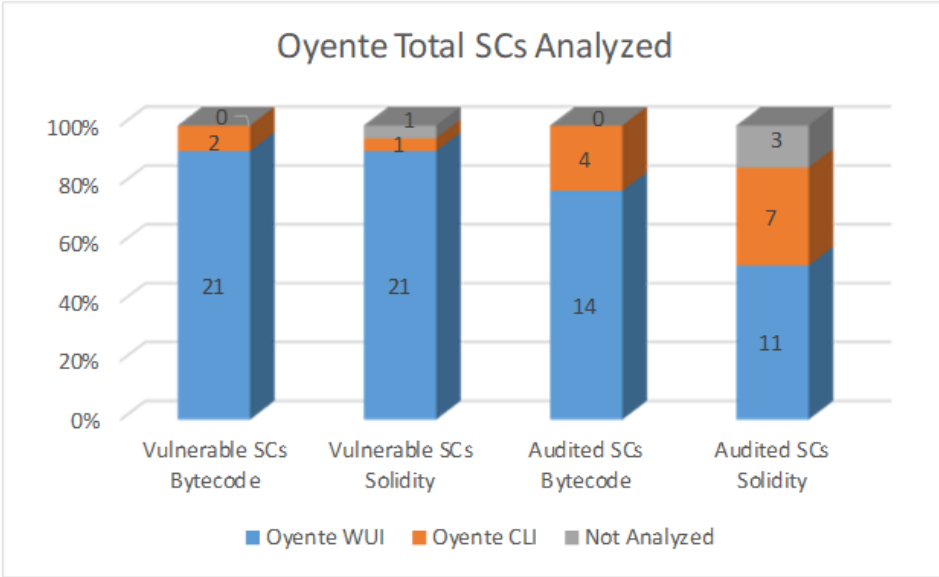

Figure 4: Oyente (WUI and CLI) Total SCs Analyzed

Figure 4 illustrates the total number of smart contracts analyzed from both WUI and CLI versions. Additionally, highlighted in grey, are the smart contracts which

are not analyzed from any of the versions. As we can see, Oyente WUI failed to analyze two bytecode-vulnerable SCs , two vulnerable-Solidity SCs, four bytecode-audited SCs, and ten Solidity-audited SCs. Out of these, Oyente CLI was able to analyze 14 SCs and failed to analyze in total four Solidity (1 vulnerable and 3 audited) SCs. Overall, Oyente achieved to analyze 100% of vulnerable and audited bytecode SCs. On the other hand, it performed worse on analyzing the Solidity-audited SCs with 85.7% (or 18 SCs), followed by Solidity-vulnerable SCs with 95.6% (or 22 SCs).

Oyente results consist of four vulnerability checks. Additionally, it provides an EVM code percentage for each SC analyzed. Analyzing the SC's bytecode gives us four results (vulnerabilities): **callstack depth, timestamp dependence, reentrancy**, and **transaction-ordering dependence**. The Solidity analysis gives us one more additional check, which is **assertion failure**. The results generated from Oyente are either true or false, where *true* means the vulnerability is present in a SC, and *false*, the vulnerability is not present in a SC.

Oyente WUI		**EVM coverage**	**Callstack depth**	**Timestamp dependency**	**Reentrancy**	**ToD**
Vulnerable	**Bytecode**	52.60%	20-T | 1-F	2-T | 19-F	4-T | 17-F	9-T | 12-F
	Solidity	53.99%	20-T | 1-F	2-T| 19-F	4-T | 17-F	9-T | 12-F
Audited	**Bytecode**	61.21%	0-T | 14-F	0-T | 14-F	0-T | 14-F	1-T | 13-F
	Solidity	59.96%	0-T | 11-F	1-T | 10-F	0-T | 11-F	1-T | 10-F

Table 13: Oyente - WUI Results

Oyente CLI		**EVM coverage**	**Callstack depth**	**Timestamp dependency**	**Reentrancy**	**ToD**
Vulnerable	**Bytecode**	53.80%	2-T | 0-F	1-T | 1-F	0-T | 2-F	1-T | 1-F
	Solidity	64.00%	1-T | 0-F	1-T | 0-F	0-T | 1-F	1-T | 0-F
Audited	**Bytecode**	18.98%	0-T | 4-F	0-T | 4-F	0-T | 4-F	0-T | 4-F
	Solidity	79.67%	0-T | 7-F	0-T | 7-F	0-T | 7-F	0-T | 7-F

Table 14: Oyente - CLI Results

Tables 13 and 14 present an overview of the raw data obtained from the experiment (Oyente WUI and CLI versions). Four categories presented (callstack depth, timestamp dependency, reentrancy and ToD) are the vulnerabilities that Oyente identifies, where true is marked with **T** and false is marked with **F**. Table 15 presents an overview of the merged results from both WUI and CLI versions of Oyente, which are used for the general tool comparisons later on. These results are merged and considered indistinguishable from now on.

Oyente Web and CLI		EVM coverage	Callstack depth	Timestamp dependency	Reentrancy	ToD[1]
Vulnerable	**Bytecode**	52.77%	22-T | 1-F	3-T | 20-F	4-T | 19-F	10-T | 13-F
	Solidity	54.45%	21-T| 1-F	3-T | 19-F	4-T | 18-F	10-T | 12-F
Audited	**Bytecode**	51.82%	0-T | 18-F	0-T | 18-F	0-T | 18-F	1-T | 17-F
	Solidity	65.74%	0-T | 18-F	1-T | 17-F	0-T | 18-F	1-T | 17-F

Table 15: Oyente WUI&CLI Merged Results

Table 16 shows for each vulnerability separately, the total number and percentage of vulnerable SCs identified from Oyente. As we can see, almost all vulnerable SCs (95%) are identified with the *callstack depth* vulnerability. Moreover, four SCs are identified with the reentrancy vulnerability, three SCs with the *timestamp dependency*, and ten SCs with the *transaction-ordering dependence*. Whereas, from the whole data-set of audited SCs, Oyente identified only two vulnerable SCs, one with the *timestamp dependency* and the other one with the *transaction-ordering dependence*.

Oyente Web and CLI		Callstack depth	Timestamp dependency	Reentrancy	ToD
Vulnerable SCs	**Bytecode**	22 (95.65%)	3 (13.04%)	4 (17.39%)	10 (43.47%)
	Solidity	21 (95.45%)	3 (13.63%)	4 (18.18%)	10 (43.47%)
Audited SCs	**Bytecode**	0 (0%)	0 (0%)	0 (0%)	1 (5.55%)
	Solidity	0 (0%)	1 (5.55%)	0 (0%)	1 (5.55%)

Table 16: Oyente - Total number (and percentage) of vulnerable SCs

6.2 Remix

Remix is a web-based IDE for developing smart contracts. In addition, it serves as a security code analysis tool in which it checks for possible vulnerabilities. It could be used to analyze only the Solidity source code.

Remix was able to analyze 100% of the vulnerable SCs, and 90% of the audited SCs[2]. Unlike Oyente, Remix does not only show whether the vulnerability is present or not, but it also highlights the line where the vulnerability is present, and it counts how many times each vulnerability is present in a SC. Table 17 illustrates the total number of vulnerabilities identified from Remix, in both, vulnerable and audited SCs. *Gas costly patterns* present the highest number of vulnerabilities in both of the categories, followed by *similar variable names* in audited SCs, and *unchecked-send bug* in vulnerable SCs.

In addition to the vulnerabilities presented in the following tables (17 and 18),

[1]ToD: Transaction-ordering dependence
[2]Remix failed to analyze two audited SCs with *'compiler errors'*.

Remix checks for three other vulnerabilities which are not presented here because Remix did not identify any SCs with those vulnerabilities, *tx.origin usage*, *low level calls* and *this. on local calls*.

Remix	Unchecked send bug	ReEntrancy	Block timestamp	Blockhash usage	Gas costly patterns	Constant functions	Similar variable names	Inline assembly
Vulnerable SCs	83	42	18	13	282	22	44	1
Audited SCs	8	13	34	0	223	72	113	1

Table 17: Remix - Total number of vulnerabilities

Remix	Unchecked send bug	ReEntrancy	Block timestamp	Blockhash usage	Gas costly patterns	Constant functions	Similar variable names	Inline assembly
Vulnerable SCs	20 (86.9%)	23 (100%)	8 (34.7%)	2 (8.69%)	23 (100%)	10 (43.4%)	6 (26%)	1 (4.3%)
Audited SCs	6 (31.57%)	6 (31.57%)	7 (30.43%)	0 (0%)	19 (100%)	16 (69.5%)	16 (69.5%)	1 (5.2%)

Table 18: Remix - Total number (and percentage) of vulnerable SCs

Table 18 presents the total number, and their corresponding percentage, of all vulnerable SCs for each vulnerability separately. As expected from the Table 17, Remix identified all (vulnerable and audited) SCs with one or more gas costly pattern. It also identified 100% of vulnerable SCs with (at least one) re-entrancy bug. All audited SCs are identified as vulnerable with at least one bug and the results show that only the blockhash bug was not found in any audited SC.

6.3 Securify

Securify is able to run smart contract analysis in three different ways: through bytecode, Solidity and smart contract address. The latter did not work properly[3] and therefore was not taken into consideration for our experiment. We conducted our experiment for bytecode and Solidity source code only.

Figure 5 illustrates the total percentage of SCs analyzed from Securify. It performed worse (with 65.2%) on analyzing the bytecode of vulnerable SCs while it performed best (with 88,9%) on analyzing the bytecode of audited SCs. Only one SC analysis failed because of 'compiler error', and the rest of the failures are related with 'analysis timeout' and 'failed loading' errors.

