A STATIC ANALYZER AND BUG PREDICTION ENGINE FOR SMART CONTRACTS

by

Ahmet GÜL

B.S., Computer Engineering, Ege University, 2016

M.S., Computer Engineering, Boğaziçi University, 2021

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2021

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor Prof. Dr. Alper Şen who taught me how to do academic research and how to access the information and fellow jury members Prof. Dr. Can Özturan and Assoc. Prof. Dr. Şerif Bahtiyar for their invaluable feedback.

I would like to thank Yavuz Köroğlu for sharing all his knowledge in all matters related to graduate education.

I would like to thank my parents Ertuğrul and Tülay Gül and my brother Atakan Gül for their support in my graduate education life as in every field of my life.

I would also like to thank Çağrı Dikmen and Tuğberk Tunçinan, who made my graduate education life not only in computer engineering but also in other fields, by discussing new ideas at every dinner.

Even though I cannot list their names one by one here, I would like to thank all my lecturers who have added value to me.

ABSTRACT

A STATIC ANALYZER AND BUG PREDICTION ENGINE FOR SMART CONTRACTS

As blockchain solutions are becoming increasingly common, identifying potential bugs in smart contracts written in Solidity language is vital for these solutions to work accurately. To precisely detect these bugs, developers must use several state-of-the-art bug detection tools and examine the potential bugs the tools report. In this thesis, we investigate common errors in smart contracts and developed a static analysis tool (SA-Solidity), which takes Solidity codes of smart contracts as input, to detect possible bugs. SA-Solidity converts Solidity source code into parse a tree and detects the erros by querying it. In addition, we demonstrate that one tool alone is not sufficient to detect all the bugs as SA-Solidity, and the known SmartCheck, and Securify tools identify different types of bugs in SmartEmbed's experimental set of smart contracts. Furthermore, we develop Machine Learning-based Bug Predictor for Solidity (MLBP-Solidity), which predicts files that would be reported by all the previously mentioned bug detection tools and facilitates the efforts of developers by allowing them to focus on a subset of files that are most probably buggy. Our experimental results show that MLBP-Solidity achieves 90-99% accuracy depending on the type of predicted bug.

ÖZET

AKILLI SÖZLEŞMELER İÇİN STATİK ANALİZ VE HATA TAHMİN ARACI

Blokzinciri çözümleri yaygınlaştıkça, Solidity dilinde yazılmış akıllı sözleşmelerdeki potansiyel hataların tespit edilmesi, bu çözümlerin doğru çalışabilmesi için hayati önem taşıyacaktır. Bu hataları düzgün bir şekilde tespit etmek için, geliştiricinin son teknoloji ürünü hata algılama araçlarını kullanması ve bu araçların rapor ettikleri olası hataları incelemesi gerekmektedir. Bu tezde akıllı sözleşmelerdeki olası yaygın hataları araştırdık ve olası hataları tespit edebilmek için akıllı sözleşmelerin Solidity kaynak kodunu girdi olarak alan bir statik analiz aracı SA-Solidity'i geliştirdik. SA-Solidity, Solidity kaynak kodunu ayrıştırma ağacına dönüştürür ve bu ağacı sorgulayarak hataları bulur. Ayrıca, tek bir aracın tüm hataları tespit etmek için yeterli olmadığını ve bilinen SmartCheck ve Securify araçlarının da SmartEmbed'in deneysel akıllı sözleşme kümesindeki farklı hataları tanımlayabildiğini gösterdik. Bunlardan sonra, Solidity için önceki tüm hata algılama araçları tarafından ortak olarak rapor edilen dosyalardaki hataları tahmin eden makine öğrenmesi tabanlı hata tahmin aracını (MLBP-Solidity) geliştirdik. MLBP-Solidity, büyük olasılıkla hata içeren dosyaların bir alt kümesine odaklanmasını sağlayarak geliştiricinin daha az efor sarfetmesini sağlayacaktır. Deneysel sonuçlarımız, tahmin edilen hatanın türüne bağlı olarak MLBP-Solidity'nin %90-99 doğruluğa ulaştığını göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xiii
1. INTRODUCTION 1
1.1. Motivation 1
1.2. Contributions 3
1.3. Outline 3
2. BACKGROUND 4
2.1. Blockchain 4
2.2. EVM 5
2.3. Solidity 6
3. RELATED WORKS 7
4. METHODOLOGY 10
4.1. Bugs in Solidity Language 10
4.1.0.1. Return Value of transfer and transferFrom 12
4.1.0.2. Empty Loops 12
4.1.0.3. Hardcoded Address Value 13
4.1.0.4. Balance Checking Method 13
4.1.0.5. Byte Usage 14
4.1.0.6. Constant Function 14
4.1.0.7. Empty If Statements 14
4.1.0.8. Wrong Way of Erasing An Array 15
4.1.0.9. Gas-Costly Function 15
4.1.0.10. Locked Money 16
4.1.0.11. Accessing Array Length in Loop 16
4.1.0.12. Tentative Compiler Version 16

4.1.0.13. Integer Division 18
4.1.0.14. Keyword Private 18
4.1.0.15. External Library Usage 19
4.1.0.16. Flag Usage 19
4.1.0.17. Method Send Usage 19
4.1.0.18. Ether Transfer in Loop 19
4.1.0.19. Risky Authorization Check 21
4.1.0.20. Throw Usage 21
4.1.0.21. Suicide Usage 21
4.1.0.22. Integer Overflow/Underflow 22
4.1.0.23. Cryptology Functions 22
4.1.0.24. Timestamp Usage 22
4.1.0.25. Default Visibility Level 22
4.1.0.26. Approve Function Usage 22
4.1.0.27. Payable Fallback Function 25
4.1.0.28. Unsafe Types 25
4.1.0.29. Revert Usage 25
4.1.0.30. Pure Usage 27
4.1.0.31. Unnecessary Return Statement 27
4.1.0.32. View Usage 28
4.1.0.33. Assembly Usage 28
4.1.0.34. Predictable Fields in Cryptology Functions 28
4.1.0.35. Redundant Fallback Function 28
4.1.0.36. Multiple Inheritance 30
4.1.0.37. Shadow Variable Usage 30
4.1.0.38. Mutex Pattern 30
4.1.0.39. Extarnal Function Pattern 31
4.1.0.40. Unnecessary Bequest 31
4.1.0.41. Check-Effect-Interaction Pattern 32
4.1.0.42. Logger Fallback Function 33
4.1.0.43. State Changing Order 33

4.1.0.44. Confusion Between Event and Function 34
4.1.0.45. Restricted Write 34
4.1.0.46. Restricted Transfer 34
4.1.0.47. Call Function with No Data 35
4.1.0.48. Storage-Related Loop 35
4.1.0.49. Constant-Outcome Loop 35
4.1.0.50. Loop Fusion 37
4.1.0.51. Mortal Pattern 37
4.1.0.52. Multiple Return Values 38
4.1.0.53. Rejector Pattern 38
4.1.0.54. Selfdestruct Usage 38
4.2. SA-Solidity Working Process 39
4.3. MLBP-Solidity Working Process 42
5. EXPERIMENTS AND RESULTS 46
5.1. SA-Solidity Results 46
5.2. MLBP-Solidity Results 48
5.2.1. Evaluation of Bug Detection Tools For MLBP-Solidity 48
5.2.2. Evaluation of MLBP-Solidity 50
5.2.3. Experimental Results 51
5.3. Discussion 52
6. CONCLUSION 55
REFERENCES 56

