University of Southampton

Faculty of Engineering and Physical Sciences

Electronics and Computer Science

Analysis of Ethereum Smart Contracts Vulnerabilities

By

Mansur Aliyu

September 2019

Supervisor: Dr. JingHuey Khor

Second Examiner: Prof. Lie-Liang Yang

A dissertation submitted in partial fulfilment of the degree

of MSc Cyber Security

Abstract

Smart contracts are program logics running on the Ethereum Blockchain platform aimed at ensuring automation of business transactions. Like many other software systems, smart contracts are experiencing security vulnerabilities due to some weaknesses of the programming languages used to design them. Over 90% of the published security vulnerabilities in the Common Vulnerabilities Exposure (CVE) database are due to arithmetic issues. Current practices concentrate on providing detective approaches to flag them such as Oyente, SmartCheck, Securify, and Mythril. The tools sometimes yield false-positives. However, a preventive tool called SafeMath library has been developed. The library is quite effective in mitigating arithmetic overflow but can incur overhead costs called gas charges. The reduction of the cost is crucial in order to reduce out-of-gas security vulnerabilities that can affect availability. In this dissertation, an optimized SafeMath library was developed which offers low gas cost whilst maintaining current SafeMath security protections. Using appropriate security analysis tools, our proposed approach is evaluated, and findings indicate high effectiveness and proven to be correct with state-of-the-art smart contract testing tools and penetration testing. The modified SafeMath performs better by 3% than the original SafeMath when tested with a real-world smart contract.

Acknowledgements

I thank God almighty for making the whole journey possible. I also would like to appreciate my supervisor, Dr. JingHuey Khor for invaluable insights, guidance, and recommendations that make the work a lot better. Also, the remarks at the end of the demonstration by the second Examiner, Prof. Lie-Liang Yang were helpful and encouraged me to improve the presentation of the work.

A great thanks to the reviewers, particularly, Dr. Aminu Yakubu and Imrana Abdullahi Yari, your criticisms and recommendations were invaluable.

I must also thank my sponsor for a very generous funding, the Petroleum Technology Development Fund (PTDF) and similarly my employer, Kebbi State University of Science and Technology, Aliero for permitting me to leave the workplace to come to study.

Finally, I must thank my family, particularly two of my children (Sumayya and Ahmad) for endurance, spiritual support, and great patient while I am away and of course their mum too, Sarah.

Thank you very much everyone indeed.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Background

Smart contracts are self-executing programs aimed at removing third parties between two or more people making some transactions on the Ethereum [1], [2]. The absence of a third party ensures automation of transactions and accountability. For instance, an artist publishing song through blockchain and manages the sells through smart contracts don't have to worry about the marketing costs and has confidence in return of investment as there is openness as to who buy the published songs.

The term has been coined since 1997 before the advent of blockchain by Nick Szabo [3]. An Ethereum is a blockchain platform becoming more popular over earlier blockchain platforms such as bitcoin. One important feature of Ethereum which has become quite popular over Bitcoin is that its execution is Turing complete [1], [4]. A Turing complete means it can distinguish and understand two or more distinct data manipulation or rule sets. A blockchain is a technology that links the records of transactions between multiple and mutually distrusting parties through a chain of encrypted hashes known as a Distributed Ledger Technology (DLT) or public ledger in short [5].

However, smart contracts are exposed publicly through Etherscan after deployment on the blockchain [5]–[7]. An Etherscan is a platform that keeps statistics of the deployed smart contracts. The deployed smart contracts on the blockchain have predefined functions that are often exposed and are therefore exploitable and vulnerable by attackers [1]. Smart contracts hold financial transaction information. As a result, their correct execution is important. Typically, a well-known Decentralised Autonomous Organisation (DAO) attack which engulfed $60m [2] arises as a result of exposure of smart contracts and their exploitation by attackers [1], [2], [4], [5], [8]. The DAO is an organisation focusing in the future of blockchain technology [9], [10].

Current practice to mitigate security vulnerabilities are mostly detective tools designed to flag-out vulnerable section of the code written in Solidity. The number of state-of-the-art techniques concentrates mostly on detective mechanisms which are mostly through a static analysis, not protective ones. But one popular protective mechanism employed is the use of a library called the *SafeMath* [11] which prevents the occurrence of the arithmetic vulnerabilities automatically.

Nevertheless, the execution of the smart contract is associated with some cost whenever it is executed there is a considerable amount of incentives to be paid by the owner called 'gas cost' which is measured in a unit called 'wei'. The gas cost is introduced to avoid a number of malicious users who might be deploying smart contracts blockchain and or execution some function calls between smart contracts deployed in order to create too much load that could lead to Denial of Service (DoS). But, the addition of the *SafeMath* library to mitigate blockchain leads to an increase in the gas costs. Whenever the gas cost exceeds the allocated 'gas limit' specified by the developer, the smart contract would not be executed. The gas value and gas cost are the actual cost incurred by the execution of the smart contract and the initial value set up by the developers of smart contract respectively. There are two categories of gas costs namely; the transaction and the execution. A transaction cost happens when deploying a contract to the blockchain and an execution cost is when a transaction occur [12].

The empirical analysis of this library in terms of security and performance in terms of gas cost is the focus of this dissertation research project.

1.2 Problem Statement

Despite the emerging success of the smart contracts, there is quite a number of security vulnerabilities arising such as an integer overflow and/or underflow, out of the gas, re-entrancy, call to unknown, etc. [1].

As at the time of writing this work, out of 525 published security vulnerabilities in the Common Vulnerabilities Exposure (CVE) database related to Ethereum smart contracts, up to 486 (92.6%) are related to arithmetic flaws like integer overflow and/or underflow also known as the two's complement [13]. These indices clearly indicate the importance of the arithmetic vulnerabilities on smart contracts. Similarly, the arithmetic vulnerabilities are listed among the top ten security issues in smart contract applications [14].

1.3 Motivation

As earlier explained, there is a commonly used library called the *SafeMath* used by developers to mitigate arithmetic flaws, there are key interesting issues here:

- The library consumes more gas. An efficient mechanism to create an optimised or low gas cost smart contracts is desirable because heavy-weight smart contract could affect the availability of data or record on the blockchain.
- However, If the arithmetic vulnerability is exploited it could lead to compromise to the integrity of blockchain data. While the *SafeMath* attempts to render integrity, the underlying costs associated could lead to the availability being lost without which an integer overflow can lead the adversary to steal money [15].

The trade-off between these two-security requirements is paramount important to any modern information system employing the Ethereum smart contract technologies.

1.4 Aim and Objectives

The aim of this research project is to analyse the security vulnerabilities in Ethereum smart contracts, in order to mitigate them through analysing the *SafeMath* library and optimise it to provide better performance and security protection.

In order to achieve the aim, the following objectives need to be attained.

1. To analyse the *SafeMath* library in terms of performance as well as security weaknesses.

2. To design a new library to improve performance and security protection of the existing *SafeMath* library.

3. To evaluate the optimised library in terms of security and performance.

Chapter 2: Research Management

2.1 Introduction

This chapter discusses the stages of the tasks planned before carrying out the research project. It also shows the actual time used. To avoid the occurrence of risks, some contingency, and risk assessment planning were considered. The cost analysis and issues encountered were finally stated.

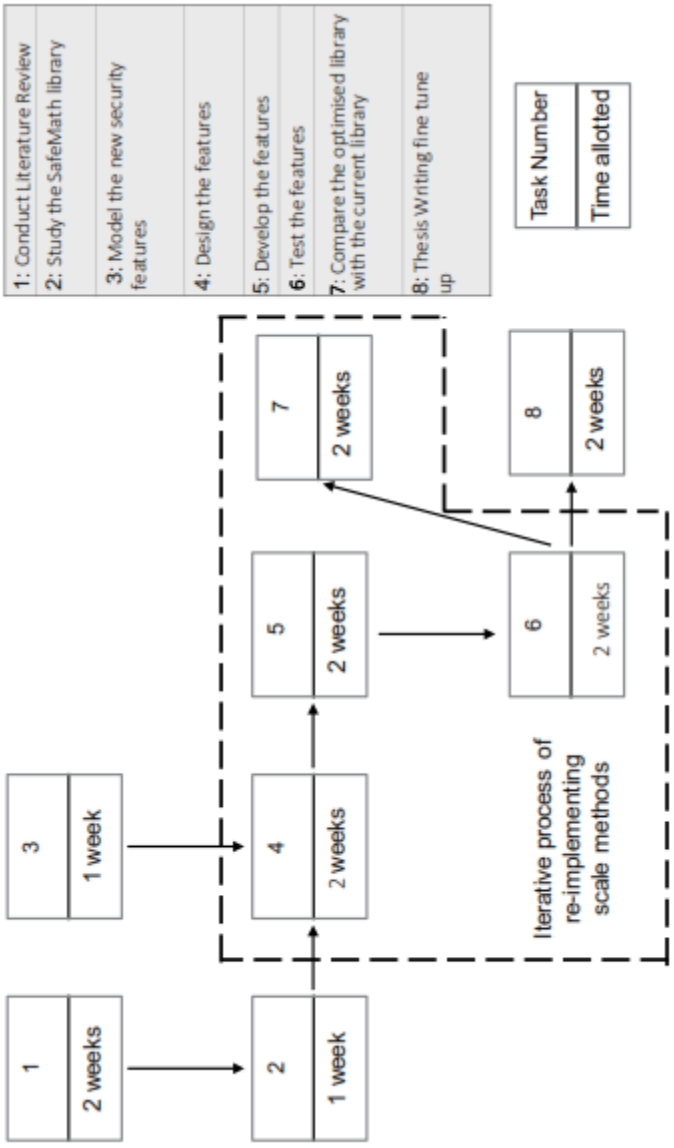

Figure 2. 1: Pert Chart demonstrating tasks linkages

2.2 Project Management and Reflections

2.2.1 Tracking, Project Management, and Contingency plans

The Program Evaluation Review Technique (PERT) chart shown on Figure 2.1 shows the linkages of the series of activities carried out for this dissertation project. From the figure, more time was given to literature review for the fact that this is an interesting new area and more activities need to be studied from scratch such as learning the syntaxes and semantics of Solidity programming and the working environment set up. Due to the nature of the work, the PERT chart was chosen in order to start a number of tasks concurrently in order to speed up the research work.