[3]Before running the experiments, we examined/inspected the tools, and analyzing a smart contract with its Ethereum address did not seem to work.

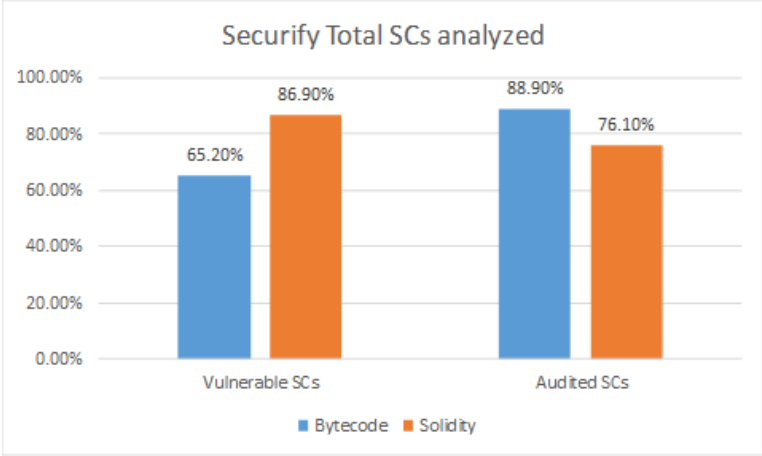

Figure 5: Securify - Total SCs Analyzed

Table 19 presents an overview of the results obtained from the Securify experiment and Figure 6 presents a visualized version of the same results. As mentioned in the methodology chapter, Securify has many vulnerabilities which are locked, and we did not have access to those. The table only presents the vulnerabilities which we were able to test, whereas the *tx.origin usage*, is not presented since none of the SCs were identified with that bug. **AT** presents the abbreviation for 'analysis timeout'. This is different from 'failed loading', since when the latter happens none of the vulnerabilities are checked (it is considered a failure to analyze the whole SC). With AT, Securify is able to analyze the SC as a whole but fails to analyze a particular vulnerability. Securify identified 100% of vulnerable SCs with the *transaction reordering* bug.

Securify Results		**Transaction Reordering**		**Insecure Coding Patterns**	**Use of untrusted Input in Security operations**
		Ether receiver affected	**Ether amount affected**	**Unhandled Exceptions**	**Use of untrusted input**
Vulnerable SCs	**Bytecode**	15 (100%)	15 (100%)	13 (86.6%) (13.4% AT)	0 (0%)
	Solidity	20 (100%)	20 (100%)	15 (75%) (25% AT)	2 (10%)
Audited SCs	**Bytecode**	4 (25%)	4 (25%)	0 (0%)	0 (0%)
	Solidity	5 (31.25%)	5 (31.25%)	0 (0%) (6.25% AT)	1 (6.25%)

Table 19: Securify - Total number (and percentage) of vulnerable SCs

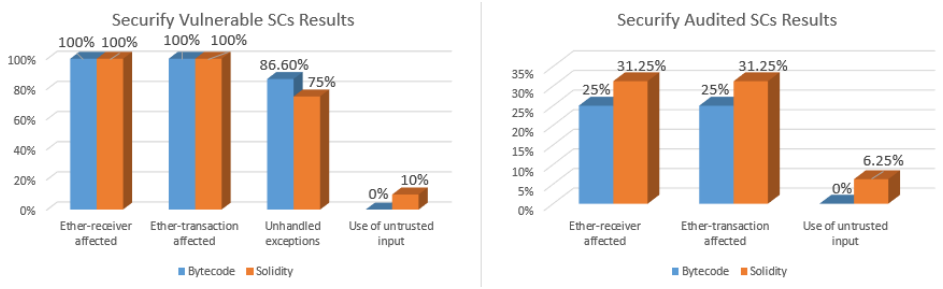

Figure 6: Securify - Results

6.4 SmartCheck

SmartCheck too runs analysis in Solidity source code only. It successfully analyzed all 100% of smart contracts from both categories; vulnerable and audited. Unlike Oyente and Securify, same as in Remix, SmartCheck identifies whether there is more than one issue for the same vulnerability in a SC. Therefore, Table 20 shows an overview of the total number of vulnerabilities detected by SmartCheck. The most present issue is **send instead of transfer** within the vulnerable SCs (85), followed by **re-entrancy** vulnerability (67) – again, in the whole data-set of vulnerable SCs. The *send instead of transfer* issue can not be considered as a vulnerability, because it only represents a warning and a recommendation that is better to use *transfer* than *send*. Since, SmartCheck checks 21 vulnerabilities in total, the ones that have not been identified by any SC are removed from the tables and charts. Those are: *strict balance equity, byte array, transfer forwards all gas, redundant fallback function, tx.origin usage,* and *unchecked math*.

SmartCheck Results	DoS	Token API	Costly loop	Integer division	Locked money	Malicous libraries	Compiler version
Vulnerable SCs	15	0	17	34	1	1	5
Audited SCs	49	14	13	6	0	6	20
	Private modifer	**Re-entrancy**	**send transfer**	**Style guide**	**Time-stamp**	**Unchecked call**	**Unsafe inference**
Vulnerable SCs	57	67	85	17	18	0	7
Audited SCs	4	15	10	2	38	1	2

Table 20: SmartCheck - Total number of vulnerabilities

Table 21 presents the total number and percentages of vulnerable SCs as identified by SmartCheck, for each vulnerability separately. As expected from Table 20,

almost all vulnerable SCs (91.3%) are using send instead of transfer. Whereas, 86% of the vulnerable SCs are identified with the re-entrancy bug.

SmartCheck Results	DoS	Token API	Costly loop	Integer division	Locked money	Malicous libraries	Compiler version
Vulnerable SCs	13 (56.52%)	0 (0%)	13 (56.52%)	11 (47.82%)	1 (4.34%)	1 (4.34%)	5 (21.73%)
Audited SCs	14 (66.6%)	7 (33.33%)	6 (28.57%)	4 (19.04%)	0 (0%)	6 (28.57%)	18 (85.71%)
	Private modifer	Re-entrancy	send transfer	Style guide	Time-stamp	Unchecked call	Unsafe inference
Vulnerable SCs	7 (30.43%)	20 (86.95%)	21 (91.3%)	5 (21.73%)	8 (34.78%)	0 (0%)	2 (8.69%)
Audited SCs	1 (4.76%)	6 (28.57%)	7 (33.33%)	1 (4.76%)	8 (38.09%)	1 (4.76%)	1 (4.76%)

Table 21: SmartCheck - Total number (and percentage) of vulnerable SCs

Figure 7 visualizes the results from Table 21 – for the purpose of having a better representation of the overall SmartCheck results.

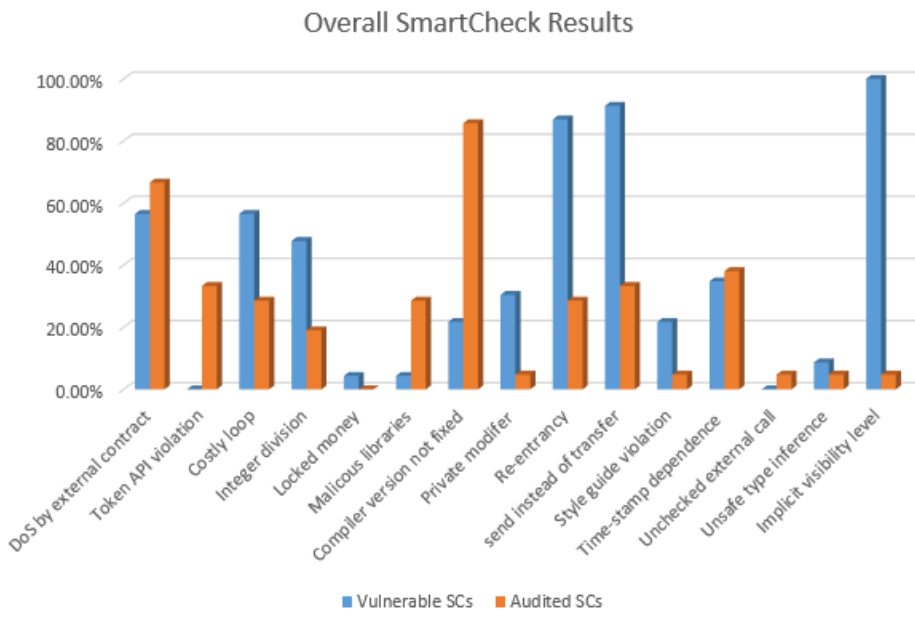

Figure 7: SmartCheck - Total Vulnerable SCs visualized

6.5 Overall Results

This section provides an overall interpretation and comparison of the results for all four tools. It consists of four subsections: effectiveness, accuracy, consistency, and overall assessment.

6.5.1 Effectiveness

Figure 8 and 9 visualize the comparison between the effectiveness of the tools and Table 22 presents the merged results from both figures, additionally with the number of SCs analyzed. The results (percentages) obtained are based on how many SCs, from the whole data-set, each tool was able to analyze. As we can see, only SmartCheck achieved 100% effectiveness on both categories. Oyente achieved 100% on both categories, but only in bytecode analysis. On the other hand, Remix achieved 100% only in vulnerable SCs analysis. Overall, Securify performed worse with 65% effectiveness in bytecode-vulnerable SCs analysis, and 86% in Solidity-vulnerable SCs analysis.