LIST OF FIGURES

Figure 2.1. Sample Blockchain Structure 4

Figure 4.1. Wrong Usage of TransferFrom of OpenZeppelin library 12

Figure 4.2. Empty Loop 13

Figure 4.3. Hardcoded Address Usage 13

Figure 4.4. Balance Checking Method 13

Figure 4.5. Bytes Usage 14

Figure 4.6. Constant Function 14

Figure 4.7. Empty If Statement 14

Figure 4.8. Wrong Way of Erasing An Array 15

Figure 4.9. Gas-Costly Function 15

Figure 4.10. Locked Money 16

Figure 4.11. Array Length in Loop 17

Figure 4.12. Compiler Version Declaration 17

Figure 4.13. Integer Division 18

Figure 4.14. Keyword Private 18

Figure 4.15. External Library Usage 19

Figure 4.16. Flag Usage 20

Figure 4.17. Method Send Usage 20

Figure 4.18. Ether Transfer in Loop 20

Figure 4.19. Risky Authorization Check 21

Figure 4.20. Throw Usage 21

Figure 4.21. Suicide Usage 22

Figure 4.22. Safe Math Library 23

Figure 4.23. Cryptology Functions 24

Figure 4.24. Timestamp Usage 24

Figure 4.25. Default Visibility Level 24

Figure 4.26. Approve Function of EIP20 Contract 24

Figure 4.27. Payable Fallback Function 25

Figure 4.28. Unsafe Types 26

Figure 4.29. Revert Usage 26

Figure 4.30. Pure Usage 27

Figure 4.31. Unnecessary Return Statement 27

Figure 4.32. View Usage 28

Figure 4.33. Assembly Usage 29

Figure 4.34. Predictable Fields in Cryptology Functions 29

Figure 4.35. Redundant Fallback Function 29

Figure 4.36. Multiple Inheritance 30

Figure 4.37. Shadow Variable 30

Figure 4.38. Mutex Pattern 31

Figure 4.39. External Function Pattern 31

Figure 4.40. Unncessary Bequest 32

Figure 4.41. Check-Effect-Interaction Pattern 32

Figure 4.42. Logger Fallback Function 33

Figure 4.43. State Changing Order 33

Figure 4.44. Confusion Between Event and Function 34

Figure 4.45. Restricted Write 34

Figure 4.46. Restricted Transfer 35

Figure 4.47. Call Function With No Data 35

Figure 4.48. Storage Related Loop 36

Figure 4.49. Constant-Outcome Loop 36

Figure 4.50. Loop Fusion 37

Figure 4.51. Mortal Pattern 37

Figure 4.52. Multiple Return Values 38

Figure 4.53. Rejector Pattern 38

Figure 4.54. Selfdestruct Usage 39

Figure 4.55. SA-Solidity Overview 39

Figure 4.56. JSON Parse Tree of Part of a Contract 40

Figure 4.57. XML View of Part of a Contract 41

Figure 4.58. XPath Queries to Check Locker Ether 41

Figure 4.59. MLBP-Solidity Overview 42

Figure 5.1. Venn Diagram of the Rule Violations Checked by MLBP-Solidity, Securify, and SmartCheck 49

LIST OF TABLES

Table 4.1. Rules priority mapping 11

Table 4.2. Bug prediction metrics of Solmet 43

Table 4.3. Smart contract bugs and associated bug prediction metrics 44

Table 5.1. Number of buggy files reported by SA-Solidity for all 22,725 files . 47

Table 5.2. Number of buggy files reported by SA-Solidity, Securify, and SmartCheck for all 14,464 files 48

Table 5.3. Experimental results 52

1. INTRODUCTION

In this chapter, we explain the driving factors behind our research, our contributions, and provide an outline of this thesis.

1.1. Motivation

A blockchain is a decentralized database that contains sequential, cryptographically linked blocks of digitally marked asset transactions managed by an aggregate model [1]. In other words, a blockchain is a system wherein there is no central authority but rather a full authorization for everyone. The essence of a blockchain is the mistrust of any individual or organization.

Because of their decentralized nature, blockchain platforms are also used to build cryptocurrencies that no one can regulate. For example, Bitcoin, one of the most common crypto currencies, reached a market value of US$ 160,731,625,135 in June 2019 [2].

A smart contract is software that describes a real-world contract, i.e, it stores rules for negotiating the terms of an agreement and is capable of automatically checking its success and enforcing the terms or acts agreed upon [3]. Blockchain technology is the perfect platform for smart contract storage and operation in terms of protection, durability, consistency, and reliability.

Ethereum is an open-source blockchain platform explicitly designed to build smart contracts [4]. Smart contracts can be described as immutable applications that operate on the virtual machine Ethereum (EVM) [5]. This immutability implies that even if they are wrong and no other party can change them, the parties recognize the conditions and codes of the contracts.

Blockchain platforms, popular for storing digital properties, are enticing to cyber pirates because of smart contract bugs and mistakes. Therefore, it is important to develop intelligent, error-free contracts. For instance, in June 2016, US $150 million was robbed from the famous smart DAO contract [6]. People who develop smart contracts need tools that can detect faults and bugs beyond question.

Solidity is a programming language for the development of smart contracts on EVM. Smart contracts operate on EVM and are spread out to blocks. To prevent unfamiliar situations, bugs and errors in smart contracts should be detected with the help of static analysis, dynamic analysis, or artificial intelligence methods.

Static analysis is an analysis process without running computer programs [7]. Dynamic program analysis, however, is a method of software analysis performed with real or virtual processors by running programs [8]. Bug prediction is a learning method used by computers to predict possible bugs, defects, and mistakes by using program metrics. It helps in understanding the bug-related properties of software.

We developed a tool named Static Analysis for Solidity (SA-Solidity) in this research, which uses the approach of static analysis to find potential bugs in smart contracts written in Solidity programming language. We preferred static analysis because it provides complete coverage of source code with all potential bugs and errors. We trained Machine Learning Bug Predictor for Solidity (MLBP-Solidity) to predict buggy files in smart contracts in light of SA-Solidity and two other known bug detection tools, SmartCheck and Securify. It allows developers to quickly check bugs in smart contracts before their deployment. To the best of our knowledge, this is the first predictive model trained for bug prediction in Solidity language.

1.2. Contributions

In this thesis, the following contributions are made,

(i) We developed a tool called Static Analysis for Solidity (SA-Solidity), which works on Solidity source code, to detect bugs in smart contracts. We demonstrated that SA-Solidity and two proven bug detectors, SmartCheck and Securify could detect specific bugs in the experimental set of smart contract files found in SmartEmbed.

(ii) We implemented and trained (MLBP-Solidity). To the best of our knowledge, this the first predictive model trained for bug prediction in Solidity language.

(iii) In the SmartEmbed experimental set of smart contract files, we predicted seven different types of bugs. Our test results indicated that MLBP-Solidity succeed with 90-99% accuracy, contingent upon the kind of predicted bug.

1.3. Outline

We give a background on blockchain systems, EVM, and Solidity in Chapter 2. In Chapter 3, we present related work on blockchain bugs' characteristics and existing static and dynamic analysis tools for Solidity. We describe our knowledge base and methodology for SA-Solidity and MLBP-Solidity in Chapter 4. We present and discuss our experimental results in Chapter 5. Finally, we describe future work and our overall contributions in Chapter 6.

2. BACKGROUND

2.1. Blockchain

A blockchain is a structure wherein data are not stored at a single point and all data are available to all stakeholders. In other words, there is no central authority in a blockchain, and all data store in every participant. Anyone can participate as a stakeholder in a blockchain by a consensus model. The data flow in a blockchain is unidirectional. A blockchain consists of blocks, and a block primarily has four parts: data, timestamp, hash value, and previous block hash value. The data part contains transaction information, and the hash value is the unique ID of a block. Figure 2.1 shows a sample blockchain structure. Each block is added to the chain if it is appropriate for the consensus model and cannot be changed again. Blocks are appended to the end of the chain. Furthermore, it is available for all to view when a new block is attached to the chain. As in a blockchain, each block contains the hash of its preceding block, so a block's hash is modified if anyone attempts to change the data in it. Therefore, the attacker has to alter the next block's previous hash as well. This will also alter the actual hash of the next block. Ultimately, the attacker must manipulate every block's hashes, which is difficult due to the recalculation of all those hashes. Thus, the data in a blockchain prove flawlessly and maintain their validity.

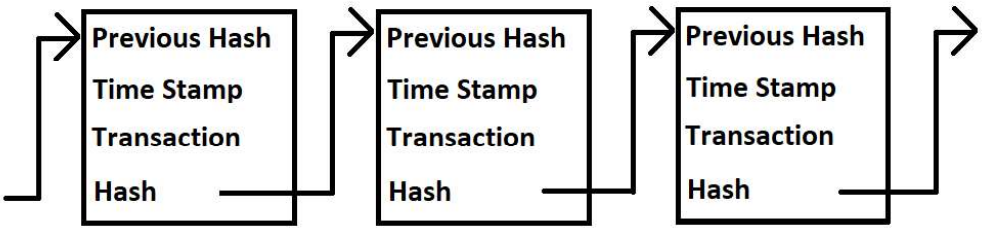

Figure 2.1. Sample Blockchain Structure

The purpose of a blockchain is to allow digital content to be stored, disseminated but not altered by anyone. Bitcoin is the first implementation of blockchain systems in real-world. It was introduced in January 2009 by Satoshi Nakamoto [9] as "a new electronic cash system that is fully peer-to-peer, with no trusted third party".