Although it tends out that, the literature review (Task 1) continued to be a continuous process throughout the work and was reassessed several times in order to validate facts and to get new methods for the analysis and testing of smart contracts. So, the first two weeks were starting point. The actual time spent was stated in Table 2.1 below. A similar time was allocated to the modeling new *SafeMath* library (Task 3) alongside studying the existing one. Moreover, the development of the modified library (Task 5) and testing of the functionality and security of the existing *SafeMath and* the optimised library (Task 7) took about a month due to the commitments on the familiarisations with the number of tools used. These are discussed in Chapters 4 and 5.

The remaining even-numbered four tasks (Tasks 2 and 4) were allocated one week each. This is because they were fully dependent upon those tasks with long time allocation than those odd-numbered tasks except Task 6 and 8, which has to do with rigorous tests and updates to the writing respectively were given two weeks each. As soon as a task is confirmed to be achieved, a note was taken to a tracking table as presented in Table 2.1 below under the remark column so that the next task is started. As can be observed, there are some changes to the actual time spent to execute planned tasks.

The note-taking was efficiently used throughout when analysing literature when conducting technical analysis of the *SafeMath* library as well during a series of meetings with the supervisor. The important notes taken has really helped in improving the quality of the work and allow the project work to be managed effectively.

Whereas the project work is supposed to take a period of fourteen (14) weeks period, the research was planned for one week less for contingency. Similarly, part of the contingency plans or risk planning was the backing up of all codes. Although, the Remix compiler used,

save the works automatically, *GitHub* repository was used to keep all codes (Appendix A). This is very crucial to avoid data loss abruptly.

Table 2. 1: Tracking of work completed

Task	**Start time**	**Finish time**	**Duration**	**Remarks**
Task 1: Literature Review	10 June 2019	26 June 2019	11 weeks	Completed beyond time scheduled
Task 2: Studying the S*afemath* Library	10 June	17 June	1 week	Completed on time scheduled
Task 3: Modelling new security features	17 July	24 June	1 week	Completed on time scheduled
Task 4: Designing the Features	24 June	8 July	2 weeks	Completed on time scheduled
Task 5: Developing the Features	8 July	22 July	2 weeks	Completed on time scheduled
Task 6: Testing the Features	22 July	5 August	2 weeks	Completed on time scheduled
Task 7: Comparative analysis	5 August	19 August	2 weeks	Completed on time scheduled
Task 8: Update to the Write-Up	19 August	4 September	2 weeks	Completed beyond time scheduled

2.2.2 Cost Management

Virtually all the tools and software used are free and open-source, as a result, there is no financial cost associated.

Nonetheless, in terms of other costs, like the time to execute some of the work, only one tool delayed in given results by taking a long time to understand and to analyse smart contracts. The tool is called *Mythril*. Other than that, all other tools used are almost seamlessly used. The only commitment made to manage time efficiently is the use of some short tutorials from the LinkedIn learning academy subscribed by the University which is also free of use. The academy has really helped in learning basic and concepts to help understand the complex ones on the research articles and the source codes that were analysed from the *GitHub* repository.

2.2.3 Reflections

The literature review and technical skill preparations were started on time barely a week before the commencement that is immediately after the semester two exams. In the first three weeks, it was quite easier with some guide and corrections from the supervisor.

Nevertheless, a compatibility issue was encountered between the *SafeMath* library versions and a number of vulnerable smart contracts. As a result, some of the constructs and syntaxes must be modified before they conform to work together.

While testing the implemented features to some of the real-world vulnerable smart contracts, negative values were experienced even though the results were positive using the Demo smart contract. The literature was then checked where it was learnt that the scheme could be suitable when only the vulnerable components were considered. These arguments were highlighted in chapter 3 section 3.2 and chapter 6 section 6.3.

After the demonstrations of the research work with the examiners and upon receipt of some feedbacks on the structure and the presentation of the work, missing and non-completeness of some of the important key information was observed. A one contingency plan left has helped greatly in order to effect corrections and fix issues on the write-up.

Chapter 3: Review of Related Work

3.1 Introduction

The birth of Blockchain has given rise to numerous opportunities across the financial industry, supply chains, and the Internet of Things (IoT). Although, Bitcoin [16] is the first and the most popular blockchain since its inception in 2009 [2], [17], Ethereum [12] appeared to be upsurging in popularity in the last couple of years due to its promising technologies [4] by introducing smart contracts, eliminating third parties, allowing the development of Decentralised Applications (Dapps). However, it is faced by the number of security issues mainly due to the crude nature of the programming languages (typically Solidity) use to build smart contracts that run on top of Ethereum.

As outlined by [1], [4], [18]–[20], it is important to deal with security issues of Ethereum smart contracts because money is often lost as a result of the existential threats therein. A number of attempts to deal with the Ethereum smart contracts security vulnerabilities emerged. Overall, there are three main techniques in handling the smart contracts' vulnerabilities *viz-a-viz:*

- formal verification,
- symbolic execution, and
- taint analysis.

This chapter discusses how those techniques shaped the way in which smart contract vulnerabilities are dealt with whilst focusing on the techniques employed to resolve the arithmetic vulnerabilities are briefly given. The strengths and weaknesses of these techniques were highlighted as well.

3.2 Approaches to Smart Contract Vulnerabilities

3.2.1 Formal Verification

A formal verification scheme is an approach in which the system's specification is compared with the actual implementation to ensure completeness of such specification versus the implementation. It uses a mathematical proof concept.

According to [17], the intuition of developers when coding smart contract programs often leads to vulnerable contract deployment because of some mismatch between the semantics considered during design and development and the actual semantics. Therefore, [17]

developed a framework called *FsolidM* for considering fresh and secure smart contract development. Although the plugin allows safer writing of smart contracts, it incurs more gas costs to the transaction and execution of the smart contracts. They have considered a number of vulnerabilities without integer overflow despite its huge number reported on the CVE database.

On the other hand, [21], developed a similar formal verification tool to *FSolidM* but explicitly for the smart contract name service which is an Ethereum decentralised name registry. His techniques involve manual analysis of the code and are therefore prone to errors. Similarly, [22], emphasises on the safety and correctness of the smart contracts through conversion of the Solidity code and the Ethereum Virtual Machine (EVM) to F* programming language code. The EVM is responsible for the conversion of smart contract code written in high-level language like the Solidity to the Ethereum bytecode. The main essence is to analyse vulnerable patterns. However, the F* translation does not support all features of Solidity which means it is unlikely for the framework to indicates all the corresponding vulnerable patterns present on the smart contracts.

3.2.2 Symbolic Execution

The symbolic execution or evaluation scheme determines sets of input variables or parameters that make a program run in a given way thereby giving some clue to the developers to find out if the contract is vulnerable [2]. The very first attempt to study smart contracts based on symbolic execution is the work of [2] who developed a tool called *Oyente* [23]. Virtually majority of subsequent works are based on that or with an extension of it. Notably, quite a number of works that followed *Oyente* are *Zeus* [24] by [5] and *Securify* [25] by [8]. Both *Oyente* and *Zeus* do handle the integer overflow, which is one of the classes of arithmetic vulnerabilities. The main demerit of *Oyente, Securify,* and *Zeus* is that they produce false positive (FP); meaning they state that vulnerability as absent even if it is present on the smart contracts leading to the exploitation of the deployed contracts without notice. Similarly, these tools are used to test for the existence of the vulnerability not to embed to the program for prevention, unlike the *SafeMath* library. But they are quite helpful before the deployment of a smart contract to the live blockchain. *Securify* does not indicate explicit results for an overflow vulnerability. *Oyente* by far is better among these tools for overflow, as it flags out specific a section of the code that has overflow vulnerability.

However, [26], developed a novel approach called *Maian* that consider the lifetime symbolic analysis of smart contracts by applying reasoning techniques. They are a greedy contract (it remains alive sporadically), suicidal contract (can be halt by the owner while executing) and prodigal contract (returns funds to the owner). The work of [26] argued that earlier works consider a single call to smart contract and are therefore subject to produce a huge false positive. The *Maian* tool demonstrated high true positives. But, the underlying success of the smart contract is to develop a tool that allows very good preventive mechanism like the *SafeMath* library, because as stated in Chapter 1, once a smart contract is deployed it cannot be modified when a bug is fixed, it has to be redeployed again.

Generally, the symbolic execution method is an infeasible way of dealing with large scale smart contracts because it utilises all the paths or nodes that input goes to therefore a huge amount of resources is needed to conduct it of which is chargeable in Ethereum smart contract applications or programs.

3.2.3 Taint Analysis

This is a novel approach of ensuring that a smart contract program is checked to ensure that the variable is not altered by users' input maliciously. Contrary to symbolic execution techniques in 3.2.2, quite a few authors focus on these methods.

A novel approach of [27], developed a tool called an *EasyFlow* [28] which underpins the *SafeMath* library mechanism and the taint analysis to trigger arithmetic vulnerabilities. They criticized the SafeMath to produce false-positive results in preventing arithmetic vulnerabilities. However, *EasyFlow* is the detective tool at the code level. It is not a preventive tool like the *SafeMath* that that can be invoked at run time. Nevertheless, the approach looks unique but the given website [28] to use the tool is no longer available.