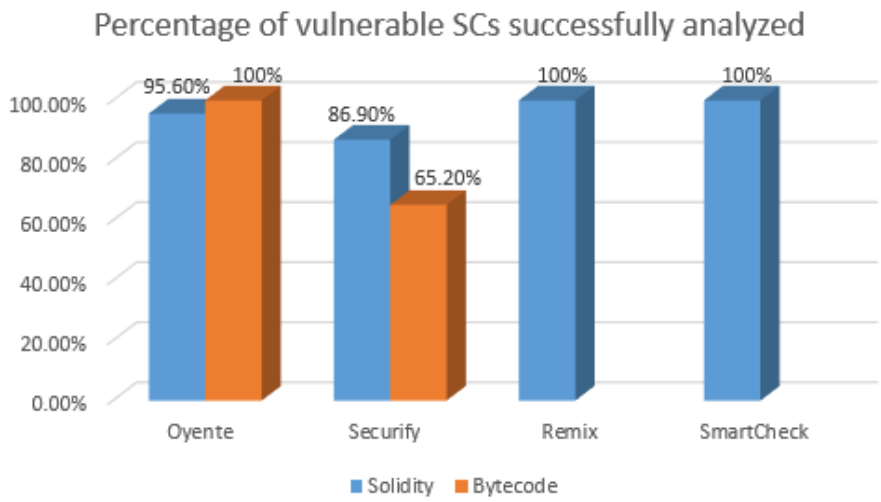

Figure 8: Tool Effectiveness in Vulnerable SCs

Total SCs Analyzed	SmartCheck		Oyente		Remix		Securify	
	Vulnerable	Audited	Vulnerable	Audited	Vulnerable	Audited	Vulnerable	Vulnerable
Bytecode	/	/	23 (100%)	18 (100%)	/	/	15 (65.2%)	16 (88.88%)
Solidity	23 (100%)	21 (100%)	22 (95.6%)	18 (85.7%)	23 (100%)	19 (90.4%)	20 (86.9%)	16 (76.1%)

Table 22: Tools Effectiveness (Vulnerable & Audited SCs)

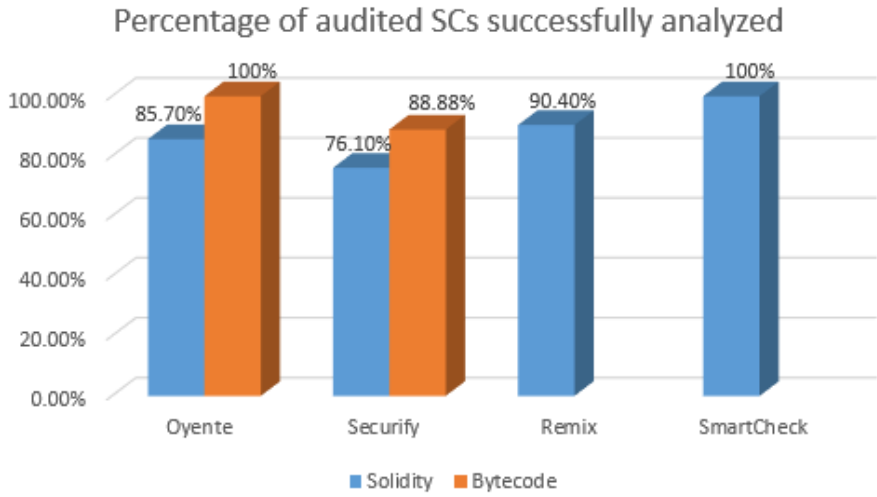

Figure 9: Tool Effectiveness in Audited SCs

6.5.2 Accuracy

The accuracy of the tools is obtained through an in-depth inspection of audited SCs result analysis. To assess the false positive rate for each tool, we chose only **five** severe vulnerabilities which could be manually analyzed.

Since in the beginning each vulnerability detected in an audited SC is considered to be a false positive, Figure 10 presents the initial results for false positives – before the manual check. As discussed in the methodology chapter, considering the fact that audited SC are not necessarily secure, we ran manual checks to ensure the detection of false positives. Figure 11 presents the results after the manual check. It can be seen in both of the figures that only four vulnerabilities are presented, and this is because the *tx.origin usage* vulnerability is removed since none of the tools identified a SC with this bug. The percentages show how many SCs were identified as vulnerable with a specific bug. For example, SmartCheck identified 38.09% of audited SCs with the *timestamp dependence* and after the manual check, it turned out that only 9.52% SCs in total (or 25% of the 38.09%) were false positives.

The most substantial difference before and after the manual analysis is within the timestamp-dependence vulnerability. The other three results from *reentrancy*, *transaction re-ordering*, and *unchecked-send bug* remained unchanged. That means that, e.g. out of 28.57% SCs identified with the reentrancy vulnerability from SmartCheck, after the manual analysis, 100% of the 28.57% SCs, are false positives.

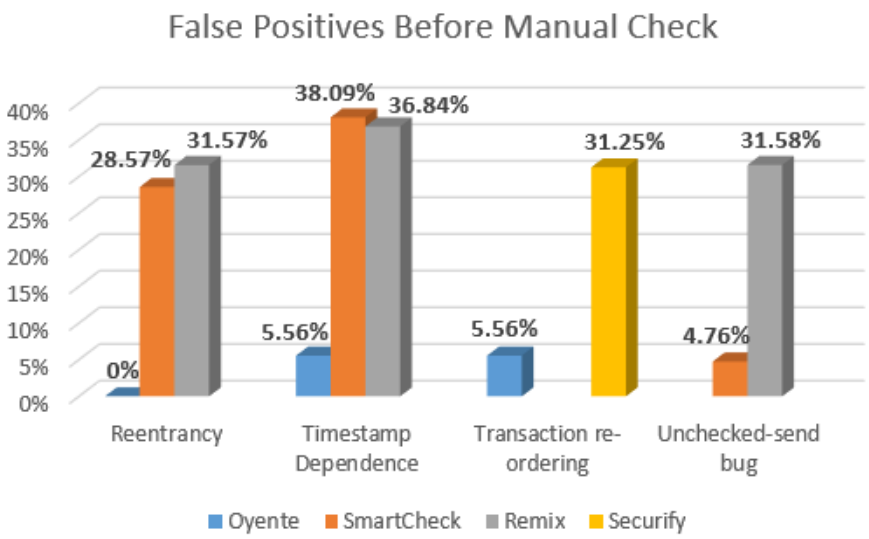

Figure 10: False Positives Before Manual Check

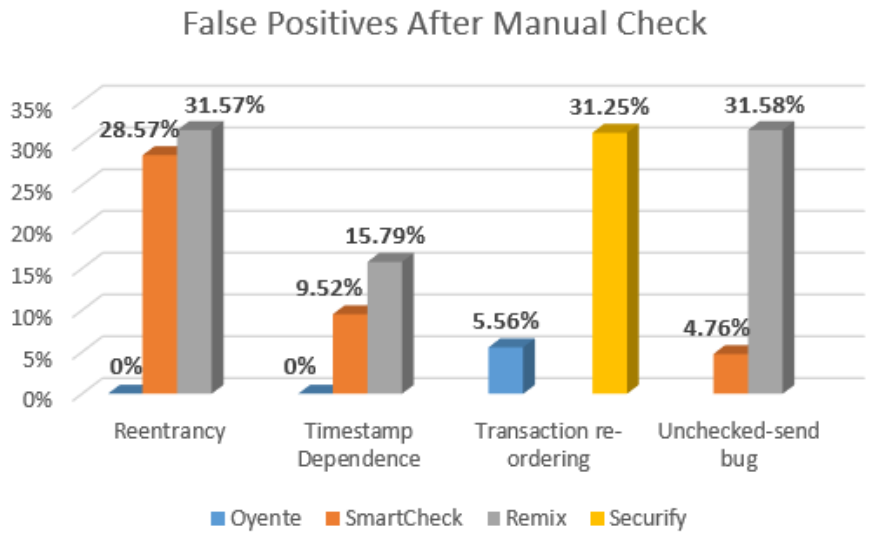

Figure 11: False Positives After Manual Check

The next assessment done in regards to the accuracy of the tools, is the false

negative rate (Figure 12). The percentages are obtained with a thorough inspection of the vulnerable SCs analysis results. In our case, considering the small data-set size, we only found one case of false negative (which represents 4.35%). The smart contract **TheRun**, suffers from the **timestamp dependence** vulnerability, which is used to generate random numbers and reward a jackpot based on that. Even though Oyente states that it identifies potential timestamp-dependence vulnerabilities, in this case, it failed to do so.

Furthermore, this thorough analysis on discovering the false negative is also used to check the vulnerabilities which the tools are not supposed to cover. As we can see from Figure 12, that percentage is significantly high for all four tools. However, that percentage is vastly influenced from our data-set, which is furthermore discussed in Chapter 7.

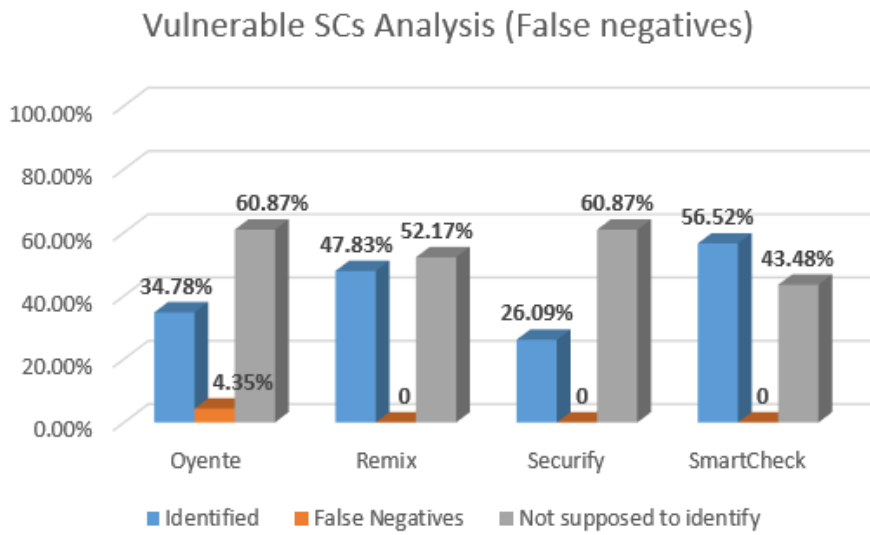

Figure 12: False Negatives

The raw data results obtained from the analysis of the false positive and false negative rates are presented in Appendix A.3.