2.2. EVM

Vitalik Buterin described EVM in a white paper [10] at the end of 2013 to build decentralized applications. Buterin stated that Bitcoin requires a scripting language to build applications. Therefore, he suggested the creation of a new framework with a more general language for scripting.

Ethereum as a whole can be seen as a transaction-based state machine [11]. Every node on the system holds a copy of the transaction history and is run by volunteers.

A specific gas fee is defined for each operation performed on this platform. In other words, the gas charge represents the cost of the operation [12]. Smart contracts are operated on the miners' computers, and for each process carried out there, the miners earn as much gas as the value of that process. The gas charge can be converted into real money as Ether. Gas limit defines the maximum amount of gas that can be spent on a specific transaction. A higher gas fee implies that miners should conduct further efforts to execute the transaction. Insufficient limit of gas price may cause miners to disregard the transaction.

There are two different types of accounts on EVM [13]. The first is external accounts, which can be created by humans and contain money as Ether. The second is created when defining smart contracts. Currently, there are 1.7 million smart contracts in the Ethereum blockchain; in other words, there are 1.7 million accounts on the Ethereum blockchain, and 500.000 of them are active [14]. Each account has public and private keys. Account addresses can be treated as public keys. Clients can communicate with smart contracts through these addresses.

2.3. Solidity

There have been many languages developed for EVM. Solidity is the most used object-oriented programming language for smart contract development. Solidity was inspired by C++, JAVA, and Python [15]. Thus, it is easy to learn for many developers.

As Solidity language is statically typed, it has an integer, string, bool, address, and map data types. Solidity codes are compiled to byte codes, which are executable on EVM. Moreover, Solidity language is object-oriented and supports inheritance.

As smart contracts work on blocks, Solidity stores variables that can access the blocks' information, i.e., block hash number, contract address, address of the caller of the contract, amount of Ether sent to the contract, amount of Ether in the contract account, current block timestamp, current block gas limit, call data and gas price of the transaction. This information is represented as global variables in Solidity smart contracts.

Solidity smart contracts generally consist of multiple functions, state variables, and local variables. Functions are the gateways to smart contracts. Clients and smart contracts can access the smart contracts by calling the functions defined on them. Solidity language also supports function development for smart contracts. Each function call is considered a transaction. As EVM is a state machine, accessing state variables defined on smart contracts may change the state on the chain. In addition, accessing state variables is costly. In other words, using state variables consumes more gas than using local variables. Apart from these, there are local variables in Solidity as well. These are available while the function is on execution. In other words, they are just temporary variables and are not stored in contract storage.

3. RELATED WORKS

Bug checking on Ethereum smart contracts is an active research field in software engineering. A survey about bug characteristics on blockchain systems was conducted by Wan et. al. [16], who examined 1108 bug reports and classified them into 10 different categories: semantic, environment, GUI, concurrency, build, security, memory, performance, compatibility, and hard fork. They indicated that semantic bugs are the most common ones, it takes tremendous amount of time to patch security problems. They also revealed that dependability of blockchain systems is weaken by bugs. They stated that bug characteristics show similar trends in blockchain systems as well.

Additionally, a research about attacks on Ethereum smart contracts was conducted by Atzei et al. [17]. The survey discussed the common mistakes and errors that cause buggy smart contracts. Atzei et al. also stated that a noteworthy portion of the mistakes that make Ethereum smart contracts error-prone are related to Solidity features. Most of the previous research aimed to develop tools that analyze EVM bytecodes. For example, Luu et al. [18] developed the tool OYENTE, which is based upon symbolic execution to check vulnerabilities in EVM bytecode. The tool constructs a control-flow graph and produces a set of symbolic traces. After that, a constraint solver helps analyze the traces. Mythril is another symbolic execution tool that works using EVM bytecode [19]. Unlike OYENTE, Mythril analyzes with concolic testing [20], and Fontein et al. [21] showed that Mythril can detect more errors than OYENTE. Nikolic et al. [22] also developed a tool called MAIAN, which works on EVM bytecode to detect greedy, prodigal, and suicidal contracts. The tool uses the symbolic execution technique to detects contracts that violate safety and liveness properties. However, none of these tools are guaranteed to explore all program paths.

In contrast to symbolic execution, there are several static analysis tools to improve smart contracts' reliability. For example, Brent et al. [23] developed a static analysis tool called Vandal to detect security problems in Ethereum smart contracts. The tool converts EVM byte-code to intermediate representation and executes logic specifications written in Souffle language. Brent et al. claimed claim that Vandal is much faster than OYENTE and Mythril. Moreover, Tsankov et al. [24] developed Securify, a static analysis tool, to show whether contracts' codes are safe or unsafe based on exploring all behaviors of the contracts. Securify classifies secure and insecure behaviors from a contract's semantic facts, obtained from decompiled EVM bytecode. After these steps, the tool checks the set of security patterns and shows the results to the user. Tikhomirov et al. [25] developed SmartCheck, a static analysis tool that queries XML parse tree, that obtained from Solidity source code to check vulnerability patterns in smart contracts. In addition, Gao et al. [26] developed SmartEmbed to detect the erroneous codes by using similarities in the source code of the contracts. Unlike the previous works, however, TonTon Hsien-De Huang [27] knuckled down on reducing labor costs and developed a software to hunt buggy contracts. The tool converts bytecode to RGB-colored code and gives them to convolutional neural network to extract features and train. There have only been a few studies about finding gas-costly patterns that do not lead security problem but the cost. Chen et al. [28] proposed finding gas-costly patterns to optimize smart contracts and EVM compiler.

There is also little research on comparing bug detection tools. Parizi et al. [29] conducted a study to measure the effectiveness and accuracy score of smart contract vulnerability detection tools. They used OYENTE, Mythril, Securify, and SmartCheck tools and their results implied that SmartCheck could be the most effective tool despite not being more accurate than Mythril.

A tool called FSolidM was developed by Mavridou et al. [30] to generate secure smart contracts as well. The tool uses finite-state machines and converts them to Solidity source code. There have been a few research works about collecting metrics in Solidity. For instance, Peter Hegadüs [31] developed a software called SolMet to collect Solidity source code metrics and to analyze the complexity of smart contracts.

4. METHODOLOGY

In this chapter, we discuss our research on potential undesired mistakes.

4.1. Bugs in Solidity Language

We defined 54 different rules as our knowledge base for SA-Solidity and then divided these rules into four different groups. Our main starting point during the classifying process was to measure the effect of damage on smart contracts. We classified 7 of the rules as first-priority, which meant they were vulnerabilities that could either cause the smart contract to be damaged and lose money or cause the rules created in the smart contract to fail. Then, we classified 13 of the remaining rules as second-priority rules, i.e., rules that contained the detection of code particles that would not cause the smart contract to be attacked directly but would prevent it from working effectively. We grouped 28 of the remaining rules as third-priority, i.e., they generally detected the code smells. These codes did not directly damage the smart contract or make it work ineffectively. Errors in this group generally indicated that the software developer used old versions of functions, did not use up-to-date methods or used expressions that decreased the readability of the code. We evaluated the last 6 rules as the fourth-priority, as they contained good software patterns created for the Solidity language. Not using these patterns did not create any added value for the software developer. However, our aim here was to show software developers that they are on the right track or using a previously approved and known solution. Table 4.1 shows the knowledge base rules' priority levels.