3.2.4 Static Analysis

In fact, virtually developers utilise this approach in some ways for designing, development, and testing of smart contracts programs. A handy work of [15] applies the Extensible Mark-up Language (XML) based static analysis paradigms to flag vulnerable patterns for smart contracts programs. The tool offers some recommendation for the best practices and show the severity of those flagged patterns. Unlikely, it flags the use of *SafeMath* as a potential issue that could lead to a known out of the gas vulnerability.

3.3 Arithmetic Vulnerabilities

Virtually, the approaches in 3.2 are generic to a number of smart contract security vulnerabilities aside arithmetic ones which are the specific focus of this dissertation project. Unfortunately, the EVM has no standard checkmating mechanisms across libraries for the arithmetic overflow [29]. For this reason, the *SafeMath* library developed by *OpenZeppelin* remains one of the commonly used mechanisms to mitigate or protect them.

An integer overflow and underflow are mainly exploited on the addition or subtraction operations respectively. For instance, an overflow is triggered through repeated addition until a value is reached beyond the range allocated to it. That is dangerous because it could lead to users stealing ether on the network.

The *SafeMath* is recommended by [6], [15], [27] to be a de-facto standard for getting rid of unsafe arithmetic operations. Unlike the *EasyFlow* tool that does detection, the *SafeMath* is a preventive tool as shown in Figures 3.1 and 3.2. From Figure 3.1; first-line one is the normal expression that is vulnerable to overflow due to the use of unprotected (+) operation by most compilers. The second line however uses 'add' operation defined on the *SafeMath* library shown in Figure 3.2 in order to mitigate the overflow vulnerability.

The potential overflows are halted using one of the popular guards functions called assert(), require(), and throw() depending on the Solidity compiler version as shown in Figures 3.1 and 3.2.

```
total =x+y;  //normal operations
total =x.add(y); // using the add function from the SafeMath //library
```

Figure 3. 1: Function call from SafeMath

```
function add(uint256 a, uint256 b) internal pure returns (uint256 ) {
uint256 c = a + b;
require(c >= a, "SafeMath: addition overflow");
return c;
}
```

Figure 3. 2: Typical block of function to prevent addition

Moreover, the *SafeMath* library is proven to be the an option to avoid integer overflow or underflow without necessarily modifying the EVM and/or the Solidity compiler [6]. This solution is known to be in the application layer. However, [6] argue against the correctness of *SafeMath*, where and showed that it produces false positives compared to their tool called *Orisis*. The trigger of the vulnerability by the *SafeMath* is not always true. *Orisis* was developed based on symbolic execution and the taint analysis. Like the earlier approaches outlined in

section 3.2, *Orisis* is a detective tool focusing mainly on arithmetic overflow. Although, they criticised the *SafeMath* library in yielding false-positive protection, *Orisis* also even though it is reported to perform better and more correct detection of overflow than the existing tools like *Zeus*, it is not always correct too.

3.4 Gas Cost Analysis

In order to counter the gas overheads for smart contracts programs, a number of novel approaches were developed. The work of [7], [30], [31] are hereby discussed.

Overall prior works concentrated on developing tools to trace the gas-focused vulnerabilities. For instance, [31] came up with a symbolic execution tool called *GASPER* that discovers costly pattern on smart contracts. It used the idea of taint analysis like [27] as stated in 3.2.3 above. Going further, [31] proved that under optimised smart contracts cost more gas than the optimised one that were designed with consideration to gas costly coding in order to avoid them. In the course of their research, identified seven gas costly patterns.

One of the typical patterns is the use of expensive operations on the loops which run continuously could consume more gas. The *GASPER* tool is only useful in analysing gas costly patterns present. On the other, [30] analysed expensive running smart contracts with a visual representation of the gas cost analysis to aid understanding of the interpretation of such analysis. This is useful for clarity because even in the work of [8], it is argued that it is hard to understand and interpret the results of most the security analysis tools for smart contracts. Similarly, [7] developed a tool called *Madmax* utilising static analysis and symbolic execution to flag-out smart contracts that run out-of-the gas so as to avoid losing out of funds.

In a nutshell, unlike *GASPER* and *Madmax* that do the flagging out the costly patterns one a smart contracts and gas-focused analysis on collections of smart contracts. It is worth noting that, there have not been adequate efforts to reduce gas cost practically. This work is based on optimising one the library that causes such costs called *SafeMath* as earlier stated

3.5 Research Gaps

While prior literature concentrated at analysis and verification of written and deployed smart contracts as depicted summarily in Table 3.1 below, the focus of this work is targeting the improvement of the design stage by leveraging on the identified arithmetic bug. This will be achieved by improving the *SafeMath* library which prevents the arithmetic overflow or underflows automatically, with the ultimate goal of identifying a trade-off between performance and security.

Further, in Table 3.1, there are four key features used to summarise some of the key information from the literature. One of which is the main purpose of which the tool is developed. Overall, they are detective tools except the SafeMath which is a preventive tool. Secondly, speed of use; among the tools used, virtually is seamless, meaning that they are fast to use except *Mythril*. Thirdly, the number of tools are many, therefore those not able to be analysed are called Not Applicable (NA). It is worthy to note also that, some of these tools are not available on the Web. Lastly, the approaches associated are highlighted namely; Formal Verification (FV), Symbolic Execution (SE), Static Analysis (SA), and Gas Cost Analysis (GCA).

Table 3. 1: Smart Contract Vulnerability Tools

Feature	**Tools**										
	FSolidM	*Oyente*	*Mythril*	*Zeus*	*Securify*	*SmartCheck*	*Maian*	*EasyFlow*	*Gasper*	*MadMax*	*SafeMath*
Purpose	Design	Detective	Detective	Detective	Detective	Detective	Detective	Detective	Detective	Detective	Preventive
Speed	NA	Seamless	Slow	NA	Seamless	Seamless	NA	NA	NA	NA	Seamless
Correctness	NA	FP	FP	NA	Not explicit	Not explicit	NA	NA	NA	NA	Effective
Approach	FV	SE	SE	SE	SE	SA	SE	TA	GCA	GCA	NA

Chapter 4: Methodology

4.1 Introduction

This chapter discusses the series of steps followed and the descriptions of tools used to achieve the set objectives. It begins by illustrating the workflow of the main tool used called the Remix compiler and shows its weaknesses to deal with the arithmetic vulnerabilities. The chapter went ahead to demonstrate the effectiveness of the *SafeMath* library to arithmetic issues. The underlying costs incurred by the library have been showed thereby showing the proposed improvement of the library. Finally, both the libraries were tested with some tools for security vulnerabilities namely *Mythril*, *Oyente*, *Securify* and *SmartCheck* and some cases of integer overflow with attack vectors showcased too.

4.2 Remix Compiler

Essentially, the Remix compiler [7] was employed for most of the analysis and testing of the smart contracts and associated libraries (both original and the modified *SafeMath).* It is a renowned compiler for simplifying creating, analysing and testing of Ethereum smart contracts written in solidity programming language. The EVM that translates a high-level Solidity code to bytecode is already bundled to it. It has all the needed platforms to test the functionality of the *SafeMath* library. It is a browser-based Integrated Development Environment (IDE) with both the live and local options. Particularly, the live option was used throughout this work. The latter option allows the use of the compiler even without the Internet connection.

There are two testing streams followed in the analysis of the *SafeMath* which are explained briefly as follows:

- Testing the default and the proposed *SafeMath* libraries with the Demo smart contract that has basic operations (addition, subtraction, multiplication, etc.) in it with some defined expression for overflow and underflow vulnerabilities.
- Going further, the libraries were tested with some real-world vulnerable contracts that capture real-world logic of smart contracts in handling financial transactions with some overflow vulnerabilities present.

The latest pragma version (^0.5.x) is maintained when testing the Demo smart contract. However, the existing real-world vulnerable smart contracts published on the CVE are using old pragma version (^0.4.x), of which importing the *SafeMath* library with latest header make

it incompatible to the functions defined in the library. Therefore, to maintain compliance, the 'pragma version' has to be downgraded to the lower version (^0.4.x). The term pragma is used to denote a header information that indicates the version declaration when writing Solidity smart contract programs, where 'x' stands for numeric value (0-9) denoting sub versions.

There are not many prerequisites needed to set up the Remix, what is required is the browser such as Chrome, Safari, etc. The compilation and running are quite straightforward as can be seen on the right panel of Compile and Run tabs in Figure 4.1 below. Remix also possesses a very good analytic tool for static analysis that prompts basic errors of the smart contracts and warning messages. Moreover, Remix is handy as it has 'auto compile' and 'autosave' features allowing automatic compilation and saving of work by itself. All the saved files are showed on the left pane called browser. By default, as a file is created and while code is being added, it is automatically saved. Similarly, even if the browser is closed and restarted again, the data is not lost. On the other hand, more than one file can be opened using tabs as can be seen in Figure 4.1 below, there are three opened files namely; *SafeMath*2.sol, Test.sol and *SafeMath*.sol.

The bottom-most pane on the status pane displays if the smart contract is executed correctly or not. It also shows the costs associated with it in terms of execution and transaction time. The former is the adverse cost due to the number and weight of the operations present on the smart contracts whereas the latter has to do with the cost of deployment of the smart contracts to the blockchain. The total sum of the two is referred to as the gas costs.