6.5.3 Consistency

Consistency is assessed only in two tools, the ones that analyze both bytecode and Solidity analysis, which are Oyente and Securify. It is assessed throughout both experiments to find any inconsistency within the tools, e.g., if the tool produces some results for a SC with the bytecode analysis, and different results for the same

SC with the Solidity analysis.

The inconsistency of the tools could potentially be explored through the Oyente general results (Table 15), and Securify results (Table 19). However, taking into consideration that the data-size is different for each analysis, it is hard to do so.

In Oyente, we encountered **one** case of inconsistency. The **HKG** smart contract with the Solidity analysis identified the *timestamp dependence*, whereas with the bytecode it did not. This can also be considered as a false negative case with Oyente (which makes it two). In Securify, we encountered three cases of inconsistency in one SC analysis. Securify, in **Kin** smart contract Solidity analysis identified **three** vulnerabilities (transactions affect Ether receiver, transactions affect Ether amount, and block timestamp), whereas with the same contract in bytecode analysis it did not identify any of the aforementioned vulnerabilities. These corresponding results are already considered as false positives in the previous section.

6.5.4 Overall Assessment

The observations that we made throughout our experiments are:

- Oyente's website was down several days during the period of time when we inspected the tools. The developers of Oyente were not aware of the issue, and before we started the experiment, they were informed and fixed it.
- Oyente failed to analyze 13 (out of 81) analysis in the first try.
- Securify falsely identified two SC bytecodes as contract creation codes.

Additional Information

The complete data-set (vulnerable and audited smart contracts) and the results for each tool experiment separately can be found in the GitHub repository in https://github.com/DikaArdit/Master-Thesis.

7 Discussion

Ethereum smart contract security represents a challenging issue, and is tackled by many studies. Some of them focus on vulnerability taxonomies, or best practices that should be followed, whereas others focus on security code analysis tools and other automated means to run security checks. Therefore, to answer the research questions of this thesis, we performed a comprehensive investigation of security vulnerabilities in smart contracts, and provided several taxonomies. Additionally, we thoroughly investigated several popular security code analysis tools.

The general tool analysis and comparisons based on the effectiveness, accuracy and consistency represent the main results of the experiment conducted. However, the examination of each tool separately, is of a great importance as well. Firstly, the manual analysis of the tools gave us an overview of their behaviour and other observations which we would not be able to obtain through an automated experiment. Secondly, inspecting each tool separately aimed to identify specific pitfalls within each tool.

The first research question and its sub-questions are answered through our research work and mostly rely on the proposed taxonomies. The second research question and its sub-questions are answered through the results obtained from our experiment. Lastly, a limitations section is presented with the main pitfalls we experienced and the issues which may have jeopardized the overall validity of the results.

First Research Question: **What kind of security vulnerabilities arise in smart contract programming?**

Through our research work we identified several types of smart contract vulnerabilities. Mainly, the vulnerabilities are presented jointly with their severity level, i.e., the damage they could cause and the overall risk they carry. Therefore, based on the severity level, the vulnerabilities could be identified as:

- **Low** - Useful warnings and minor issues.
- **Medium** - Vulnerabilities that lead to incidents and do not constitute an opportunity for an adversary to hack the smart contract. Medium severity vulnerabilities are considered to be problematic because they might affect the effective continuity of a SC, where – in case of exploiting that vulnerability, nobody benefits from it. For example, the GovernMental smart contract incident, in which 1100 Ether got stuck because of a gas costly pattern.

- **High** - Major and critical vulnerabilities present risky issues in the sense that they endanger the wholeness of smart contracts and make them prone to hacks and malicious modifications from adversaries/hackers.

Categorizing vulnerabilities only through their severity level does not tell us much about the whole nature of the vulnerability. To answer the **first sub-question** on **how can we categorize these vulnerabilities**, we proposed the taxonomy of vulnerabilities in Section 10. Based on our research and categorization we argue that the categories (levels) used, are the most valid representation for all types of security vulnerabilities in Ethereum smart contracts. Our taxonomy, same as the one introduced by [23], categorizes the vulnerabilities in:

- Blockchain
- EVM
- Solidity (applicable for other high-level programming languages)

We append a severity level from 1-3, i.e., low, medium, high, to each security vulnerability. On the other hand, in [23], they append an attack example for each vulnerability. Furthermore, the same categorization is also used by Li et al. [11], which additionally attach a *cause* category to explain the reason (root) for each vulnerability exploit. Despite the similarity in categorizing the vulnerabilities with the same levels with the two aforementioned taxonomies, our taxonomy covers all discovered vulnerabilities so far, whereas they merely cover the most severe ones. Our results indicate that sometimes some minor issues exploited in crucial SC components can lead to huge losses. Thus, we stipulate the need to incorporate issues with low-severity as well in future taxonomies of vulnerabilities.

In the **second sub-question** on **why do these vulnerabilities occur and what are the factors that raise these vulnerabilities**, we partially rely on the taxonomies proposed and the systematic mapping study for smart contracts by Alharby and Moorsel [5]. They state that all smart contract vulnerabilities arise because of four reasons, including here:

- Codifying issues
- Security issues
- Privacy issues
- Performance issues

However, we argue that most of the already discovered security vulnerabilities occur with the ground basis of **codifying issues**, i.e. **developers' mistakes**. Even though in our taxonomy we categorize some vulnerabilities as EVM or blockchain specific, they are usually exploited by developers' mistakes, such as the case with the privacy-preserving vulnerability, which is usually due to developers failing to

apply cryptography or unintentionally making crucial functions/data public, which should have been private in the first place. Additionally, even though *transaction-ordering dependence* is a security issue concerning blockchain technology, it could be mitigated with the use of 'sendIfReceive' function or other preventive methodologies (e.g. guarded transactions) [7]. Furthermore, if some use cases of applications could not be implemented in Ethereum or blockchain generally, for a specific reason, those limitations should be taken into consideration by developers before they implement such an application, and not blaming the platform or the technology. Thus, generally speaking, we argue that the initial cause (factor) of all vulnerabilities is as a consequence of **developers' mistakes**.

The **third sub-question** concerns the concept of **'how can we mitigate these vulnerabilities and promote new engineering mindset for programming secure smart contracts'**. Throughout our research, in order to mitigate security vulnerabilities and generally to implement more secure and trustworthy smart contracts, we composed a list of proposals to follow. The list consists of:

- Following Ethereum smart contract best practices (recommendations), such as, the ConsenSys list [41]
- Using only documented and official (approved by Ethereum) programming languages and programming patterns
- Including SC test coverage
- Being up-to-date with ongoing Ethereum discussions (attacks, updates, modifications) through community forums (Reddit), chat groups (Gitter, Slack, etc.), and networking (communicating) with blockchain developers
- Developing smart contracts with a new engineering mindset, i.e., everything is public, immutable, and decentralized
- Utilizing the use of security code analysis tools
- In heavy financial smart contracts, utilizing the services of security audit firms

Overall, there is no single technology or method to put an end to security issues in Ethereum smart contracts. Thus, a developer should heavily rely on many sources and techniques simultaneously. The proposals mentioned above establish a safe base-knowledge to start developing securely on Ethereum.

Second Research Question: **What are the limitations with the current security code analysis tools in Ethereum?**

The answer to this question, somewhat constructs the answer for the **second sub-question** as well, which relates to the **possible future improvements within the tools**. Based on the limitations we identified, we propose several possible improvements. Thus, the answers to these questions are merged and jointly presented. The proposed improvements are a collection of the best practices and func-

tionalities from all security tools (based on our observations). The aim is to propose what an ideal/optimal security code analysis tool should include. The following limitations and possible improvements are:

User Interface

The method of how the security tools display and visualize the obtained results from security checks, presents the main limitation. Consequently, it becomes the most appropriate starting point for future improvements. First of all, all four analyzed security tools present the results in different ways. We choose the most convenient, again based on our observations, which could be used as future improvements in the rest of existing security tools or future new tools. **Results visualization** should mainly include the following:

- Highlight the line where the vulnerability is present
- Explain the vulnerability and the associated consequences
- Propose possible solution (with examples) for the detected vulnerability
- Provide links to relevant articles for further aid on solving or mitigating a particular vulnerability
- Provide the option for a user to *confirm* or mark a particular result as *false*
- Show the total (sum) number of detected vulnerabilities
- Save past-analysis results

For research purposes and other tool experiments, such as ours, it would be helpful if the security tools provide a before-hand **list with all the vulnerabilities** that they are supposed to identify. So far, this is done by SmartCheck (KnowledgeBase section), and Remix. The vulnerabilities that Oyente identifies, are covered in their paper [7] and we were not able to find Securify's list of vulnerabilities, apart from the general statement that they cover critical security vulnerabilities. Thus, to get the list from Securify we had to run one test analysis, and based on the results, to assemble a list of vulnerabilities they cover.

Furthermore, it is important for the security tools to boldly state which analysis they are able to run (bytecode or Solidity), and properly distinguish their usage (such as in Securify, Figure 15). On the other hand, Oyente, considering the fact that it uses an identical UI as Remix, it is not perfectly suitable for bytecode analysis and most importantly it is not self-explanatory, as it could be seen in Figure 13. We had to rely on developers' help to realize whether the web version of the tool supported bytecode analysis yet.