Table 4.1. Rules priority mapping

Rule Number	**Priority Level**	**Rule Number**	**Priority Level**
4.1.0.1	3	4.1.0.28	3
4.1.0.2	2	4.1.0.29	3
4.1.0.3	3	4.1.0.30	3
4.1.0.4	2	4.1.0.31	3
4.1.0.5	2	4.1.0.32	3
4.1.0.6	3	4.1.0.33	3
4.1.0.7	3	4.1.0.34	1
4.1.0.8	2	4.1.0.35	3
4.1.0.9	2	4.1.0.36	3
4.1.0.10	1	4.1.0.37	3
4.1.0.11	2	4.1.0.38	4
4.1.0.12	3	4.1.0.39	4
4.1.0.13	3	4.1.0.40	3
4.1.0.14	3	4.1.0.41	4
4.1.0.15	3	4.1.0.42	3
4.1.0.16	3	4.1.0.43	1
4.1.0.17	2	4.1.0.44	3
4.1.0.18	2	4.1.0.45	1
4.1.0.19	1	4.1.0.46	1
4.1.0.20	2	4.1.0.47	2
4.1.0.21	3	4.1.0.48	2
4.1.0.22	3	4.1.0.49	3
4.1.0.23	2	4.1.0.50	2
4.1.0.24	1	4.1.0.51	4
4.1.0.25	3	4.1.0.52	3
4.1.0.26	3	4.1.0.53	4
4.1.0.27	3	4.1.0.54	4

4.1.0.1. Return Value of transfer and transferFrom. ERC-20 [32] is a token standard. It has functions i.e. transfer, transferFrom, approve. Those who implement tokens usually use the standard. However, sometimes developers ignore some features of the functions while override them. For example, when an exception occurs in the transfer or transferFrom function, it will return false to the caller function. Therefore, it is better to avoid the default return value as false. Figure 4.1 shows an incorrect usage of the transferFrom function .

```
contract StandardToken is ERC20, BasicToken {
 function transferFrom( address from, address to, uint256 value)
    public returns (bool)
  {
    require(value <= _balances[from]);
    require(value <= _allowed[from][msg.sender]);
    require(to != address(0));
    _balances[from] = _balances[from].sub(value);
    _balances[to] = _balances[to].add(value);
    _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(
        value);
    emit Transfer(from, to, value);
  }
}
```

Figure 4.1. Wrong Usage of TransferFrom of OpenZeppelin library [33]

4.1.0.2. Empty Loops. Empty loops are unnecessary and cause extra gas consumption in Solidity. It is always better to get rid of them. Figure 4.2 shows an empty loop.

```
address[] public myArray;
uint lengthOfArray = myArray.length;
for ( uint j=0;  j < lengthOfArray; j++) {
    // Empty loop is unncessary. Consumes extra gas
 }";
```

Figure 4.2. Empty Loop

4.1.0.3. Hardcoded Address Value. The address type serves as a base for all contracts and contracts inherit some members and functions from the address type. When a smart contract sends or receives money, address types are used. According to this knowledge, hardcoded address values should be avoided, as the address might be used for malicious activity. Figure 4.3 has a constant address variable.

```
contract Sample {
// Address might be unused or malicious
address constant public myAddress =  0
   xE0f5206BBD039e7b0592d8918820024e2a7437b9;
}
```

Figure 4.3. Hardcoded Address Usage

4.1.0.4. Balance Checking Method. this.balances should be checked using “>=” or “<=” instead of “==”. The main reason we do not suggest using “==” is that strict balance equality causes security issues. Attackers can manipulate it using a selfdestruct method. Figure 4.4 shows a secure way to check balance.

```
if (address(this).balance >= 10 ether ) {
   /* ... */
 }
```

Figure 4.4. Balance Checking Method

4.1.0.5. Byte Usage. There are multiple ways to use data type byte in Solidity. To decrease the gas consumption, "bytes" should be used instead of "byte[]". Figure 4.5 shows how to define byte variables with correct and incorrect forms.

```
contract C {
bytes goodChoice;
byte[] badChoice;
}
```

Figure 4.5. Bytes Usage

4.1.0.6. Constant Function. Keyword constant is not used for functions anymore. Therefore, functions should be declared as View instead of constant. Figure 4.6 shows a constant function.

```
function hasEntry(address _account) public constant returns (bool)
   {
        return listIndex.length > 0 && _account == listIndex[list[
            _account].index];
    }
```

Figure 4.6. Constant Function

4.1.0.7. Empty If Statements. Empty if statements are unnecessary as they are ghost codes. Therefore, we suggest removing them. Figure 4.7 shows an if block.

```
    function Foo(address adr){
        if(adr = msg.sender) {
            // Empty statement !
        }
    }
```

Figure 4.7. Empty If Statement

4.1.0.8. Wrong Way of Erasing An Array. Setting an array length to zero may cause storage overlap. Therefore, as a safe method, the delete keyword should be used to erase an array. Figure 4.8 shows a bad practice of array erasing.

```
contract C {
  function erase(address[] array) {
          array.length=0; // This is the bad practice !
    }
}
```

Figure 4.8. Wrong Way of Erasing An Array

4.1.0.9. Gas-Costly Function. Functions should only be responsible for a single job. Multiple usage of send, transfer and call functions, which causes high gas consumption, within the same function will cause both the whole transaction to be reverted and the entire gas to be exhausted when one of the transactions gets an error. In this case, we recommend that the functions should be arranged as the smallest possible size and to be responsible for only one job, since it may cause unnecessary gas consumption. Figure 4.9 shows a gas-costly function.

```
contract C {
    function Foo(uint balance1, uint balance2) public {
          msg.sender.transfer(balance1);
          owner.transfer(balance2);
    }
}
```

Figure 4.9. Gas-Costly Function

4.1.0.10. Locked Money. A way to withdraw Ether must be implemented by contracts programmed to get Ether. Otherwise, the money will have a possibility to remain locked in the contract. The recommended method to withdraw is using a transfer function. However, call or send methods can be used as well. Figure 4.10 shows a contract that has a payable function to receive Ether but does not have a function to send it. Therefore, this contract may have a locked money problem.

```
contract C {
  function receiver() payable {
      require(msg.value > 0);
  }
}
```

Figure 4.10. Locked Money

4.1.0.11. Accessing Array Length in Loop. Solidity has state and local variables. Accessing the state variables is more expensive than accessing the local variables. In light of this information, accessing the length of an array continuously in the loop causes extra gas consumption. Alternatively, assigning the array length to a local variable before the loop and accessing the variable throughout the loop will reduce gas consumption. Figure 4.11 shows correct and incorrect ways to use array length in a loop.

4.1.0.12. Tentative Compiler Version. Each compiler version has different features. Therefore, using a specific compiler version and writing the codes based on that version's rules is vital. Otherwise, there will be a possibility of losing one or more of the available features with any update, which would cause the contract to be error-prone. Figure 4.12 shows correct and incorrect ways to declare compiler version.

```
address[] public addressArray;
 addressArray.push(currentAddress);
 ...
// Correct way !
// Array size assignment
uint lengthOfArray = addressArray.length;
for (uint count=5; count < lengthOfArray; count++) {
    // TODO
}
// Wrong way !
for (uint count=5; count < addressArray.length; count++) {
    // TODO
};
```

Figure 4.11. Array Length in Loop

```
pragma solidity 0.4.18; // The good way
pragma solidity ^0.4.18; // The bad way
```

Figure 4.12. Compiler Version Declaration

4.1.0.13. Integer Division. For integer division, the quotient used to be rounded down in Solidity; however, from version 0.4.0 onward, the literal division is translated to a rational number. For example, 11 / 5 does not equal 2, instead it is equal to 2.2. Rounding down may cause inaccurate results. Therefore, it is always best to be careful in integer division. Figure 4.13 shows an integer division.

```
contract SmartDivision {
    function getDivided(uint divided, uint divider) public returns
       (uint) {
        uint result = divided / divider;
        return result;
    }
}
```

Figure 4.13. Integer Division

4.1.0.14. Keyword Private. The keyword "private" does not make a variable or function invisible. Therefore, it is wise to avoid storing sensitive data by relying on it. In Figure 4.14, function numberGenerator has logic to generate numberGenerator numbers, but this logic is visible to everyone.

```
contract C {
    uint numOfParticipant;
     // Function is private but the function can be seen by any
        user !
     function numberGenerator(uint256 p) private view returns (
        uint256) {
        return uint256(sha3(block.difficulty, now)) % p;
    }
}
```

Figure 4.14. Keyword Private

4.1.0.15. External Library Usage. Solidity is an object-oriented language, and there are libraries to import. However, external libraries might be malicious. Therefore, any library that will be used should be checked. Figure 4.15 shows how to call an external library in a contract.

```
contract C {
    using X for X.Data; // X Library should be inspected.
    X.Data ourMapping;
    function register(uint value) public {
        require(ourMapping.insert(value));
    }
}
```

Figure 4.15. External Library Usage

4.1.0.16. Flag Usage. Flags are a strong code smell and will affect the behavior of a program based on their states. Any flag usage in the loop should be reviewed. Instead of a flag, break, continue, and return should be used. Figure 4.16 shows a contract whose function, named HasCustomer, uses a boolean variable as a flag. Instead of this, however, the break statement could be used, which would help to reduce gas usage.

4.1.0.17. Method Send Usage. Method transfer should be used instead of method send. Since method transfer handles exceptions without requiring the developer's effort, it is more secure than method send. Figure 4.17 shows an unsafe way to send Ether.