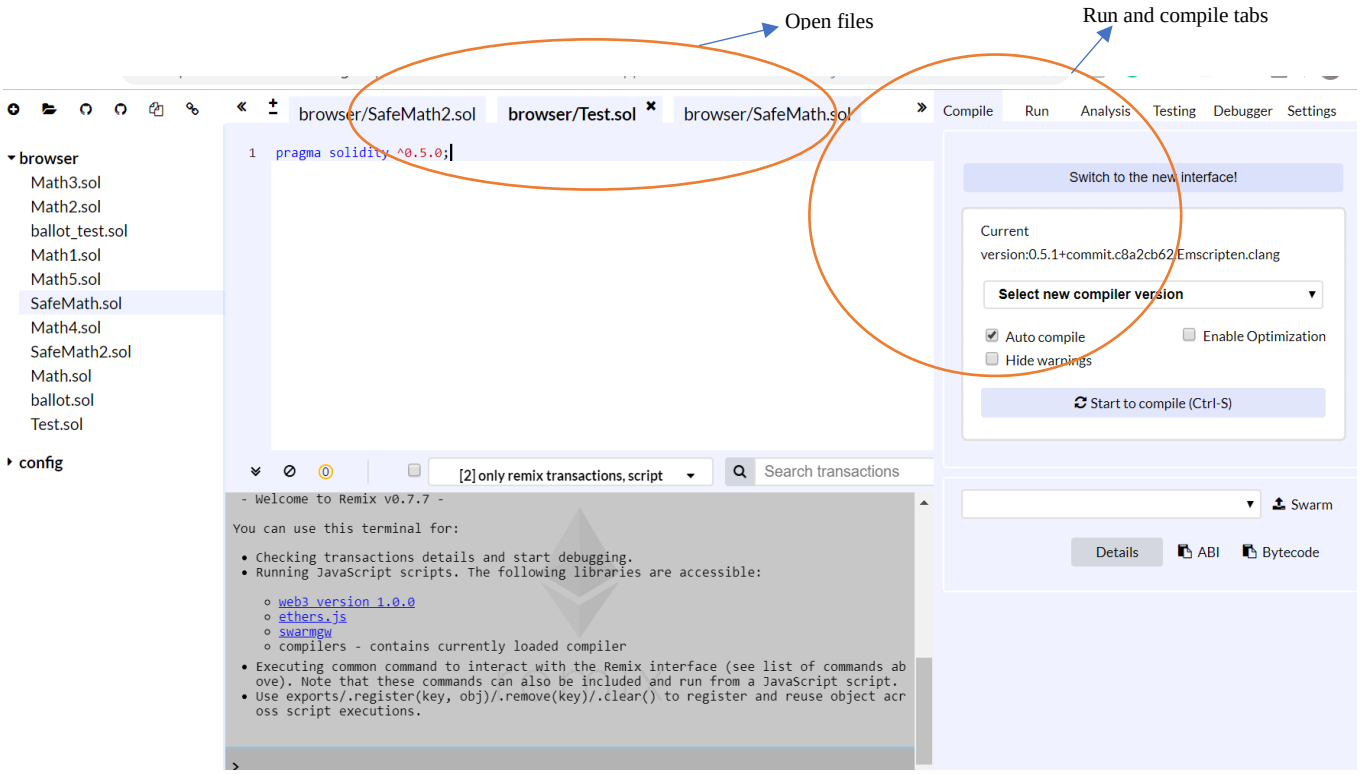

Figure 4. 1: Remix Compiler environment

Figure 4.2 shows typical costs when a smart contract runs on the Remix compiler. It is noteworthy that before the contract is deployed it gas limit should be specified closer to what is expected to be incurred due to the deployment of the contract. If the gas limit is less than or equal to the underlying gas costs (transaction cost and execution cost) then the contract is executed and run successfully, otherwise, it is halted abruptly [12].

Figure 4. 2: Execution cost and transaction cost from the Remix Compiler

The main essence of the Remix compiler is for testing purposes, other testing tools like the *Truffle* [32] and *Ganache* [33] allow similar experiments with the functionality and security requirements of the smart contracts prior to real deployment to the blockchain.

4.3 Smart Contracts Overflow Exploitation

SafeMath library is written using Solidity which is the most popular programming language for writing smart contracts. It is C and JavaScript-like language, making it popular for those with a similar background to exploit it quite easily.

However, it experiences a number of challenges. For example, it does not handle integer overflow. Nevertheless, it is the language chosen even though the Vyper promises better security, but it is still under experiments and by far most of the contracts are written in Solidity.

Below shows the description of the account from a vulnerable smart contract deployed called EthereumBlack. The EthereumBlack contract [34] is a kind of Initial Coin Offerings (ICO) that allows its developers to create a cryptocurrency that could be exchanged to other forms of currencies using some standards called tokens. The contract has a vulnerable function called mintToken which allows owners to send arbitrary value of token to any account of choice. The address, 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a, which has zero (0) balance when the balance is checked, as shown in Figure 4.3.

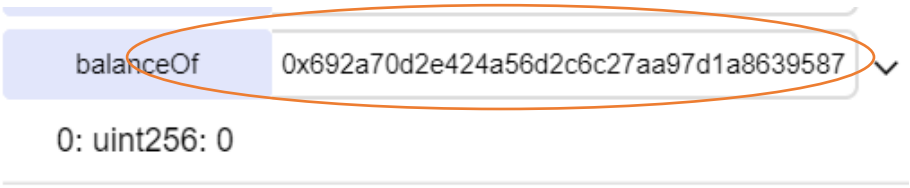

Figure 4. 3: Checking account balance of vulnerable smart contract

Now, when a negative value of mintedAmount is given as for example -1 (see Figure 4.4), the transaction is supposed to be terminated by the Remix compiler. As a result of this, an overflow occurred making the initial value hold by the mintedAmount as 0 to be the highest possible value *uint256* bit can hold, which is up to 2^256 – 1 as can be seen in Figure 4.5. This exploitation offers the account to have a very huge amount of value as a result of non-protection by the compiler.

Figure 4. 4: The transaction leading to overflow

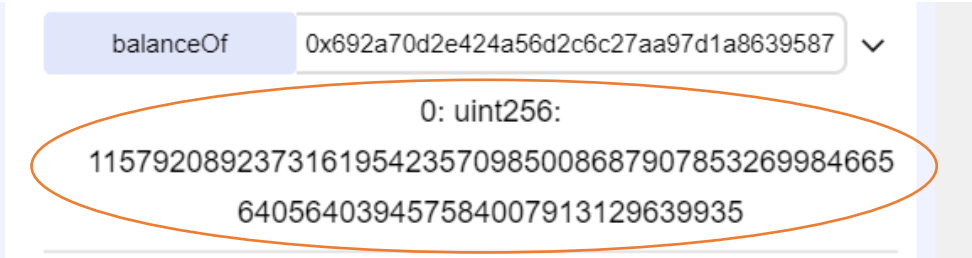

Figure 4. 5: An integer overflow

Nonetheless, the Remix compiler has been experiencing mapping issues to the EVM with respect to the integer data type sizes. For example, even though there are lower byte types of *uint8, uint16* and so on provided by the Solidity, such do not exist on the EVM [6]. As a result, current practice by developers uses either *uint8, uint16, uint256* etcetera throughout the code. Otherwise a mixed of smaller bit size may be recommended under some circumstances to avoid the occurrence of a huge overflow if it happens.

4.4 Analysis of the *SafeMath*

The *SafeMath* library captured five operations namely— addition, subtraction, multiplication, division, and modulus. Each of these is encapsulated in such a way that, if the running computation will lead to either underflow or overflow, it will be reverted.

Figure 4.6 depicts the overall architecture of the work. The processes designed were based on the seven phases of the Penetration Testing Execution Standard (PTES) [35]. The PTES is proven to be more effective compared to other similar frameworks in yielding less false-positives results by [36]. Out of seven PTES phases, three are applied to this work they are; threat modelling, vulnerability analysis and exploitation.

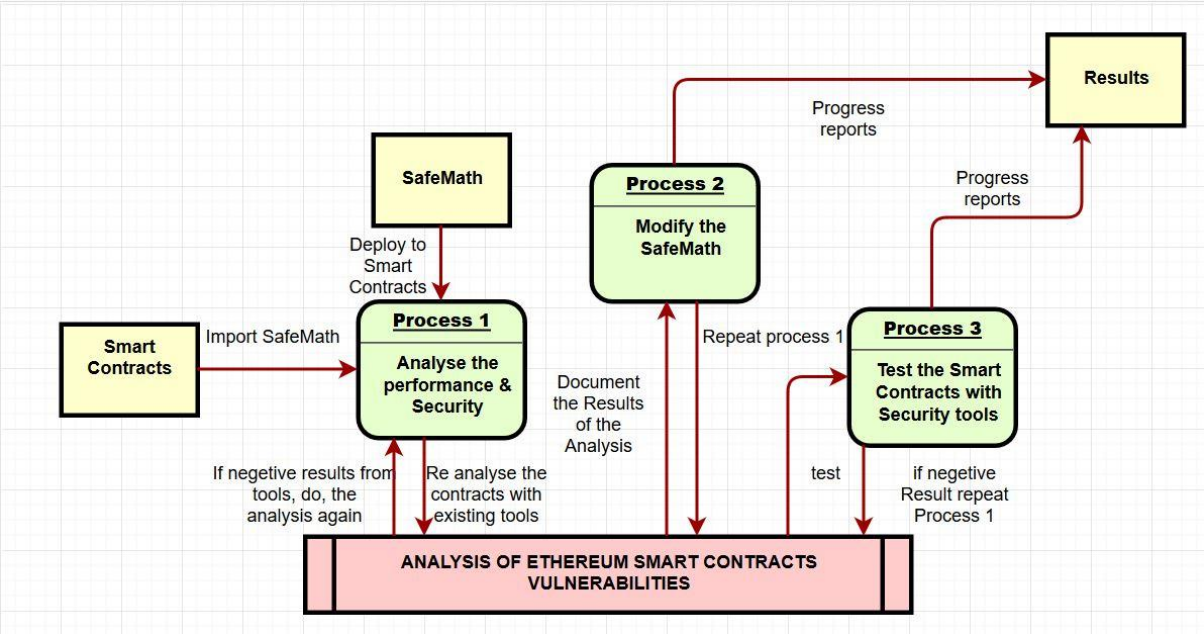

Figure 4. 6: Analysis of the *SafeMath*

At process 1, the original *SafeMath* library is tested and analysed for security and gas costs with some smart contracts' programs. Then, the library is modified at process 2 and the same test and analysis is undertaken again. Overall, both the original and the modified libraries are tested for vulnerabilities in process 3. Generally, both analysis and tests are recurrently observed a certain number of times. Both the results of the tests and analysis were recorded under Results and later discussed and evaluated under relevant chapters.