Generally, these usability (UI) issues make developers reluctant to embrace

the use of such security tools and have a negative impact on the overall adoption.

Vulnerability checks

Each security tool covers a distinct number of vulnerabilities. Some tools focus for example on several severe vulnerabilities, such as Oyente, and Securify. Whereas, other tools cover a more comprehensive list of vulnerabilities (including here, warnings, medium-severe, etc.), such as Remix and SmartCheck. Based on our experiment, we suggest that security tools should try and cover as many vulnerabilities as possible (low, medium and high). More specifically, considering the fact that a vulnerability which is referred to as a 'minor' issue – the **blockchash usage**, led to a theft of 400 Ether (approximately $200,000) from the "SmartBillions" smart contract. This vulnerability was **only** discovered by Remix, and all other three tools were not able to identify/discover the vulnerability because they consider it as a minor issue and do not include it in their checks, event though the hack showed otherwise. This further justifies our decision to include all vulnerabilities (low to high) in our taxonomy.

Furthermore, as we have already explained, some **severe** vulnerabilities, if used in non-crucial components of a smart contracts, do not present any risk from adversaries or hackers, such as the *timestamp dependence* vulnerabilities detected in the audited SCs. Therefore, it is helpful and beneficial to **state what severity level a specific vulnerability detected indicates**, and to avoid making a general assumption, as all the tools analyzed so far do.

Moreover, **compiler bugs**[1] are not covered by any security tool. It would be beneficial if the security tool, based on the compiler version that SCs use, report those potential Solidity compiler bugs. Also, considering the unpredictable state of smart contracts and Ethereum, **further work on detecting zero-days vulnerabilities**[2] would positively affect the tool performance.

Overall Observations

Two out of four tools, Oyente and Remix, have more than one website for the same tool. This causes confusion for users and developers. We had to contact the tool development teams to find out which websites were the official and legitimate ones. Having one website, i.e. one official source for the security tool, eliminates this unnecessary confusion and simplifies the use, in terms of accessibility. Other overall future improvements that could be made are:

[1] Link with the list of compiler bugs: https://etherscan.io/solcbuginfo
[2] Unknown vulnerabilities.

- Include a verification technique for the obtained results
- Measure the degree of testing for each SC analysis
- Provide tool-usage explanation[3]

Lastly, the **first sub-question** in the second research question is regarding the **current state of security tools based on their accuracy, effectiveness and consistency**. This question is fully answered through our conducted experiment which is presented in Chapter 6. However, it is worth mentioning several significant results. Oyente and SmartCheck were the only tools which achieved 100% on successfully analyzing all SCs. Since the accuracy of the tools was assessed for five vulnerabilities only, and not all tools identify those five vulnerabilities, it was challenging to obtain an overall accuracy. However, based on the results, Oyente seems to have the lowest false positive rate, whereas it is the only tool in which we detected one false negative. Nevertheless, almost 80% of the *timestamp dependence* vulnerabilities identified by Remix and SmartCheck, after the manual check, came out as true positives. This could imply that Oyente was not able to identify them, and these should be considered as false negatives. However, this is not necessarily the case because the timestamp issues identified are not severe and the reason for Oyente not identifying them might be that it only checks for severe issues[4]. As for the *reentrancy* vulnerability and the *unchecked-send bug*, Remix has the highest rate of false positives, followed by Securify in *transaction reordering* vulnerability checks, and SmartCheck in *reentrancy* analysis. Since none of the vulnerabilities (five chosen for manual checks) are covered by all four tools, we cannot draw general conclusions of the tools' accuracy.

As for the consistency assessment, Securify performed worse compared to Oyente in which we identified only one case of inconsistency. In contrast, in Securify we identified three cases of different results for the same SC analysis and also two cases of incorrectly recognizing the bytecode as a contract creation code.

Overall, the results indicate that some tools outperformed the others, but only in some specific aspects, not generally. SmartCheck's strongest points are: the results visualization, the high coverage of known vulnerabilities, and the effectiveness of successfully analyzing the whole data-set. Remix's strongest point is that the security analysis is incorporated within a compiler[5]. Whereas Oyente's strongest point

[3] Only in security tools in which you have to manually set up some arguments (timeout variable, gas limit, depth limit, etc.), such as in Oyente.

[4] Disputable statement.

[5] We argue that it is more convenient for developers to have the security analysis incorporated within a compiler.

is on the accuracy level due to having the lowest false positive rate compared to the other tools. Since we had limited access to Securify, the results are also limited. However, Securify's weakest point is the ability to analyze the data-set provided because it failed multiple times and performed worse than any other tool in this task.

Generally speaking, despite the fact that the same data-set was used for all four tools analysis, their results are vastly dissimilar. Thus, this indicates that there is a general inconsistency within the tools and the results they produce. However, these security tools, at this stage, should be used as an aid (in addition to our proposals for safe smart contract programming) for increased security and developers should not rely on them 100%.

7.1 Limitations

This section covers the project's limitations and the main pitfalls we encountered throughout our experiment.

Since Ethereum and the security of its smart contracts especially are on their infancy stage, the primary limitation of this thesis is the **lack of high-quality and peer-reviewed papers**. Therefore, this thesis relies mostly on grey literature and web articles from trusted sources. Another significant limitation which affected the results of our experiments, is that most of the **security tools** we analyzed (3 out of 4), **are in their beta-versions**, except for Remix, which is considered to be in a more stable state.

For example, Oyente WUI version was recently released. As such, a major limitation that effected Oyente's results in general is the **improper function of the 'timeout' variable**. When we set the 'timeout' variable to 30 minutes, the analysis failed[6] after some time. When we set the timeout variable to less (e.g. 2-5 minutes), the tool did not detect the vulnerability. This might also be the reason why Oyente did not identify the timestamp dependencies in the audited SCs[7]. Overall, the limitation of early-versions (i.e. unstable) of security tools led to other restrains, such as, the security tools experiencing unidentified issues, and sometimes acting strangely.

A significant limitation regarding the taxonomies we proposed is based on their validation. At this stage, we *assume* based on our research and the results obtained that the proposed taxonomies would aid developers in mitigating some already known vulnerabilities, which is not conclusive. Thus, currently this thesis **lacks the validation process and the overall community-acceptance** of these taxonomies.

[6] Error: Some errors occurred. Please try again!

[7] In a discussion with Oyente's developer, they proposed to use a timeout of 2000 seconds for one specific SCs.

A limitation concerning the experiment is the fact that **all security tools approach the SC analysis in different ways, with a variety of functionalities**. Therefore, it was challenging and unfeasible to do a direct comparison of the tools and state which one performed better. This is due to them not identifying the same vulnerabilities, as well as producing distinct results. More importantly, these tools were still considered a work-in-progress, and as such, not widely common in practice.

Additional limitation related to the experiment results, is that the majority (19 out of 23) of SCs from the **vulnerable SCs data-set are old**, i.e., use old compiler versions and deprecated techniques which are prone to errors. As such, most of these SCs are identified as vulnerable with many issues from all four tools. This is the reason why we decided to assess only the false negative rate in the vulnerable SCs category. This also indicates that the compiler-version had a major impact on the results obtained from the vulnerable SCs category.

The **small data sample** did not affect our results. However, the results would have been more comprehensive if we had several other vulnerable SCs with some specific severe issues. I.e., the results would have been more meaningful and supplementary expressive if we had a more structured data-set for the vulnerable SCs category. Furthermore, we used **audited SCs** for the other category, **which does not necessarily mean they are secure and bug-free**. Even though we took precautions to eliminate the possible vulnerable SCs from the audited SCs category, we still had to perform the manual analysis to ensure for the validity of the results.

Moreover, the data sample has four vulnerable SCs with the issue of 'contracts that do not refund', which represents around 17% of the total category of vulnerable SCs. Considering the fact that this is an old vulnerability (occurring only in SCs with old compilers), the security tools do not run such checks. Thus, the high percentage of 'not supposed to identify' vulnerabilities for each tool, was significantly influenced by this.

To conclude, the aforementioned limitations present a challenge on replicating the experiment and obtaining the same results as us, for the sole reason of the security tools being in their beta versions, i.e. under heavy development work and experiencing ongoing changes, which means that they are continuously upgrading and becoming more efficient in detecting security vulnerabilities.

8 Conclusion

Ethereum is a relatively new platform, launched officially in June 2015. It introduces the concept of smart contract applications in a blockchain technology. Since Ethereum is a decentralized, public, and most importantly an immutable platform – it requires a novel engineering mindset from developers. Smart contracts are self-executable and self-verifying agents which cannot be altered once they are deployed in the blockchain. As a result, they are vulnerable and a constant target for adversaries. Motivated by security breaches and financial losses that Ethereum smart contracts have experienced so far, academia and the industry have directed their attention towards the security of smart contracts. Similarly, the purpose of this thesis was to investigate the security vulnerabilities on Ethereum smart contracts and assess the overall effectiveness of several popular security code analysis tools used to detect those vulnerabilities. Contributing to a more secure and trustworthy Ethereum environment was the main motivation behind this work.

We have conducted a comprehensive research on peer-reviewed papers and online websites. The research outcome was two proposed taxonomies. The first taxonomy outlines already exploited vulnerabilities and classifies them based on their architectural and severity level. It serves as a list of issues that can aid developers who plan to develop smart contract applications. The second one, to the best of our knowledge, is a novel taxonomy of current security tools. We classified the tools based on the methodology they use, the user interface, and the analysis they are able to execute, which allows us to build a 'state of the art' of security tools on Ethereum. Lastly, we construct a matrix of security tools and the vulnerabilities they cover in order to identify gaps and absent vulnerability checks. Moreover, we provide a scheme with a list of suggestions to follow, for the purpose of avoiding and mitigating the deployment of vulnerable smart contracts. We also developed an experiment on the most popular security tools and assess their effectiveness, accuracy, and consistency.