4.1.0.18. Ether Transfer in Loop. Transfer, send, or call methods should not be used in a loop. Otherwise, they will cause extra gas consumption. In Figure 4.18, method send is used in a loop. Therefore, it may drain all gas from the contract.

```
contract  C {
    address[] public addressIndices;
    function HasCustomer() returns (bool)
    {
        bool hasFound;
        for (uint count=0; count < addressIndices.length; count++)
           {
            if (addressIndices[count] > 0 )
                hasFound = true;
        }
        return hasFound;
    }
}
```

Figure 4.16. Flag Usage

```
contract C {
  function send(address _receiver) payable {
    _receiver.send(msg.value);
  }
}
```

Figure 4.17. Method Send Usage

```
contract C {
    address[] public players;
    uint fee = 2 Ether;
    function awardWinners() public {
        for(uint j = 0; j<players.length; j++){
            players[i].send(fee);
        }
    }
}
```

Figure 4.18. Ether Transfer in Loop

4.1.0.19. Risky Authorization Check. Msg.sender should be used instead of tx.origin. Tx.origin-based authorization checks can lead to harmful callback attacks. Figure 4.19 shows a risky authorization check.

```
contract Victim {
    address currentOwner;
    function TxOriginVictim() {
        currentOwner = msg.sender;
    }
    function SendEther(address addr, uint fee) public {
    require(tx.origin == currentOwner);
        addr.transfer(fee)();
    }
}
```

Figure 4.19. Risky Authorization Check

4.1.0.20. Throw Usage. The revert function should be used instead of a throw statement. The reason we suggest using function revert is that it is usually called low-cost throw because it refunds unused gas. Figure 4.20 shows an expensive way to revert transaction.

```
function bar() public {
    throw;
};
```

Figure 4.20. Throw Usage

4.1.0.21. Suicide Usage. Suicide is an old-fashioned way to destroy a contract. Selfdestruct command should be preferred instead of suicide. Figure 4.21 shows an outdated way to kill a contract.

```
function ownerKill() onlyOwner {
    suicide(owner); // suicide usage should be avoided.
};
```

Figure 4.21. Suicide Usage

4.1.0.22. Integer Overflow/Underflow. SafeMath [34] library should be used in all arithmetic operations. The reason we recommend this library is that smart contracts can fail because the largest number, and smallest number limits, 2^{256} and -2^{256} respectively, can easily be exceeded in Solidity. However, SafeMath library catches such errors by itself and does not compromise the reliability of the program. Figure 4.22 shows an example of SafeMath library.

4.1.0.23. Cryptology Functions. In Solidity, sha3 and keccak256 are aliases for the same function. However, keccak256 is more accurate. Thus, we suggest using keccak256 instead of sha3. Figure 4.23 shows sha3 usage.

4.1.0.24. Timestamp Usage. Timestamp usage should be avoided because it might affect the predictable fields. In Figure 4.24, variable theGuess can be easily predicted due to its dependency on timestamp variable now.

4.1.0.25. Default Visibility Level. There are several visibility levels of functions in Solidity. The default level is public. We argue that it is necessary to explicitly define visibility to avoid confusion. Figure 4.25 shows functions whose visibility is defined or default.

4.1.0.26. Approve Function Usage. Approve function of ERC-20 standard should solely be used to modify the allowed amount from 0 or to 0. Since the approve function is vulnerable. Figure 4.26 shows a sample approve function.

```
library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns(
       uint256) {
        if (a == 0) { return 0;}
        uint256 c = a * b;
        require(c / a == b);
        return c;
    }
    function div(uint256 a, uint256 b) internal pure returns(
       uint256) {
        require(b > 0); // Solidity only automatically asserts when
            dividing by 0
        uint256 c = a / b;
        return c;
    }
    function sub(uint256 a, uint256 b) internal pure returns(
       uint256) {
        require(b <= a);
        uint256 c = a - b;
        return c;
    }
    function add(uint256 a, uint256 b) internal pure returns(
       uint256) {
        uint256 c = a + b;
        require(c >= a);
        return c;
    }
    function mod(uint256 a, uint256 b) internal pure returns(
       uint256) {
        require(b != 0);
        return a % b;
    } }
```

Figure 4.22. Safe Math Library in TheWeakestHodler Contract [35]

```
function numberGenerator(uint p) private view returns (uint) {
        return uint(sha3(block.difficulty, now))%p;
    }
```

Figure 4.23. Cryptology Functions

```
function guess(bool theGuess) external payable returns (bool) {
   if (theGuess == now % 2 > 10) { // now is used as timestamp.
          uint fee = msg.value / 10;
          msg.sender.transfer(msg.value* 2 - fee);
   }
}
```

Figure 4.24. Timestamp Usage

```
function foo() public;
function foo() internal;

should be used instead of

function foo()
```

Figure 4.25. Default Visibility Level

```
contract EIP20 is EIP20Interface {
{
    function approve(address currentAddress, uint256 money) public
       returns (bool) {
        allowed[msg.sender][currentAddress] = money;
        Approval(msg.sender, currentAddress, money);
        return true;
    }
}
```

Figure 4.26. Approve Function of EIP20 Contract [36]

4.1.0.27. Payable Fallback Function. If there is a payable function, there should be a payable fallback function as well. The reason we recommend is that the fallback function will be revoked on Ether transfer. If there is no payable function, the contract will be unable to receive Ether and will throw an exception. Figure 4.27 shows a contract that cannot receive Ether due to the absence of a payable fallback function.

```
contract C {
    address currentOwner;
    function foo () public {
        currentOwner = msg.sender;
    }
    function withdrawMoney() payable public {
        require(currentOwner == msg.sender);
        owner.send(this.balance);
    }
    function putMoney(uint256 money) payable public {
        require(msg.value == money);
        // no action !
    }
}
```

Figure 4.27. Payable Fallback Function

4.1.0.28. Unsafe Types. Unsafe type usage should be avoided. It is better to declare a variable type to increase readability. Figure 4.28 shows a variable whose type is not defined.

4.1.0.29. Revert Usage. Revert function usage should be avoided inside if statements. The require function should be preferred to follow the state-of-art in Solidity. Figure 4.29 shows an old-fashioned way to create a logic with if statements.

```
contract Storage {
    var number;
    function store(uint256 num) public {
        number = num;
    }
    function retreive() public view returns (uint256){
        return number;
    }
}
```

Figure 4.28. Unsafe Types

```
contract Holder {
    uint public holdUntil;
    address public holder;
    function withdraw (uint a, uint b) external {
        if (now < holdUntil){
            revert();
        }
        holder.transfer(this.balance);
    }
}
```

Figure 4.29. Revert Usage

4.1.0.30. Pure Usage. Functions that read or change the state should not be declared as pure. Only functions that do not read or update the EVM state should be declared as pure. Figure 4.30 shows a pure function.

```
function calculate() internal pure returns(uint sum){
        uint valueFirst = 1;
        uint valueSecond = 2;
        sum = valueFirst + valueSecond;
        return sum;
    }
```

Figure 4.30. Pure Usage

4.1.0.31. Unnecessary Return Statement. Functions that do not return any value should not have return statement. Otherwise, it will decrease readability and mislead the software developer. In Figure 4.31, correct and incorrect functions are shown. Function fooIncorrect does not return any variable but contains the keyword returns. Therefore, this usage is wrong. However, function fooCorrect has the keyword returns and returns variable. Thus, it is the correct usage.

```
//The function does not return any value !
function fooIncorrect(uint256 temp) returns (uint256){
    uint256 variable = temp;
}
Correct usage !
function fooCorrect(uint256 temp) returns (uint256){
    uint256 variable = temp
    return variable;
}
```

Figure 4.31. Unnecessary Return Statement

4.1.0.32. View Usage. Functions that change the state should not be declared as view, but functions that do not write the state should be. Figure 4.32 shows a view function that should also be declared as pure because it does not read from the state. Pure functions cannot read or update states, whereas view functions can do the former.

```
function calculate() public view returns(uint sum){
        uint a = 1;
        uint b = 2;
        sum = a + b;
        return sum;
    }
```

Figure 4.32. View Usage

4.1.0.33. Assembly Usage. According to EVM documentation, inline assembly is a way to access EVM at a low level. This circumvents some significant security features and controls. It should only be used for tasks that require it, and only when it is used with trust by the user. Therefore, the assembly keyword usage should be avoided. Figure 4.33 shows a library whose function has inline assembly codes.