4.4.1 Design Prototype

After the original *SafeMath* functionality is been analysed, it was recognised to have five functions that were modified into two functions to cut off the gas costs. Both the original and modified *SafeMath* libraries were used to test the overflow vulnerabilities with a Demo smart contract as well as real-world smart contracts. After that smart contract vulnerability detective tools were utilised to analyse the effectiveness of the libraries. Lastly some sample of attacks for overflow were undertaken in three flavours:

- Without any library protection
- With the original *SafeMath* library
- With the modified *SafeMath* library

The steps undertook were fully explained in detail in bullet points (a, b, and c) below.

a. Merging functions together to reduce gas costs

Essentially, based on empirical analysis and in the recognition of the ideal principles that could be used to optimised gas costs offered by *GASPER* and *Madmax* as stated in Chapter 3, Review of Related Work, the number of functions in the library is responsible for adding overheads of the gas costs to the contracts when they are imported. And the existing library focuses on the users' input only, a decision to focus on the users output rather is made, because that is the most dangerous stuff. Even if the inputs are sanitised, if the output could be exploited in some ways, then such protection is of no use.

The addition and multiplication operations have similar logic in common, in both operations, the output (c) is often greater than the inputs (a,b) as shown in Figure 4.7. The functions definitions of the original *SafeMath* library functions could be obtained from *OpenZeppelin's GitHub* repository page [11]. However, for the sake of comparison, the original snippet of code for addition (add) and multiplication (mul) operations that were merged are shown in Figures 4.7 and 4.8 respectively.

```
function add(uint256 a, uint256 b) internal constant returns (uint256) {
        uint256 c = a + b;
        require(c >= a);
    }
```

Figure 4. 7: The original function SafeMath library 'add' function [11]

```
function mul(uint256 a, uint256 b) internal constant returns (uint256) {
                if (a == 0) {
                return 0;
                }
           uint256 c = a * b;
           require(c / a == b);
           return c;
         }
```

Figure 4. 8: The original function SafeMath library 'mul' function [11]

```
//merging add and mul functions
function addmult(uint256 c,uint256 a, uint256 b) internal
constant {
    (c >= a && c >= b, "SafeMath: overflow encountered");
}
```

Figure 4. 9: Merging Addition and Multiplication Operations

On the other hand, the modulus, division, and subtraction operations have output (c) less than the corresponding inputs (a,b). In addition, the operand b must be greater than 0. Figure 4.8 (line 4) illustrates the bundling of the functions together.

```
// merging mod and div and sub functions
function moddivsub(uint256 c, uint256 a, uint256 b  )
internal constant {
    (c<=a && b > 0, "SafeMath: overflow encountered");
}
```

Figure 4. 10: Merging Modulus, Division and Subtraction Operations

b. Testing the implementation with the vulnerable Smart Contracts

One of the seven phases of the PTES is vulnerability analysis which was used to analyse the *SafeMath* library effectiveness. A Demo smart contract named Math was designed to have some basic operations stated in the *SafeMath* library. The modified *SafeMath* and the original SafeMath have been tested in order to see their effectiveness to mitigate arithmetic vulnerabilities. Moreover, going further the real vulnerable contracts indexed in the CVE[37] and Etherscan [38] databases were collected. Those real contracts are written with the principle of the ICO. The CVE databases that captured over 525 vulnerabilities, majority of those reported contracts (almost 394 of 525) have one function that is common, mintToken function, with addition overflow vulnerability that allows a contract to set any arbitrary value to an address as can be seen in Figure 4.11 PolyAi [39].

```
function mintToken(address target, uint256 mintedAmount)
onlyOwner {
    balanceOf[target] += mintedAmount;
    Transfer(0, owner, mintedAmount);
    Transfer(owner, target, mintedAmount);
    }
```

Figure 4. 11: A Vulnerable function causing Overflow Vulnerability on PolyAi smart contract

A typical use of the modified *SafeMath* library is given in Figure 4.13 below. The snippet of code is from the EthereumBlack smart contract with the full source code on [34] as explained in detail earlier in section 4.3. Unlike the original *SafeMath* library implementation that captures the expressions of the operation mitigated, the proposed modified *SafeMath* uses the operation that is vulnerable and then calls the multipurpose/merged function (addmult) afterward for overflow mitigation. The essence of doing this is sanitised the final output rather than inputs alone.

```
function mintToken(address target, uint256 mintedAmount) onlyOwner {
    uint256 temp=balanceOf[target]; // temp varible for modified SafeMath expression
    balanceOf[target] =temp + mintedAmount; // modified SafeMath expression
    balanceOf[target].addmult(balanceOf[target],mintedAmount);  //SafeMath8  function
    call
    uint256 temp  = totalSupply;
    totalSupply = temp + mintedAmount;
    totalSupply.addmult(totalSupply,mintedAmount);
    Transfer(0, this, mintedAmount);
    Transfer(this, target, mintedAmount);
}
```

Figure 4. 12: The use the proposed *SafeMath* library to mitigate overflow vulnerability

c. Security checks for *SafeMath* and the Vulnerable contracts

After a while, the initial prototypes and the vulnerable contracts were tested with the existing security tools namely *SmartCheck* [40], *Securify*[25], *Mythril*[41] *and Oyente*[23]. Both *SmartCheck* and *Securify* are browser-based detective tools for smart contracts security vulnerabilities. In order to use them, a copy of smart contracts suspected is either pasted or uploaded. However, *Mythril* and *Oyente* are software-based testing tools that work on Docker Engine virtual machine [42]. The commands used to test these tools with the Docker are available on the *GitHub* link stated in Appendix A.

4.4.2 Attacks Vectors

An exploitation and threat modeling phases of the PTES was used to test for some given attacks that could lead an overflow vulnerability. The highest value allowable by the Ethereum for arithmetic using unsigned integer is 2^256-1 which is equal to 115792089237316195423570985008687907853269984665640564039457584007913129639935, which is 78 decimal digits. In order to test for overflow, the first operand was set to be the highest value and the second operand to be >=1 was issued. Any computation leading to the value outside the range defined, say, *uint256* which allows 0 to 2^256-1 will wrap around and

lead to dangerous overflow or underflow. Some experiments were made with the possible values that are above the range and tested the vulnerable contracts without the library, with the *SafeMath* library, and with the modified *SafeMath* library.

Summarily, the following values in Table 4.1 below.

Table 4. 1: Attack Vectors for Integer Overflow

Operation	Operand 1 (a)	Operand 2 (b)
Addition c=a+b	Highest value 2^256-1	Any positive value from 1>=1
Subtraction c=a-b	Any value a such that <b	To satisfy b>a
Multiplication c=a*b	Suitable value of a to make c>2^256-1	Suitable value of b to make c>2^256-1
Division c=a/b	Not applicable	b must always be bigger than 0
Modulus c=a*b	Not Applicable	b must never be equals to zero

Chapter 5: Results and Discussions

5.1 Introduction

This chapter discusses the details of interpretation of the results. The meanings of the figures shown as part of the outcomes of the dissertation research project presented therein are hereby analysed.

5.2 Results

This section compares the gas costs by the original and the modified *SafeMath* libraries. It also shows the effectiveness of security detective tools in detecting the presence of arithmetic vulnerabilities.

5.2.1 Transaction costs and Execution cost

Table 5.1 shows the percentage improvement of the deployment of the Demo smart contract for the gas costs in terms of execution cost and the transaction cost by 16.50% and 12.17% respectively. Similarly, Table 5.1 shows the improvement of the execution cost and the transaction by 3.24% and 3.18% respectively when the modified *SafeMath* is applied to the EthereumBlack at the deployment stage.

Table 5. 1: The Computation of the gas cost improvement by the modified *SafeMath* for the Demo smart contract deployment to the Blockchain.

	Without *SafeMath*	**With the original *SafeMath***	**With the modified *SafeMath***	
Execution Cost (gas unit)	111,555	166,204	147,792	
Deficits		54,649	36,237	**Improvement**
Deficits (%)		48.99	32.48	**16.50**
Transaction Cost (gas unit)	200,123	272,504	248,156	
Deficits		72,381	48,033	
Deficits (%)		36.17	24.00	**12.17**

Table 5. 2*:* The Computation of the gas cost improvement by the modified *SafeMath* for the EthereumBlack smart contract deployment to the Blockchain

	Without *SafeMath*	**With the original *SafeMath***	**With the modified *SafeMath***	
Execution Cost (gas unit)	1,639,623	1,834,439	1,781,383	
Deficits		194,816	141,760	**Improvement**
Deficits (%)		11.88	8.65	**3.24**
Transaction Cost (gas unit)	2,229,891	2,489,719	2,418,899	
Deficits		259,828	189,008	
Deficits (%)		11.65	8.48	**3.18**

Figures 5.1 and 5.2 show the effect of the gas limit when deploying an EthereumBlack smart contract with the original *SafeMath* and the modified *SafeMath* library respectively. It could be observed that the 'gas limit' was set up to be the same worth 2,460,000 gas in each case. It has been earlier outlined in Table 5.2 above that the actual gas requirements for the deployment of the EthereumBlack smart contract to the blockchain with the original *SafeMath* and the modified *SafeMath* for the transaction cost are 2,489,719 and 2,418,899 respectively. Hence the gas limit restricts the execution of the former contract whilst allowing the latter to execute successfully.