Our taxonomies are an improvement of the existing ones because they take into consideration vulnerabilities of all severity levels. This approach is advantageous because the consequences of a vulnerability cannot always be predicted. Therefore, our taxonomies include low severity level vulnerabilities that may potentially cause a lot of damage if exploited in crucial smart contract components. On the other hand, the results from our experiments differentiate the security tools according to different properties. In terms of effectiveness, SmartCheck outperformed the other

tools with a score of 100% by successfully analyzing both vulnerable and audited SCs. Oyente, on the other hand, successfully analyzed the complete data-set on bytecode analysis but performed slightly worse on Solidity analysis. When assessing accuracy, Oyente performed generally better than the rest of the tools although it did not perform well on the false negative rate. Finally, Securify performed worse compared to Oyente when assessing consistency.

Our experiment shows that the current performance of the existing security tools is not ideal. The tools being in their beta-version may be one indication as to why they do not cover the full range of vulnerabilities outlined in the thesis. Additionally, the tools were under heavy development during the experiment which may have affected their overall performance. Being in their infancy, the tools have promising potential to evolve further. Thus, based on the drawbacks identified throughout our experiment, we propose a list of possible future improvements within: the user interface (i.e. results visualization), verification techniques, and vulnerability checks.

To the best of our knowledge, this is the first attempt on evaluating the security analysis tools on Ethereum. We acknowledge that there is extensive room for improving security and vulnerability detection in smart contracts. The contribution of our results is twofold in that: (1) it will help developers of security tools with their future developments, and (2) advise smart contract developers on mitigating security errors. To conclude, we quote the founder of Ethereum:

> "There will be further bugs, and we will learn further lessons; there will not be a single magic technology that solves everything". - Vitalik Buterin [27]

8.1 Future Work

This thesis proposes several potential future work studies and experiments. First of all, since the whole experiment is manually conducted and it proved to be labour-intensive, an **automated test environment** would simplify the process, reduce the amount of time needed, and overall improve the procedure performance. Additionally, considering the rapid advancement of security tools and smart contracts, **the experiment could be replicated once the security tools are in a more stable version and with a larger, more comprehensive data-set**.

As for the proposed taxonomies, we propose future work on **validating the generated taxonomies** through an experiment with SC developers or through a university course in programming. This would evaluate the effectiveness of the taxonomies and identify potential avenues for changes.

Further work could be conducted on **discovering the unknown vulnerabilities** and how we can prevent them, i.e. have a **self-protection (revocable) mechanism** for smart contracts in order to automatically take some precautions if an attack occurs. This would be highly important work, considering that once the smart con-

tract is deployed, the owner can no longer alter it.

Moreover, we consider the incorporation of tests to be a bare minimum criteria for application security and thus recommend future work on **analyzing all deployed smart contracts in Ethereum and evaluating how many of them incorporate tests**. Also, we briefly discussed that there are multiple attempts to provide a categorization of the range of possibilities for the use of smart contracts. A future project could be to **generate a list of application domains for Ethereum SCs** based on a **systematic analysis of all current-deployed smart contracts**.

Lastly, Ethereum users often become victims of various Ponzi schemes which negatively stain Ethereum's philosophy of a trustworthy environment. In the scope of our work, we only focus on the security of smart contracts from developers' point of view. As an effort to decrease the negative impact of Ponzi schemes on users' adoption of Ethereum, it would be beneficial to **assess the security of smart contracts from a users' point of view**. Thus, we argue that the users would benefit from **a blockchain explorer which checks if smart contracts behave as intended**. This would aid users in mitigating scams, and backdoor options as well as provide them with comprehensible information by e.g. translating the bytecode to op-code or an alternative approach.

To conclude, smart contracts hold valuable digital assets but are simultaneously vastly vulnerable even to simple development errors. Therefore, it is vital for academia and the industry to engage in future work which will enhance the security of smart contract programming and assist towards a more secure and trustworthy Ethereum environment.

Bibliography

[1] Zheng, Z., Xie, S., Dai, H.-N., & Wang, H. 2016. Blockchain challenges and opportunities: A survey. *Work Pap*.

[2] Dannen, C. 2017. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition.

[3] Nakamoto, S. 2008. Bitcoin: A peer-to-peer electronic cash system.

[4] Szabo, N. 1997. Formalizing and securing relationships on public networks. *First Monday*, 2(9).

[5] Alharby, M. & van Moorsel, A. 2017. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372*.

[6] GitHub. Problems - ethereum/research wiki. https://github.com/ethereum/research/wiki/Problems. (Accessed on 11/12/2017).

[7] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 254–269. ACM.

[8] Karst, J. J. & Brodar, G. 2017. Connecting multiple devices with blockchain in the internet of things.

[9] Egbertsen, W., Hardeman, G., van den Hoven, M., van der Kolk, G., & van Rijsewijk, A. 2016. Replacing paper contracts with ethereum smart contracts.

[10] Christidis, K. & Devetsikiotis, M. 2016. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4, 2292–2303.

[11] Li, X., Jiang, P., Chen, T., Luo, X., & Wen, Q. 2017. A survey on the security of blockchain systems. *Future Generation Computer Systems*.

[12] Buterin, V. et al. 2014. A next-generation smart contract and decentralized application platform. *white paper*.

[13] Delmolino, K., Arnett, M., Kosba, A., Miller, A., & Shi, E. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, 79–94. Springer.

[14] Wood, G. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151.

[15] Fröwis, M. & Böhme, R. 2017. In code we trust? In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 357–372. Springer.

[16] BlockGeeks. What is ethereum classic? ethereum vs ethereum classic - blockgeeks. https://blockgeeks.com/guides/what-is-ethereum-classic/. (Accessed on 11/16/2017).

[17] Bartoletti, M., Carta, S., Cimoli, T., & Saia, R. 2017. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*.

[18] Solidity. Introduction to smart contracts — solidity 0.4.19 documentation. http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html. (Accessed on 11/17/2017).

[19] Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., & Rosu, G. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017.

[20] ConsenSys. A 101 noob intro to programming smart contracts on ethereum. https://medium.com/@ConsenSys/a-101-noob-intro-to-programming-smart-contracts-on-ethereum-695d15c1dab4. (Accessed on 11/17/2017).

[21] Zeppelin. Serpent compiler audit – zeppelin. https://blog.zeppelin.solutions/serpent-compiler-audit-3095d1257929. (Accessed on 11/17/2017).

[22] Viper. ethereum/viper: New experimental programming language. https://github.com/ethereum/viper. (Accessed on 11/17/2017).

[23] Atzei, N., Bartoletti, M., & Cimoli, T. 2017. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, 164–186. Springer.

[24] Bartoletti, M. & Pompianu, L. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. *arXiv preprint arXiv:1703.06322*.

[25] Clack, C. D., Bakshi, V. A., & Braine, L. 2016. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*.

[26] Juels, A., Kosba, A., & Shi, E. 2016. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 283–295. ACM.

[27] Buterin, V. June 2016. Thinking about smart contract security - ethereum blog. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/. (Accessed on 11/18/2017).

[28] Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*, 839–858. IEEE.

[29] Boneh, D. & Naor, M. 2000. Timed commitments. In *Advances in Cryptology—Crypto 2000*, 236–254. Springer.

[30] Marino, B. & Juels, A. 2016. Setting standards for altering and undoing smart contracts. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, 151–166. Springer.

[31] Chen, T., Li, X., Luo, X., & Zhang, X. 2017. Under-optimized smart contracts devour your money. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, 442–446. IEEE.

[32] Zhang, F., Cecchetti, E., Croman, K., Juels, A., & Shi, E. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 270–282. ACM.

[33] Solidity. Security considerations — solidity 0.4.19 documentation. http://solidity.readthedocs.io/en/latest/security-considerations.html. (Accessed on 11/19/2017).

[34] Ethereum-Wiki. Safety · ethereum/wiki wiki. https://github.com/ethereum/wiki/wiki/Safety. (Accessed on 11/19/2017).

[35] SmartDec. Smartcheck | knowledgebase | dos by external contract. https://tool.smartdec.net/knowledge/SOLIDITY_DOS_WITH_THROW. (Accessed on 12/14/2017).

[36] KotET. Kotet - post-mortem investigation. http://www.kingoftheether.com/postmortem.html. (Accessed on 11/20/2017).

[37] GovernMental. Governmental - ponzi scheme. http://governmental.github.io/GovernMental/. (Accessed on 11/20/2017).

[38] Memoria, F. October 2017. Smartbillions challenges hackers with 1,500 ether reward - cryptocoinsnews. https://www.cryptocoinsnews.com/smartbillions-challenges-hackers-1500-ether-reward-gets-hacked-pulls/. (Accessed on 11/20/2017).

[39] Olickel, H. June 2016. Why smart contracts fail: Undiscovered bugs and what we can do about them. https://medium.com/@hrishiolickel/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007. (Accessed on 11/20/2017).

[40] Araoz, M. July 2017. Introducing zeppelinos: the operating system for smart contract applications. https://blog.zeppelin.solutions/introducing-zeppelinos-the-operating-system-for-smart-contract-applications-82b042514aa8. (Accessed on 11/20/2017).

[41] ConsenSys. Recommendations for smart contract security in solidity - ethereum smart contract best practices. https://consensys.github.io/smart-contract-best-practices/recommendations/. (Accessed on 11/21/2017).

[42] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., & Zanella-Beguelin, S. 2016. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16*, 91–96.