4.1.0.34. Predictable Fields in Cryptology Functions. Predictable field usage in keccak256 or sha3 should be avoided. It is not safe to generate numbers using predictable fields. Figure 4.34 shows function random. The function generates a random value, but since the random generation process depends on known variable now, the generated value can be easily estimated.

4.1.0.35. Redundant Fallback Function. With Solidity 0.4.0, fallback function is unnecessary on contracts. Because, the payment is automatically reverted when there is no fallback function. Figure 4.35 shows a contract that has an unneeded fallback function.

```
library GetCode {
    function Foo(address currentAddress) public () {
        assembly {
            let currentSize := extcodesize(currentAddress)
            k_opp := mload(0x20)
            mstore(0x40, add(k_opp, and(add(add(currentSize, 0x40),
                0x1f), not(0x1f))))
            mstore(k_opp, currentSize)
            extcodecopy(currentAddress, add(k_opp, 0x20), 0,
               currentSize)
        }
    }
}
```

Figure 4.33. Assembly Usage

```
//The return value can be predicted by any user !
function generateNumber(uint k) private view returns (uint) {
    return uint(sha3(block.difficulty, now)) % k; // now is
       predictable in here !
}
```

Figure 4.34. Predictable Fields in Cryptology Functions

```
contract C {
    function getTime() internal returns( uint ) {
        return now;
    }
    function() {
        revert();
    }
}
```

Figure 4.35. Redundant Fallback Function

4.1.0.36. Multiple Inheritance. Multiple inheritance should be avoided. It is a common code smell in all programming languages. Anything that can be done with multiple inheritance can also be done with single inheritance. Figure 4.36 shows contract C, which inherits contract X and contract Y.

```
contract C is X,Y {
    function foo() public X(2) Y(6){
        // do something!
    }
}
```

Figure 4.36. Multiple Inheritance

4.1.0.37. Shadow Variable Usage. Shadow variable usage should be avoided. This happens when a variable defined within a specific context and has the same name as an outer scope variable. This usage decreases readability. In Figure 4.37, there is a shadow variable defined in both the contract and function.

```
contract Storage {
    var number;
    function store(uint256 num) public {
        var number = num;
    }
}
```

Figure 4.37. Shadow Variable

4.1.0.38. Mutex Pattern. When an Ethereum contract invokes a function of another contract, the caller has to wait till the call ends. This is a good code practice to bypass different transactions' effects on a contract. Figure 4.38 shows how to implement a basic mutex pattern.

```
bool private hasLocked = false;
modifier LockControl{
require(!hasLocked);
hasLocked = true;
..
hasLocked = false;
}
```

Figure 4.38. Mutex Pattern

4.1.0.39. Extarnal Function Pattern. There is a difference between the gas consumption of public and external functions. Public functions can call internally too. Therefore, its behaviour is different from external functions. Public functions write all arguments to memory whereas external functions only use calldata which is cheap. Since it will reduce gas consumption, we recommend that all functions that are sure to be called externally should be defined directly with the external keyword. Figure 4.39 shows a defined external function.

```
function foo(uint[10] myArray) external returns (uint) {
        return myArray[5] * 4;
    }
```

Figure 4.39. External Function Pattern

4.1.0.40. Unnecessary Bequest. Unnecessary Bequest should be avoided. It is a common code smell. If the inheritance is appropriate, it is best to get rid of unneeded fields and methods in the subclass. There is an unnecessary bequest problem in Figure 4.38. Contract B is inherited from contract A. However, function Y is not used in contract B.

```
contract A {
    uint value;
    function X(uint _param) {
        value = _param + 5;
    }
    function Y() returns {
        return value;
    }
}
contract B is A {
     uint value;
     function X (uint _param) {
          value = _param + 10;
     }
}
```

Figure 4.40. Unncessary Bequest

4.1.0.41. Check-Effect-Interaction Pattern. State changing in a function has to be at the end. Otherwise, it will cause a DAO attack. Check-Effect-Interaction pattern is a good programming practice to avoid DAO attacks. Figure 4.41 shows an example of this pattern.

```
function Foo() public{
require(!hasEnded)
require(now > endingTime)
hasEnded = true;
winner.send(award);
}
```

Figure 4.41. Check-Effect-Interaction Pattern

4.1.0.42. Logger Fallback Function. Fallback functions are invoked when there is no match with function name as well. Therefore, we recommend logging in the fallback function so that the callers can detect their mistakes. Figure 4.42 shows good and bad coding practices for fallback functions.

```
// incorrect usage
function() payable { emit LogMoneySender(msg.sender); }

// correct usage
function() payable { require(msg.data.length == 0); emit
   LogMoneySender(msg.sender); }
```

Figure 4.42. Logger Fallback Function

4.1.0.43. State Changing Order. This is called Reentrancy or DAO attack as well. When a state variable is changed, after a contract uses call functions, the problem will be occur. Therefore, we suggest using Check-Effect-Interaction patterns to avoid the problem. Figure 4.43 shows a state changing order problem.

```
contract Foo {
  mapping(address => uint256) balances;
  uint fee = 3 Ether;
  function sell() {
    uint property = balances[msg.sender];
    uint finalValue = property - fee;
    msg.sender.send(finalValue)();
    balances[msg.sender] = 0;
  }
```

Figure 4.43. State Changing Order

4.1.0.44. Confusion Between Event and Function. This is a code smell to use the same name for events and functions. Therefore, we suggest using the keyword log to provide clarity about the naming and reduce ambiguity between functions and events. Figure 4.44 shows a confusion problem. We recommend adding the Log prefix to event names.

```
// bad
event Foo() {}
function foo() {}

// good
event LogFoo() {}
function foo() external {}
```

Figure 4.44. Confusion Between Event and Function

4.1.0.45. Restricted Write. Contract fields that any user may change should be checked because they may contain sensitive information. Figure 4.45 shows a function that allows everyone to be an owner.

```
function transferOwnership(address _owner) {
   owner = _owner; // Everybody can be an owner !
}
```

Figure 4.45. Restricted Write

4.1.0.46. Restricted Transfer. Variables that affect the amount of money to be transferred should be inspected. It is best to check if these variables can be influenced by other transactions. Figure 4.46 shows a function wherein the amount to send can be changed by different transactions.

```
function A( uint index ) {
    uint amount = balances[index];
    msg.sender.call.value(amount)();
    balances[index] = 0;
  }
```

Figure 4.46. Restricted Transfer

4.1.0.47. Call Function with No Data. The usage of a call function with no data should be avoided. If it is necessary, a gas limit should be added. Figure 4.47 shows a function that sends Ether. In this function, the sending method has no gas limit. Therefore, the contract is vulnerable.

```
function sell() {
    uint property = balances[msg.sender];
    uint finalValue = property - fee;
    msg.sender.call.value(finalValue)();
    balances[msg.sender] = 0;
  }
```

Figure 4.47. Call Function With No Data

4.1.0.48. Storage-Related Loop. Accessing a state variable uses more gas than accessing a local variable. Therefore, storage-related loop that uses a state variable may cause extra gas consumption. Figure 4.48 shows a simple sorting function. In this function, array myArr is a storage variable. Therefore, this loop is storage related and is not gas efficient.