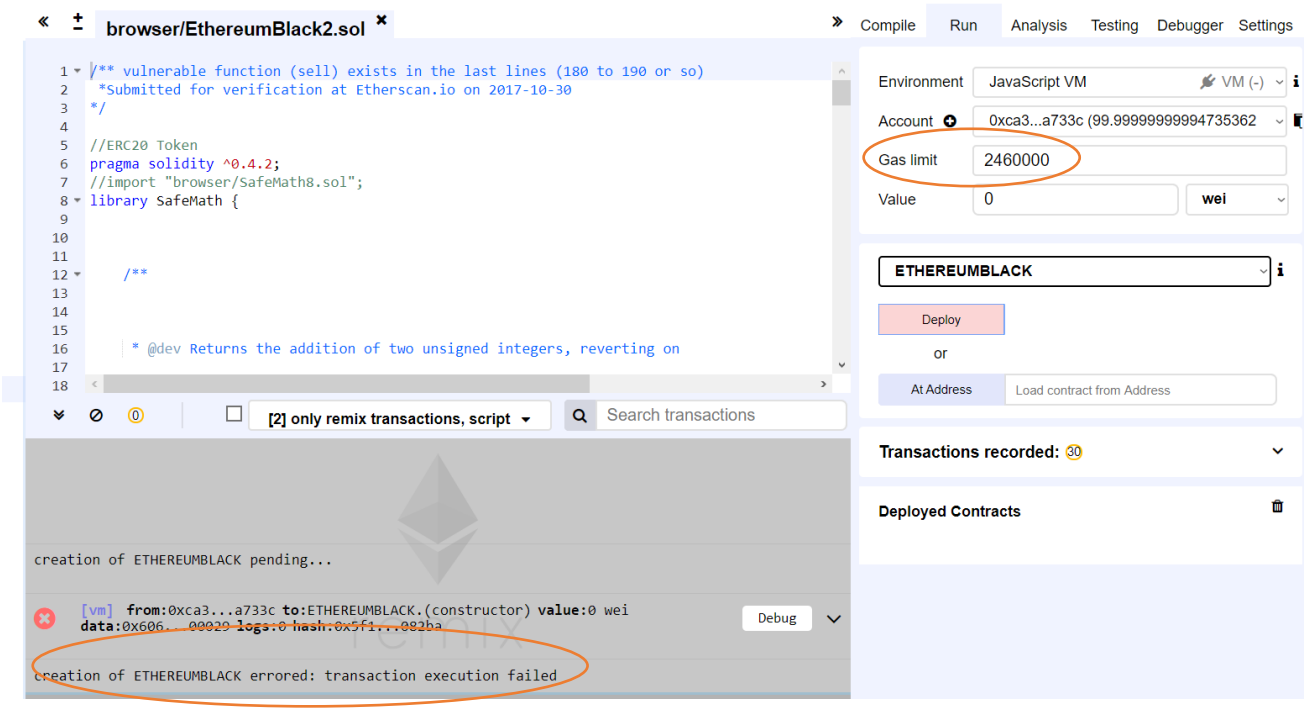

Figure 5. 1: The Implication of gas limit and the gas cost for the EthereumBlack smart contract with the original *SafeMath* library

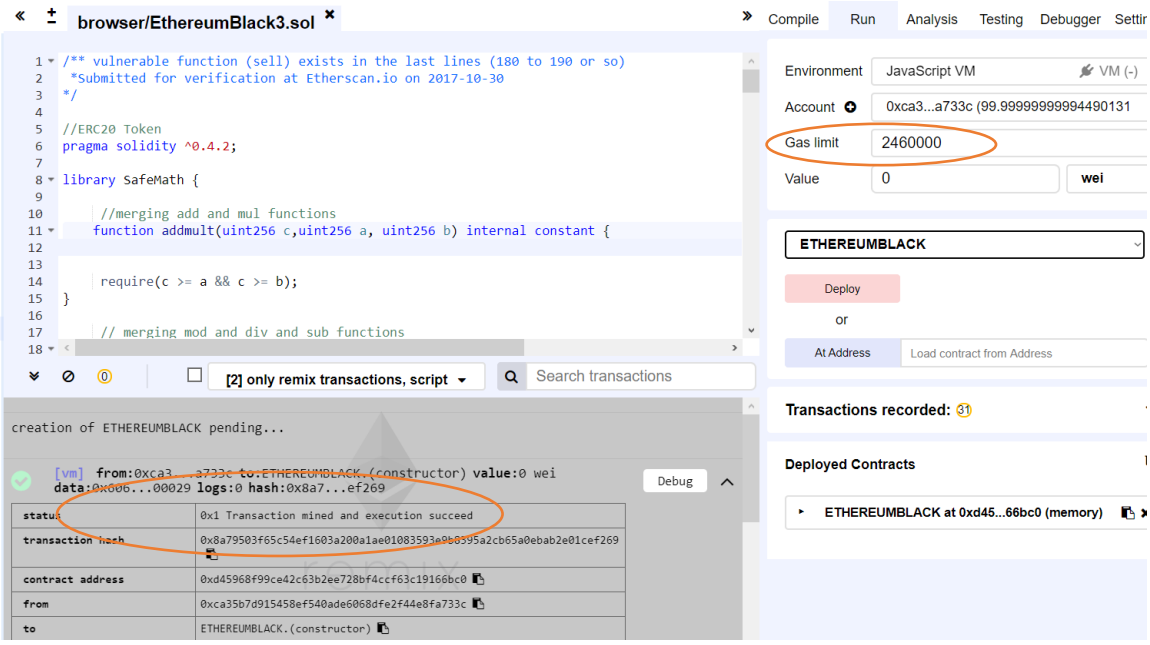

Figure 5. 2: The Implication of gas limit and the gas cost for the EthereumBlack smart contract with the modified *SafeMath* library

5.2.2 Testing the vulnerable Smart Contract (Demo and Real-World) with Security Tools

In order to test the implementation, four existing security vulnerability tools that check for overflow vulnerability were selected. However, those tools also check for other kinds of vulnerability aside arithmetic. It was done in three phases; without any supporting library, with the original *SafeMath* library, and with the modified *SafeMath* library.

Table 5.3 below illustrates how the vulnerable smart contracts under review were tested for arithmetic overflow with the current smart contracts vulnerabilities tools alongside the original *SafeMath* and modified version. The numbering schemes 1,2 and 3 are denoted for the original smart contract without any library, with the original *SafeMath* and with the modified *SafeMath* respectively. *Securify* and *SmartCheck* does not explicitly indicate any of the arithmetic flaws present. Whereas *Oyente* is effective in detecting the vulnerabilities present in the absence of the library, the vulnerabilities were not detected when the libraries are included in the code of the vulnerable smart contracts. A full description of how the security tools are used is stated in Appendix A.

Securify triggers the use of *SafeMath* library as another type of vulnerability called out-of-gas with high severity level due to the overhead incurred by the library that could lead to unavailability as earlier pointed out too. According to the *Securify* recommendation on the use of *SafeMath* is that only the vulnerable component should be applied with revert operations that *SafeMath* used to mitigate the occurrence of the vulnerability.

Table 5. 3: The Analysis of the tests of vulnerable smart contracts with the state-of-the-art security tools and the SafeMath libraries

Tool	Vulnerable Smart Contracts Without Library, with the original *SafeMath* and the modified *SafeMath*								
	Ethereumblack1	**Ethereumblack2**	**Ethereumblack3**	**Polyai1**	**Polyai2**	**Polyai3**	**Growtoken1**	**Growtoken2**	**Growtoken3**
Mythil	Present	Absent	Absent	Present	Absent	Absent	Present	Absent	Absent
Oyente	Present	Absent	Absent	Present	Absent	Absent	Present	Absent	Absent
SmartCheck	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit
Securify	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit	Not Explicit

5.2.3 Testing the vulnerable Smart Contract with a typical attack

Recall a sample exploitation to the EthereumBlack in Figures 4.4 and 4.5, let us examine the effect of the modified *SafeMath* library in mitigating the same scenario as can be seen in Figure 5.3 below. As can be observed, the transaction window shows that an overflow is been reverted from happening.

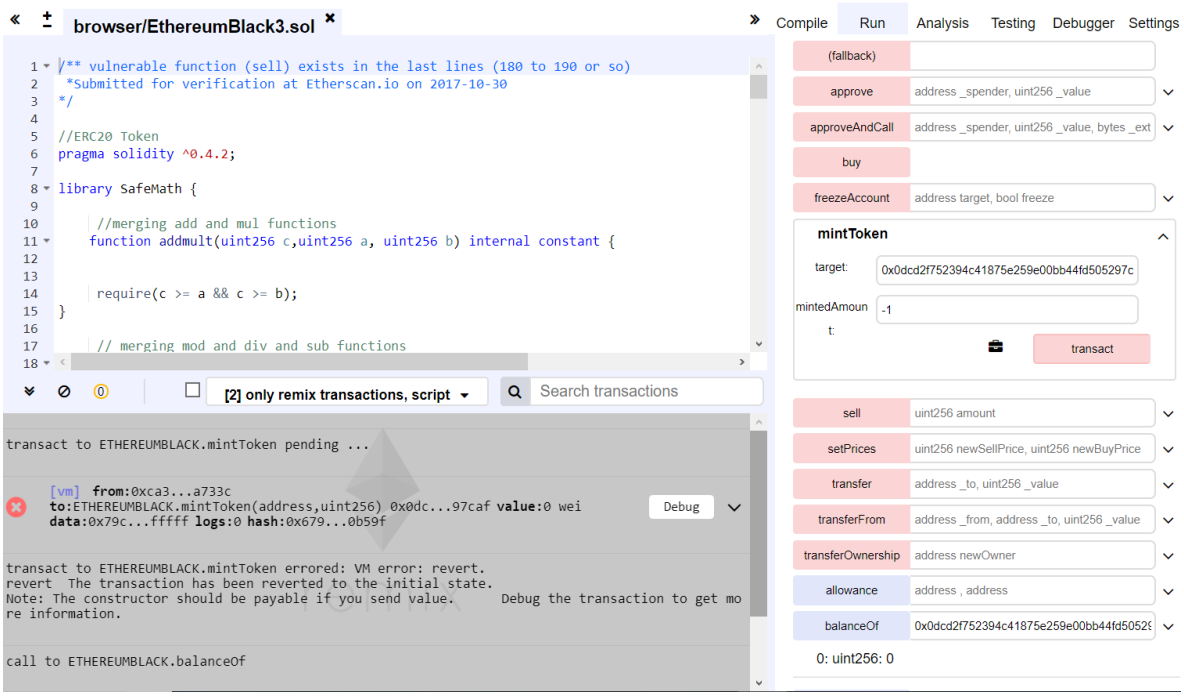

Figure 5. 3: Mitigating an overflow due to the modified *SafeMath* library protection

5.3 Evaluation

The modified *SafeMath* library is efficient over the existing *SafeMath* in terms of performance by reducing the gas cost incurred by 16.50% and 12.17% for execution cost and the transaction cost respectively with the Demo smart contracts (Table 5.1). However, with the one of the real-world vulnerable smart contracts called EthereumBlack as discussed already, the modified *SafeMath* lower the gas cost by 3.24% and 3.18 for the execution cost and the transaction cost as shown in Table 5.2. Both libraries offer the same level of security protection.