[43] Cook, T. W., Latham, A., & Lee, J. S. 2017. Dappguard: Active monitoring and defense for solidity smart contracts.

[44] KotET. Contract safety checklist. http://www.kingoftheether.com/contract-safety-checklist.html. (Accessed on 11/23/2017).

[45] Tikhomirov, S. 2017. Ethereum: state of knowledge and research perspectives.

[46] Zeppelin. Security audits. https://zeppelin.solutions/security-audits. (Accessed on 11/25/2017).

[47] CryptoNews. Ccn: Bitcoin, ethereum, neo, ico & cryptocurrency news. https://www.cryptocoinsnews.com/. (Accessed on 11/25/2017).

[48] HackingNewsWebSite. Hacking, distributed. http://hackingdistributed.com/. (Accessed on 11/25/2017).

[49] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. Attacks — a survey of attacks on ethereum smart contracts. http://blockchain.unica.it/projects/ethereum-survey/attacks.html. (Accessed on 11/25/2017).

[50] Zhang, Y., Egelman, S., Cranor, L., & Hong, J. 2006. Phinding phish: Evaluating anti-phishing tools. ISOC.

A Appendix

A.1 Security Tools User Interfaces

Figure 13: Oyente-Web User Interface

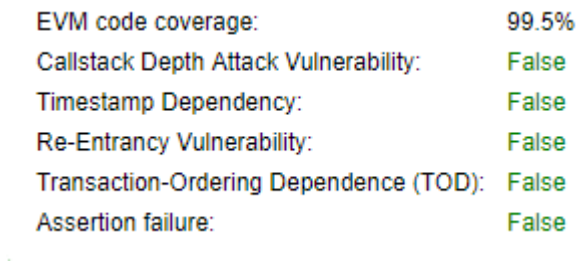

Figure 14: Oyente-Web Results - Visualized

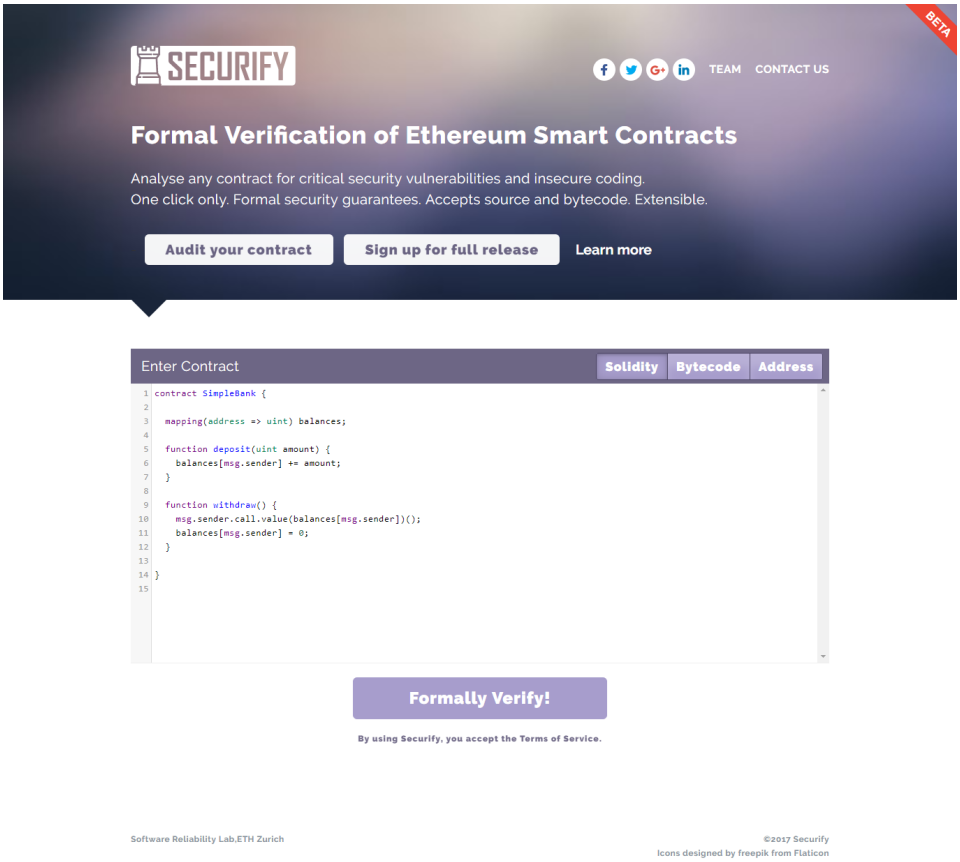

Figure 15: Securify User Interface

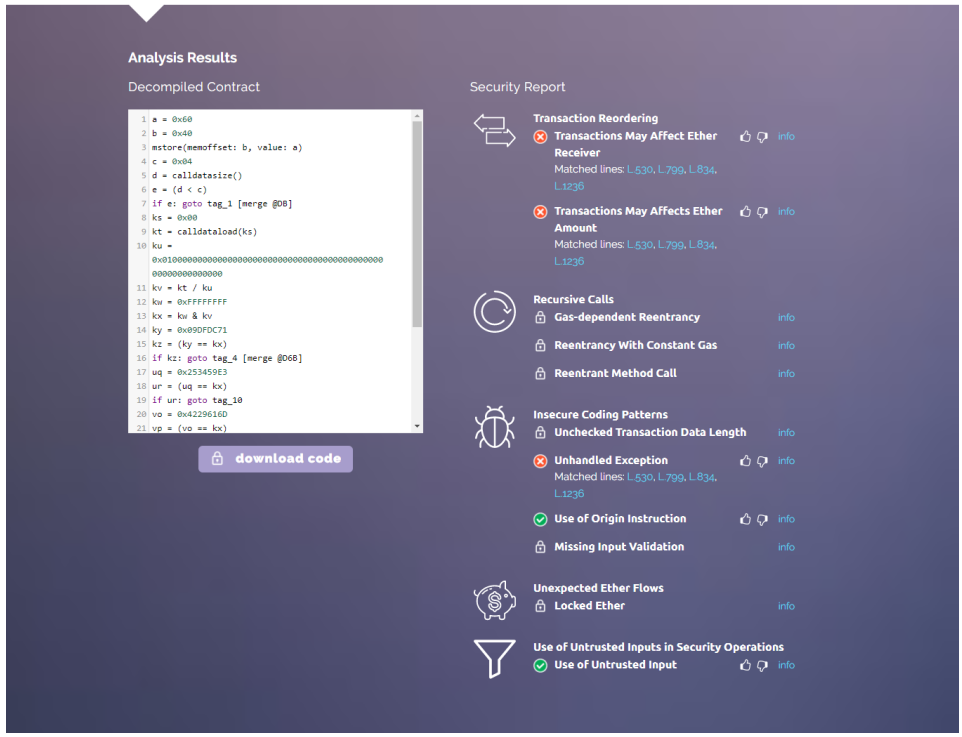

Figure 16: Securify Results - Visualized

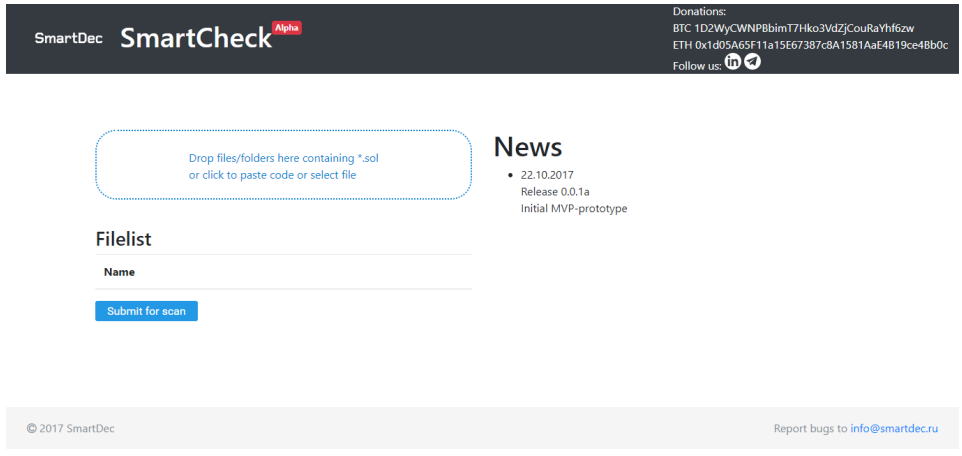

Figure 17: SmartCheck User Interface

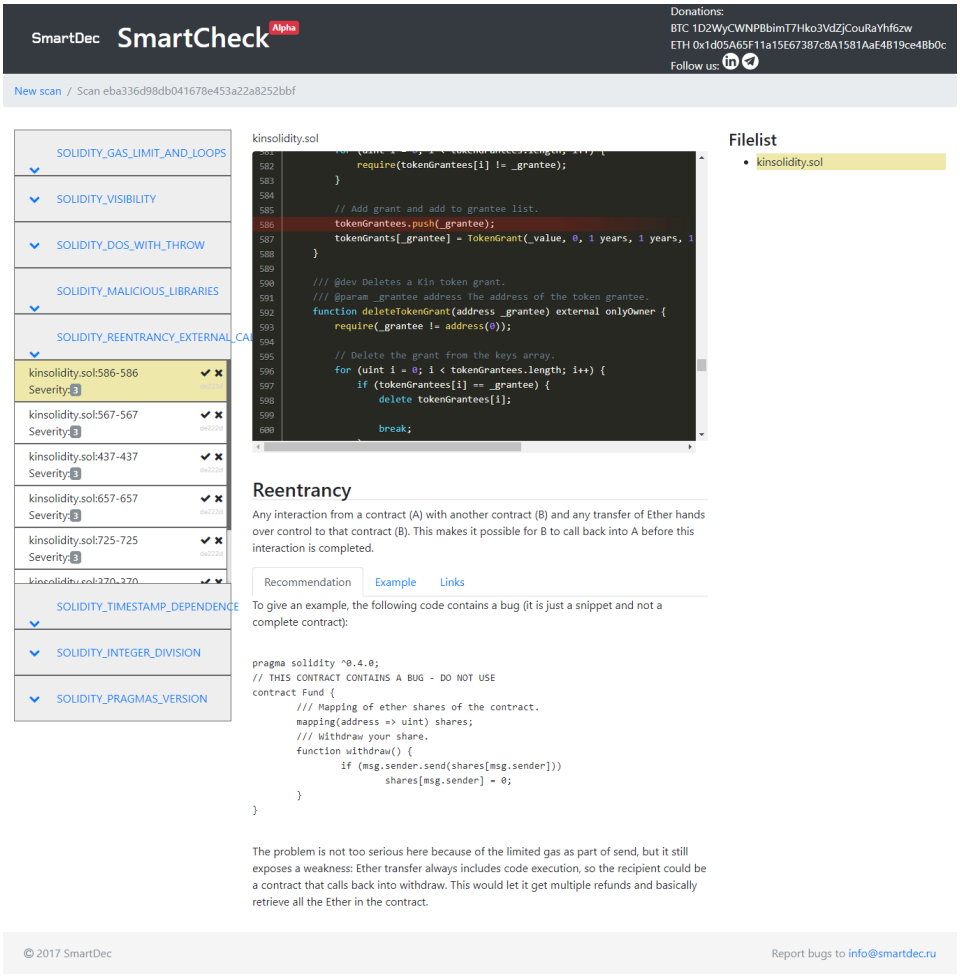

Figure 18: SmartCheck Results - Visualized

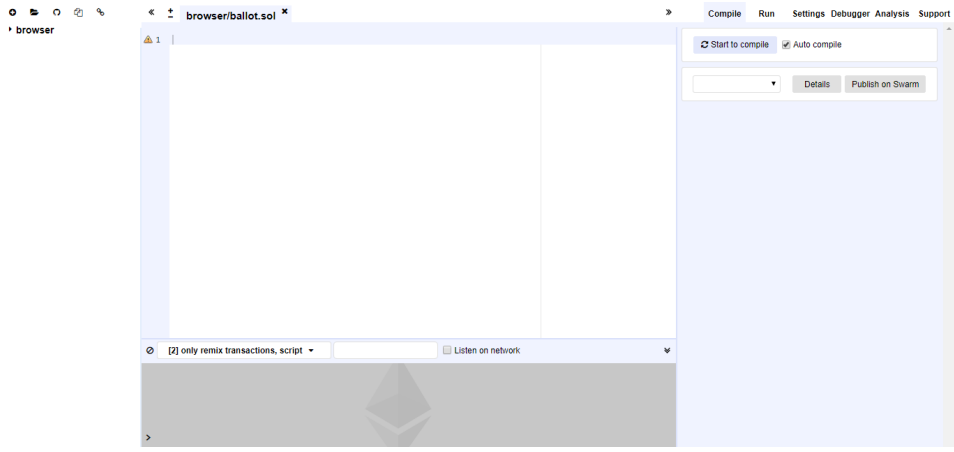

Figure 19: Remix User Interface

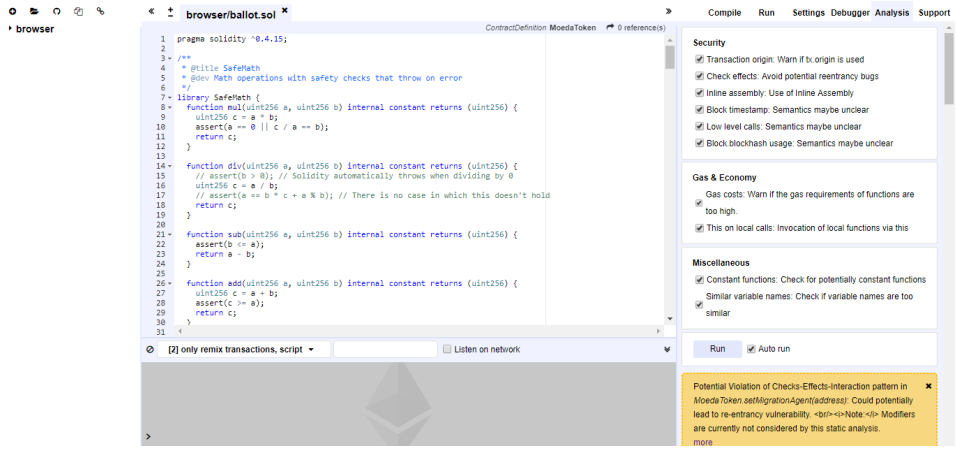

Figure 20: Remix Results - Visualized

A.2 SolC Set Up

SolC is used in Ubuntu through Oracle VirtualBox in a Windows machine. Set-up steps that we followed are provided in these links:

- https://www.npmjs.com/package/solc#usage-on-the-command-line
- http://blog.teamtreehouse.com/install-node-js-npm-linux
- http://linuxbrew.sh/
- http://solidity.readthedocs.io/en/develop/using-the-compiler.html

A.3 Accuracy Analysis - Raw-Data Results

NR.	Vulnerable SCs	Vulnerability	Security Tool			
			Oyente	Remix	SmartCheck	Securify
1	**SimpleDao Sample compiler version 0.3.1**	Re-entrancy, call to the unkown	YES	YES	YES	NA
2	**SimpleDao Sample compiler version 0.4.2**	Re-entrancy, call to the unkown	YES	YES	YES	NA
3	**King of the Ether game (KoET)**	Unchecked-send bug, Gasless send, Mishandled exception	YES	YES	YES	YES
4	**KotET Sample compiler version 0.3.1**	Gasless send	YES	YES	YES	YES