4.1.0.49. Constant-Outcome Loop. Constant-outcome loops should be avoided. They may cause extra gas consumption and code smells. Figure 4.49 shows a function that has a loop always counting from 10 to 20.

```
contract C {
    uint counter;
    uint64[] arrToSort;
    uint64 arrayLenght;
    function SortingFunction() returns (uint64 [] )
        for(count =0; count < arrayLength; count++)
        {
            for(uint secondCount =count+1; secondCount <
               arrayLength; secondCount++)
            {
                if(arrToSort[count] < arrToSort[secondCount])
                {
                    uint64 tempValue= arrToSort[secondCount];
                    arrToSort[secondCount] = arrToSort[count];
                    arrToSort[count] = tempValue;
                }
            }
        }
        return arrToSort;
    }
}
```

Figure 4.48. Storage Related Loop

```
function calculate() returns(uint) {
    uint counter = 0;
    for(uint count = 10; count < 20; count++)
    {
        counter += x;
    }
    return counter;
}
```

Figure 4.49. Constant-Outcome Loop

4.1.0.50. Loop Fusion. Loop fusion may cause extra gas consumption. However, two or more loops can be written in the same loop to decrease gas consumption. In Figure 4.50, two loops can be combined as one loop due to the cycle count.

```
function Foo(uint length) {
    uint firstCounter = 0;
    uint secondCounter = 0;
    for(uint loopCounterFirst = 5; loopCounterFirst < length;
        loopCounterFirst++)
    {
        firstCounter += loopCounterFirst;
    }
    for(uint loopCounterSecond = 5; loopCounterSecond < length;
        loopCounterFirst++)
    {
        secondCounter += loopCounterSecond;
    }
  }
```

Figure 4.50. Loop Fusion

4.1.0.51. Mortal Pattern. A mortal pattern provides a function that is dependent on an address can destroy a contract. However, it is undesirable for a contract to be dependent on a specific user. Figure 4.51 shows an example of a mortal pattern.

```
function destroy() {
if (msg.sender == contractOwner)
        suicide(contractOwner);
}
```

Figure 4.51. Mortal Pattern

4.1.0.52. Multiple Return Values. Choosing struct rather than multiple return values in private or internal functions increases code readability. Therefore, we suggest using struct instead of several parameters. Figure 4.52 shows an example of multiple return values.

```
contract multiple_return {
    function foo() internal returns(uint x, address y, unit z) {
        x = 1;
        y = msg.sender;
        z = msg.value;
    }
}
```

Figure 4.52. Multiple Return Values

4.1.0.53. Rejector Pattern. Rejector is a pattern to automatically reject all Ethers to be sent. For example, when a rejector contract is called, the fallback function will work, and it will throw all transactions. Figure 4.53 shows an example of a rejector pattern.

```
contract Rejector {
    function() { throw; }
   }
```

Figure 4.53. Rejector Pattern

4.1.0.54. Selfdestruct Usage. When a contract is no longer wanted, suicide or selfdestruct functions can be used to kill it. Even though we suggest using the selfdestruct function instead of the suicide function, it is necessary to pay attention to the usage, as this function will render the contract disabled. Moreover, the selfdestruct function should not be accessed by any user. Otherwise, it has the potential to cause problems. Figure 4.54 shows an example of a selfdestruct usage.

```
function kill() {
        selfdestruct(owner);
}
```

Figure 4.54. Selfdestruct Usage

4.2. SA-Solidity Working Process

Figure 4.55 shows an overview of SA-Solidity with three modules: PARSER, XML GENERATOR, and RULE CHECKER. To start, SA-Solidity takes the source code of a smart contract as input. The PARSER module converts these contracts to JSON(JavaScript Object Notation) parse tree using Solidity Antlr parser [37].

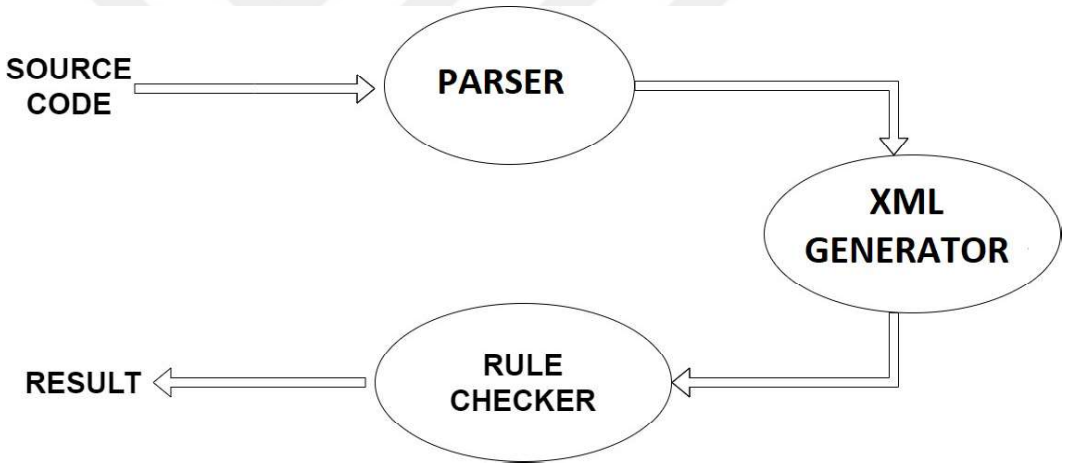

Figure 4.55. SA-Solidity Overview

Then, the XML GENERATOR module converts the parse tree to an XML tree. The purpose of this conversion process is to run XPath on XML, which allows SA-Solidity to create flexible queries. Finally, the RULE CHECKER module queries the XML trees with XPath queries to identify bugs and reports the results to the user.

We explained how SA-Solidity works with the locked money example in Figure 4.10. It first parses this source code and generates the JSON object in Figure 4.56. Then, it generates the XML tree as in Figure 4.57. Lastly, it queries this XML tree with the XPath queries in Figure 4.58 to decide if the source code contains the locked money bug. Since this example indeed contains a locked money bug, SA-Solidity correctly reports the associated file as buggy.

```
"type": "ContractDefinition",
      "name": "Foo",
      "baseContracts": [],
      "subNodes": [
        {
          "type": "FunctionDefinition",
          "name": "deposit",
          "parameters": [],
          "returnParameters": null,
          "body": {
            "type": "Block",
            "statements":
              {
              ...
              },
          "visibility": "default",
          "modifiers": [],
          "isConstructor": false,
          "stateMutability": "payable",
          "loc": {
            ...
          },
        }
      ]
```

Figure 4.56. JSON Parse Tree of Part of a Contract

```
<type>ContractDefinition</type>
<name>Foo</name>
<subNodes>
<type>FunctionDefinition</type>
        <name>deposit</name>
        <returnParameters />
        <body>
                <type>Block</type>
                <statements>...</statements>
        </body>
        <visibility>default</visibility>
        <isConstructor>false</isConstructor>
        <stateMutability>payable</stateMutability>
        <loc>...</loc>
</subNodes>
```

Figure 4.57. XML View of Part of a Contract

```
payable_functions = .//stateMutability[text() = 'payable';
send_usages = .//type[text() = 'FunctionCall']/following-sibling::
    expression/type[text() =  'MemberAccess']/following-sibling::
    memberName[text() = 'send']/parent::*
callvalue_usages = .//memberName[text() = 'call']/parent::*/
    following-sibling::memberName[text() = 'value']
transfer_usages = .//memberName[text()='transfer']
```

Figure 4.58. XPath Queries to Check Locker Ether

4.3. MLBP-Solidity Working Process

Figure 4.59 shows an overview of MLBP-Solidity, a machine learning model to predict specific bugs in Solidity source codes. It is trained using SmartEmbed's Solidity source code dataset. To label the data, we analyzed three different static analysis tools: SA-Solidity, SmartCheck, and Securify. Additionally, the source code metrics were collected by Solmet. After this process, the labeled data and source code metrics were merged. Thus, we obtained consolidated data processed with the different tools. Following this, different classifiers were used on the data to train the model. As a result, MLBP-Solidity ca decide whether Solidity source code is buggy or not.

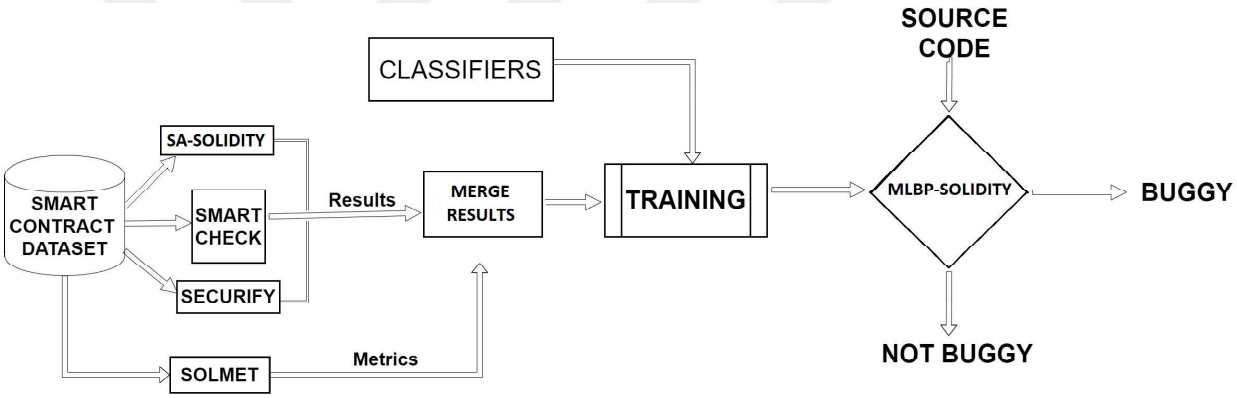

Figure 4.59. MLBP-Solidity Overview

Table 4.2 shows the list of metrics we can collect using SolMet. Most are from Chidamber-Kemerer since Solidity is an object-oriented language.