On the other hand, the security requirements happen to be the same. Both libraries were tested for overflow and are favourable against it. Overall, both the original the *SafeMath* and the modified *SafeMath* allows the revert of the overflow for negative values that could lead to subtraction overflow. Typically, 0 minus 1 sounds illogical for real business computations. As stated earlier the use of the *SafeMath* library halt the occurrence of the overflow vulnerability,

the absence of the library could lead to illogical operation to occur yielding the big balance of 115792089237316195423570985008687907853269984665640564039457584007913129639 , which is the highest range of value of the *uint256*. Similarly, security tools such as *Oyente* show that the libraries protect overflow from happening.

5.3.1 Performance

It has been noted from the results that there is less gas costs using the modified *SafeMath* library compare to the original *SafeMath* library. This happens mainly due to the reduction in the number of function definitions and function calls. The higher the number of functions definitions and calls, the more likely it is to cost more gas when the smart contract is executed.

5.3.2 Security

Although the security feature did remain the same between the original *SafeMath* and the modified *SafeMath*, there is a critical analysis of different possible security risks associated with both libraries. An efficient way for the use of the *SafeMath* function on vulnerable function was tested to be effective.

5.4 Discussions

Whereas prior work emphasis on the detection of arithmetic vulnerability and the *SafeMath* library that helps in prevention produces high gas cost, the modified *SafeMath* introduced in this dissertation projects prevent the vulnerabilities of the smart contracts from arithmetic.

For the fact, the optimised has improved the *SafeMath* by lowering the gas cost, which is crucial in getting rid-off out of the gas vulnerabilities that tend to cause in availability as shown in Tables 5.1 and 5.2. It could be seen in Table 5.3 that security tools like *Oyente* could be useful to be used together with the modified *SafeMath* because *Oyente* is proven to flag out the specific portion of the code that is vulnerable for overflow during the code testing. So, there is no need to apply the *SafeMath* protection wholly all over the as recommended by the current *SafeMath* prevention practices codes where it might not exist

The implication of the out-of-gas vulnerability has been shown in Figures 5.1 and 5.2 where the same EthereumBlack smart contract was applied with the original *SafeMath* and the modified *SafeMath* libraries. The same security protection is been tested in each implementation and it showed the same strengths in each. The same ‘gas limit’ was set up before the deployment of the contract to the blockchain in each case. Due to the gas

requirements of the contract with the original *SafeMath* exceeding the gas limit, the execution of the contract fails. The consequence of this is to do with one of the important security requirements called availability which has been highlighted in earlier chapters.

As table 5.3 further demonstrates, the four security tools used in detecting the presence of arithmetic vulnerability on the smart contracts in three situations namely without the library, with the original *SafeMath* library and with the modified *SafeMath* library show the efficacy of the use of *Oyente* and *Mythril* over *SmartCheck* and Securify in flagging-out the specific arithmetic flaws present. However, the penetration test shows non-correctness of these tools sometimes where vulnerability was exploited to be absent and it flag-out that it is present. This is known as false-positives detection. The implication is that there adapted PTEST methodology alongside the use of efficient mechanism in the use of the *SafeMath* and the use of detective tools will help in improving the correctness of the defence against an overflow vulnerability

Whereas, both the original and the modified *SafeMath* libraries offer the same security protection, new efficient mechanism for protecting overflow. It is important in two flavours. One it reduces the gas cost; two it reduces the complexity of applying the *SafeMath* library functions by developers throughout the code. Figure 5.3 shows a typical protective mechanism rendered by the proposed *SafeMath* library as against the way similar exploitation was highlighted without such protection in Figures 4.4 and 4.5.

5.5 Contributions to Knowledge

- It has been proved that the use of *SafeMath* wholly on all operations could lead to more gas costs without contributing to the security requirements. A new scheme is therefore proven to be effective with the use the state-of-the-art security testing tools in order to flag the real vulnerable components and then apply the protection with the *SafeMath*. Although the security protection of both the original and the proposed *SafeMath* tested to be the same, the viability of smart contract security testing tools has been empirically validated and compared which shows the most effective ones in flagging out the arithmetic vulnerabilities.
- An enhanced and optimised *SafeMath* library was proposed which has bundled similar logics of similar functions for performance. Instead of five functions of the original *SafeMath*, the proposed *SafeMath* now has a light version with merged functions to only two.

Chapter 6: Conclusion

6.1 Introduction

This chapter summarises major findings achieved and states the limitations as well the future work.

6.2 Conclusion

Despite the number of approaches to analyse the security vulnerabilities on the Ethereum smart contracts as outlined in Chapter Three, Review of Related Work, there has not been adequate work on arithmetic vulnerabilities prevention. The current *SafeMath* library mitigates the arithmetic flaws in some ways, thereby incurring some gas deficits. This dissertation attempt to analyse the strategy in lowering the cost at the same time maintaining the security of the library.

It has been observed that getting rid-off overflows from contract programs wholly is a very difficult task as explained in Chapter Five, Results and Discussions. However, the *SafeMath* library has been effective in dealing with the arithmetic overflow if it is used very well in all the vulnerable components. Similarly, the original *SafeMath* library focuses on the analysis of the user's inputs whereas the modified versions focus on the output. The notion was, even if the inputs are sanitised, if the output leads to overflow, then such protection is not helpful. Although the empirical tests show that the modified *SafeMath* is able to maintain the current security functionality of the current *SafeMath*.

Moreover, the applications of the original *SafeMath* library functions to all the components abruptly lead to more gas costs as a result. Also, the number of functions defined on the *SafeMath* library as well as how often these functions are called has a greater effect in doubling the associated gas costs. Therefore, the main outcome of this work is the integration of relevant and related functions into one is useful introducing the overhead as shown in Chapter Four, Methodology. Secondly, an efficient way of using the *SafeMath* by concentrating on the vulnerable components is therefore proposed. The proposed *SafeMath* reduces the execution cost by 3.24% and 3.18% to the two categories of gas costs namely; the transaction cost and the execution cost respectively when tested with a real called vulnerable ICO smart contract known as EthereumBlack.

Furthermore, the work re-enforces the use of the smart contract security testing tools such as *Oyente* and *Mythril* at the same time illustrate the viability and effectiveness of some

of them. After the analysis of the tools, one of their weaknesses tends to be yielding false-positive results to the smart contracts tested to be free of overflow vulnerabilities by *SafeMath* prevention. Similarly, some tools do not state explicitly whether an overflow exists. On the other hand, some of the published work does not have the tools available on the Web whereas some of the given tools are not reachable at the time of this research, for example, an *EasyFlow* [28].

6.3 Recommendations

In order to utilise the proposed *SafeMath* library, developers need to have experience on the vulnerable components; unlike the existing scheme is to apply the *SafeMath* library functions to all the arithmetic operations which will consume more gas. The modified *SafeMath* in addition to focusing on the overall output of the operations, it focuses on the vulnerable components.

The existing static analysis and symbolic execution tools could aid the developers to examine those vulnerable components. Similarly, it is often the best idea to do a manual audit and unit testing in order to make sure about any security tools as occasionally the tools could result in false-positives.

6.4 Limitations

Preventing an integer overflow is one of the classical issues in computing, it is very difficult to avoid it totally. Based on the analysis of the implementation offered by the *SafeMath* library on the reported vulnerable smart contracts, it has been shown that there are other function calls that fetch data other than arithmetic that may lead to overflow. Figure 6.1 shows the results of the *Oyente* analysis of fully protected smart contracts of which all arithmetic ones are not reported except one call which is not within the scope of this work. The code associated with this vulnerability is reported in Figure 6.2 where it could be observed that there are no arithmetic operations.

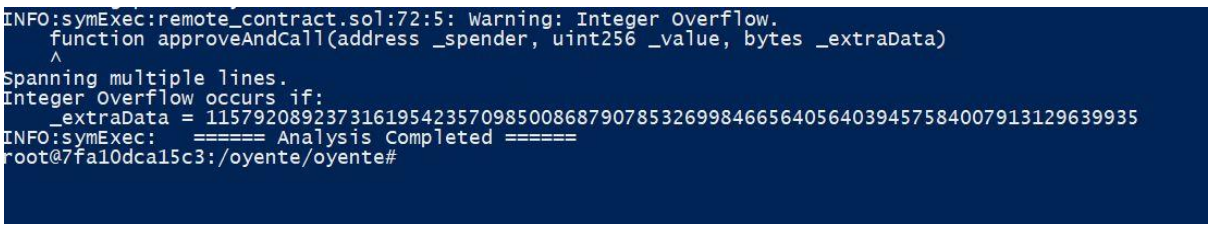

Figure 6. 1: *Oyente* Overflow Analysis Flagging _xtradata Variable

```
function approveAndCall(address _spender, uint256 _value,bytes _extraData)
returns (bool success) {
tokenRecipient spender = tokenRecipient(_spender);
if (approve(_spender, _value)) {
spender.receiveApproval(msg.sender, _value, this, _extraData);
return true;
}
}
```

Figure 6. 2: Non-arithmetic vulnerability leading to overflow issue

6.5 Future Work

The implication of the *SafeMath* library optimisation is in two ways namely improving the effectiveness of the security protection as well as lowering the gas costs. In order to ensure the credibility of the more real-world smart contracts need to be analysed.