5	**KotET Sample compiler version 0.4.2**	Gasless send	YES	YES	YES	YES
6	**GovernMental (PonziGovernmental)**	Unchecked-send bug, Call-stack limit	YES	YES	YES	YES
7	**GovernMental simplified sample 0.3.1**	Immutable bugs, exception disorder, call-stack limit	YES	YES	YES	YES
8	**Rubixi**	Immutable bugs, wrong constructor name	NA	NA	NA	NA
9	**FirePonzi**	Type casts (intentional scam)	NA	NA	NA	NA
10	**Parity Multisig**	Unintended function exposure	NA	YES	YES	NA
11	**GoodFellas**	Typo (wrong constructor name)	NA	NA	NA	NA
12	**StackyGame**	Typo (wrong constructor name)	NA	NA	NA	NA
13	**DynamicPyramid**	Contract that do not refund	NA	NA	NA	NA
14	**GreedPit**	Contract that do not refund	NA	NA	NA	NA
15	**NanoPyramid**	Contract that do not refund	NA	NA	NA	NA
16	**Tomeka**	Contract that do not refund	NA	NA	NA	NA
17	**ProtectTheCastle**	Call-stack limit, Withdraw option	YES	NA	NA	NA
18	**RockPaperScissors (RPS)**	Public moves	NA	NA	YES	NA
19	**SmartBillions**	Blockhash bug	NA	YES	NA	NA
20	**EtherPot**	Unchecked-send bug	NA	YES	YES	YES
21	**TheRun**	Timestamp dependence	No	YES	YES	NA
22	**OddsAndEvents Compiler 0.3.1 Sample**	Keeping secrets	NA	NA	YES	NA
23	**OddsAndEvents Compiler 0.4.2 Sample**	Keeping secrets	NA	NA	YES	NA

Table 23: False Negative Analysis - Raw-data Results

False Positives Analysis	Re-entrancy			Timestamp Dependence			Transaction reordering		Unchecked-send bug/ Unchecked call	
	Oyente	Remix	SmartCheck	Oyente	Remix	SmartCheck	Oyente	Securify	Remix	SmartCheck
Hacker Gold (HKG)	No	No	No	Yes	Yes	Yes	No	Yes	Yes	No
ArcadeCity (ARC)	No	Yes	No	No	No	No	No	Yes	Yes	No
Golem Network	NA	No	Yes	NA	Yes	Yes	NA	NA	Yes	No
ProjectKudos	No	No	No	No	Yes	Yes	No	NA	No	No
SuperDAO Promissory	No	Yes	Yes	No	No	No	No	NA	Yes	No
ROSCA	NA	No	No	NA	NA	Yes	NA	NA	NA	Yes
Matchpool GUP	No	No	No	No	Yes	Yes	No	No	No	No
iEx.ec RLC	No	No	No	No	No	No	No	No	No	No
Cosmos	No	Yes	No	No	No	No	No	Yes	Yes	No
Blockchain Capital (BCAP)	No	No	No	No	Yes	No	No	No	No	No
WingsDAO	No	No	No	No	No	Yes	No	No	No	No
Moeda	No	Yes	Yes	No	No	No	No	No	No	No
Basic Attention	No	No	No	No	No	No	Yes	Yes	Yes	No
Storj	No	Yes	Yes	No	No	No	No	No	No	No
Metal	No	No	No	No	No	No	No	No	No	No
Decentraland MANA	No	No	No	No	No	No	No	No	No	No
Tierion	No	No	No	No	No	No	No	No	No	No
Kin	No	Yes	Yes	No	Yes	Yes	No	Yes	No	No
Fuel	No	No	No	No	Yes	Yes	No	No	No	No
Enigma	No	No	No	No	No	No	No	No	No	No
Global Messaging	NA	No	YES	NA	NA	No	NA	No	NA	No

Table 24: False Positive Analysis - Raw-data Results