6.5.1 Adding more security protection by adding more operations

The earlier intention of adding some arithmetic operators called the ADDMOD and MULMOD which are often considered as cryptographic operations are beyond the scope of this work, the future work is to incorporate them into the modified *SafeMath* library.

6.5.2 Lowering more gas cost through the EVM analysis

It has been proven by [7] that, to understand the costly pattern on the Smart contracts programs is to exploit the machine code which is not covered in this dissertation. The EVM approach has to do with the analysis of the machine code whereas this work focuses on the investigation and analysis of the high-level code as it is been compiled by the Remix Solidity Compiler. The future work is to dwell on the EVM analysis of the gas pattern of the *SafeMath* library functions in order to have the idea of the most expensive calls and flows in order to reduce more gas costs.

6.5.3 More Analysis

Moreover, the limited time does not allow adequate analysis to the published vulnerable smart contracts on the CVE alongside many vulnerabilities highlighted. This should another future direction.

References

[1] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on Ethereum smart contracts,” 2016.

[2] L. Luu, “Making Smart Contracts Smarter,” in *CCS ’16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[3] N. Szabo, “The Idea of Smart Contracts,” 1997. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html. [Accessed: 03-Jun-2019].

[4] D. Macrinici, C. Cartofeanu, and S. Gao, “Telematics and Informatics Smart contract applications within blockchain technology : A systematic mapping study,” *Telemat. Informatics*, vol. 35, no. 8, pp. 2337–2354, 2018.

[5] S. Kalra, “Z EUS : Analyzing Safety of Smart Contracts,” *Netw. Distrib. Syst. Secur. Symp. 2018*, no. February, 2018.

[6] C. F. Torres and J. Schütte, “Osiris : Hunting for Integer Bugs in Ethereum Smart Contracts,” in *ACSAC ’18 Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[7] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, “MadMax : Surviving Out-of-Gas Conditions in Ethereum Smart Contracts,” *Proc. ACM Program. Lang.*, vol. 2, no. November, 2018.

[8] P. Tsankov, A. Dan, D. Drachsler-cohen, A. Gervais, F. Bünzli, and M. Vechev, “Securify : Practical Security Analysis of Smart Contracts,” in *CCS ’18 Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, no. June, pp. 67–82.

[9] S. Voshmgir, “Tokenized Networks: What is a DAO?,” 2019. [Online]. Available: https://blockchainhub.net/dao-decentralized-autonomous-organization/. [Accessed: 10-Aug-2019].

[10] R. Agrawal, “DAO — Democratizing ownership of organization through smart contract on blockchain,” 2018. [Online]. Available: https://medium.com/@xragrawal/dao-democratizing-ownership-of-organization-

through-smart-contract-on-blockchain-19d080332591. [Accessed: 30-Jul-2019].

[11] OpenZeppelin, “The standard for secure blockchain applications.” [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol. [Accessed: 10-Jun-2019].

[12] G. W. Founder and E. Gavin, “Ethereum: a secure decentralised generalised transaction ledger.” 2014.

[13] CVE, “Common Vocabularies Exposure,” 2019. [Online]. Available: https://cve.mitre.org. [Accessed: 20-Jul-2019].

[14] NCC Group, “Decentralized Application Security Project (or DASP) Top 10 of 2018,” 2018. [Online]. Available: https://dasp.co/#item-3. [Accessed: 10-Jul-2019].

[15] S. Tikhomirov, E. Voskresenskaya, and I. Ivanitskiy, “SmartCheck : Static Analysis of Ethereum Smart Contracts,” *2018 IEEE/ACM 1st Int. Work. Emerg. Trends Softw. Eng. Blockchain*, no. October 2017, pp. 9–16, 2018.

[16] S. Nakamoto, “Bitcoin : A Peer-to-Peer Electronic Cash System.” pp. 1–9, 2008.

[17] A. Mavridou and A. Laszka, “Designing Secure Ethereum Smart Contracts : A Finite State Machine Based Approach,” in *Twenty-Second International Conference on Financial Cryptography and Data Security 2018*, 2018.

[18] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, “Security Analysis Methods on Ethereum Smart Contract Vulnerabilities — A Survey,” pp. 1–21.

[19] S. Wang *et al.*, “Blockchain-Enabled Smart Contracts : Architecture , Applications , and Future Trends,” *IEEE Trans. Syst. Man, Cybern. Syst.*, vol. PP, pp. 1–12, 2019.

[20] Z. Zheng, S. Xie, and H. Dai, “Blockchain challenges and opportunities : a survey Blockchain challenges and opportunities : a survey Zibin Zheng and Shaoan Xie Hong-Ning Dai Xiangping Chen * Huaimin Wang,” *Int. J. Web Grid Serv. · January 2018*, vol. 14, no. January, pp. 352–375, 2018.

[21] Y. Hirai, “Formal Verification of Deed Contract in Ethereum Name Service,” The University of Tokyo, 2016.

[22] K. Bhargavan *et al.*, “Formal Verification of Smart Contracts Short Paper F *,” *Proc. 2016 ACM Work. Program. Lang. Anal. Secur.*, pp. 91–96, 2016.

[23] melonproject, “An Analysis Tool for Smart Contracts,” 2016. [Online]. Available: https://github.com/melonproject/oyente. [Accessed: 19-Jun-2019].

[24] NuoNetwork, “Zeus,” 2019. [Online]. Available: https://github.com/NuoNetwork/contracts-v2/tree/master/zeus-contracts.

[25] ICE center, “Security Scanner for Ethereum Smart Contracts,” 2018. [Online]. Available: https://securify.chainsecurity.com/.

[26] P. Saxena, “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale,” in *ACSAC ’18 Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[27] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, “E ASY F LOW : Keep Ethereum Away From Overflow,” in *ICSE ’19 Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, 2019, vol. 000.

[28] and Z. C. Jianbo Gao, Han Liu, Chao Liu, Qingshan Li, Zhi Guan, “EasyFlow Tool.” [Online]. Available: http://easyflow.cc/. [Accessed: 17-Jun-2019].

[29] D. Park, P. Daian, C. Tech, and G. Ros, “A Formal Verification Tool for Ethereum VM Bytecode,” in *The 2018 26th ACM Joint Meeting*, 2018, pp. 912–915.

[30] S. Christopher, “Gas Cost Analysis for Ethereum Smart Contracts,” ETH Z¨urich, 2018.

[31] T. Chen, X. Li, X. Luo, and X. Zhang, “Under-Optimized Smart Contracts Devour Your Money,” *2017 IEEE 24th Int. Conf. Softw. Anal. Evol. Reengineering*, 2016.

[32] Truffle Blockchain Group 2019, “Sweet Tools for Smart Contracts,” 2019. [Online]. Available: https://www.trufflesuite.com/. [Accessed: 10-Aug-2019].

[33] Truffle Suite, “Fast Ethereum RPC client for testing and development,” 2019. [Online]. Available: https://github.com/trufflesuite/ganache-cli. [Accessed: 05-Aug-2019].

[34] BlockChainsSecurity, “EthereumBlack,” 2018. [Online]. Available: https://github.com/BlockChainsSecurity/EtherTokens/blob/master/ETHEREUMBLACK/sell integer overflow.md.

[35] OWASP, “Penetration testing methodologies,” 2019. [Online]. Available:

https://www.owasp.org/index.php/Penetration_testing_methodologies. [Accessed: 17-Jun-2019].

[36] B. Liu, L. Shi, Z. Cai, and M. Li, “Software Vulnerability Discovery Techniques : A Survey,” *2012 Fourth Int. Conf. Multimed. Inf. Netw. Secur.*, pp. 152–156, 2012.

[37] CVE, “Ethereum Smart Contracts Vulnerabilities,” 2018. [Online]. Available: https://cve.mitre.org/cve/search_cve_list.html. [Accessed: 30-Jul-2019].

[38] YoBit.Net, “Ethereum Blockchain Explorer.” [Online]. Available: https://etherscan.io. [Accessed: 25-Jul-2019].

[39] N0pn0pn0p, “Smart Contract Vulnerability,” 2018. [Online]. Available: https://github.com/n0pn0pn0p/smart_contract_-vulnerability/blob/master/PolyAi.md.

[40] SmartDec, “SmartCheck,” 2018. [Online]. Available: https://tool.smartdec.net/newscan?type=upload. [Accessed: 27-Jun-2019].

[41] ConsenSys, “Mythril.” .

[42] D. Inc, “Docker Documentation,” 2019. [Online]. Available: https://docs.docker.com/. [Accessed: 20-Jul-2019].

Appendix A:

Source codes for the dissertation is available on *GitHub*: https://github.com/mansurmasama/summerproject/tree/master/codes. This is because the length of smart contracts analysed in three scenarios for each is quite lengthy involving hundreds of lines of codes for a given file.

Meanwhile, the commands used to utilise Mythril and *Oyente* could be obtained here

https://github.com/mansurmasama/summerproject/blob/master/codes/Oyente%20and%20Mythril%20Commands.

Whereas the *Oyente* and Mythril are command-based tools, *Securify* and SmartCheck are web-based. The websites are given in the Reference list.