Seven different bug types were selected to be detected by MLBP-Solidity: Locked Money (A), Reentrancy (B), Missing Input Validation (C), Unrestricted Write (D), Revert Usage (E), Transaction Ordering Dependency (F), Unhandled Exception (G).

Table 4.2. Bug prediction metrics of Solmet

Name	**Description**
SLOC	number of source code lines
LLOC	number of logical code lines
CLOC	number of comment lines
NF	number of functions
McCC	McCabe's cyclomatic complexity
WMC	weighted sum of McCC over the functions
NL	the deepest nesting level
NLE	nesting level else-if
NUMPAR	number of parameters
NOS	number of statements
DIT	depth of inheritance tree
NOA	number of ancestors
NOD	number of descendants
CBO	coupling between object classes
NA	number of attributes (i.e states)
NOI	number of outgoing invocations
Avg. McCC	average McCC
Avg. NL	average NL
Avg. NLE	average NLE
Avg. NUMPAR	average NUMPAR
Avg. NOS	average NOS
Avg. NOI	average NOI

For every bug type, we picked the seven most predictive metrics using random forest. Table 4.3 shows all the smart contract bug types and their associated bug prediction metrics. According to this table, the NLE and NE metrics predicted four out of the seven bug types. These metrics were associated with the complexity of the source code. We argue that as the source code complexity increases, the smart contract becomes intuitively more error-prone, which is why the NLE and NE metrics have more predictive power than the other metrics.

Table 4.3. Smart contract bugs and associated bug prediction metrics

Bug Types	**Metrics**
A. Locked Money	NF, WMC, CBO, NA, Avg. McCC, Avg. NUMPAR, Avg. Nos
B. Reentrancy	NL, NLE, NUMPAR, NOS, NA, Avg. NUMPAR
C. Missing Input Validation	LLOC, NF, WMC, NUMPAR, Avg. MCC, Avg. NUMPAR, Avg. NOS
D. Unrestricted Write	LLOC, NF, WMC, Numpar, NOS, NOI, Avg. NOI
E. Revert Usage	CLOC, NL, NLE, NA, NOI, Avg. NL, Avg. NLE
F. Transaction Ordering Dependency	NL, NLE, NUMPAR, DIT, NOA, CBO, NA, Avg. NUMPAR
G. Unhandled Exception	NL, NLE, DIT, NOA, NOD, Avg. NLE, Avg. NUMPAR

The training phase of MLBP-Solidity uses the known synthetic minority oversampling technique (SMOTE) [38] to eliminate imbalances in the training data; MLBP-Solidity uses the imbalanced-learn [39] package as the implementation of SMOTE.

Moreover, MLBP-Solidity trains Decision Tree, Random Forest, Multi-Layer Perceptron, Naive Bayes, Support Vector Machines, K-Nearest Neighbour, Stochastic Gradient Descent, and Gradient Descent as classifiers [40]. These classifiers are available in the Scikit-learn Python library [41].

5. EXPERIMENTS AND RESULTS

In this chapter, we discuss the experimental results of the tools.

5.1. SA-Solidity Results

SmartEmbed's dataset [42] was used to measure the performance of SA-Solidity. Table 5.1 shows the number of buggy files reported by SA-Solidity.

As stated in Table 5.1, none of the error types were found in all the files. Rule Default Visibility Level is the most common error in smart contracts and it was detected in 22,189 contracts. This means that more than 97% of these contracts contained the error. Since Unnecessary Return Statement and Return Value of Transfer and TransferFrom rules were found in 21962 and 18,322, respectively, of the contracts, they were determined as the second and third most common errors in the smart contract dataset. Only three of the errors, i.e., Default Visibility Level, Unnecessary Return Statement, and Return Value of Transfer and TransferFrom were detected in 75% of the dataset. Apart from these, four of the error types, i.e., Integer overflow/underflow, Constant Function, Approve Function Usage, and Unnecessary Bequest were found in more than half of the files in the dataset as well. Five of the error types, i.e., Integer Division, Throw Usage, Timestamp Usage, Multiple Inheritance, and Shadow Variable Usage were found in more than 25% and less than 50% of the smart contracts. Moreover, 41 of the rules were detected in less than 25% of the smart contracts. Two of the rules were detected in none of the files.

Table 5.1. Number of buggy files reported by SA-Solidity for all 22,725 files

Rule Number	**Buggy**	**Not Buggy**	**Rule Number**	**Buggy**	**Not Buggy**
4.1.0.1	18322	4403	4.1.0.28	3427	19298
4.1.0.2	903	21822	4.1.0.29	1842	20883
4.1.0.3	2205	20520	4.1.0.30	561	22164
4.1.0.4	203	22522	4.1.0.31	21962	763
4.1.0.5	15	22710	4.1.0.32	1429	21296
4.1.0.6	14530	8195	4.1.0.33	309	22416
4.1.0.7	0	22725	4.1.0.34	114	22611
4.1.0.8	1	22724	4.1.0.35	5686	17239
4.1.0.9	5289	17436	4.1.0.36	9939	12786
4.1.0.10	2502	20223	4.1.0.37	9990	12735
4.1.0.11	4143	18582	4.1.0.38	164	22561
4.1.0.12	1	22724	4.1.0.39	4854	17871
4.1.0.13	7561	15164	4.1.0.40	16044	6681
4.1.0.14	4334	18391	4.1.0.41	0	22725
4.1.0.15	1614	21111	4.1.0.42	4	22721
4.1.0.16	762	21963	4.1.0.43	6454	16271
4.1.0.17	2937	19788	4.1.0.44	17016	5709
4.1.0.18	158	22567	4.1.0.45	12715	10010
4.1.0.19	224	22501	4.1.0.46	117	22608
4.1.0.20	6638	16087	4.1.0.47	388	22337
4.1.0.21	309	22416	4.1.0.48	4148	18577
4.1.0.22	16897	5828	4.1.0.49	1345	21380
4.1.0.23	3173	19552	4.1.0.50	1660	21065
4.1.0.24	6344	16381	4.1.0.51	116	22609
4.1.0.25	22189	536	4.1.0.52	564	22161
4.1.0.26	16125	6600	4.1.0.53	3510	19015
4.1.0.27	3706	19019	4.1.0.54	170	22555

5.2. MLBP-Solidity Results

5.2.1. Evaluation of Bug Detection Tools For MLBP-Solidity

In our evaluation, we used SmartEmbed's dataset for training MLBP-Solidity. The dataset contained 22,725 files, and every file had multiple smart contracts. We evaluated the results of SA-Solidity and widely used bug detection tools Securify and SmartCheck. There were 8,261 files in dataset that some of the tools failed to terminate. For the sake of fairness, we did not consider these files in our evaluation and took only 14,464 files out of 22,725 into account. For each bug type, Table 5.2 shows the number of buggy files reported by SA-Solidity, Securify, and SmartCheck. The last column shows the number of files in which all tools detected a bug. Securify did not check for bug type E; SmartCheck did not check for bug types B, D, or F. Therefore, we denoted the associated cells in Table 5.2 as N/A for *not applicable*.

Table 5.2. Number of buggy files reported by SA-Solidity, Securify, and SmartCheck for all 14,464 files

		SA-Solidity	Securify	SmartCheck	All Tools
Bug Types	A	1662	8050	1858	642
	B	6291	2246	N/A	1871
	C	10849	7530	5365	966
	D	7936	13592	N/A	7704
	E	1284	N/A	4659	1282
	F	45	5585	N/A	31
	G	1590	1064	988	186

Table 5.2 shows that the largest number in the intersection column is for bug type D. Almost half of the files contained this bug. Morever, even though the least detected bug type was G, the smallest intersection set was in bug type F.

All three bug detection tools reported different files as buggy, and each identified bugs by checking violations of different sets of rules at the source code level. Figure 5.1 shows the number of rule violations checked by the three bug detectors. From this figure, we can conclude that the bug detectors focused on different rule violations; hence, they detected different bugs in the source code. Using all rule violations or all three bug detectors is too complex and time-consuming, which is why we trained MLBP-Solidity to predict bugs where all tools would find a bug. Thus, developers will have a quick solution to find the most probable bugs before conducting a comprehensive analysis with bug detectors.

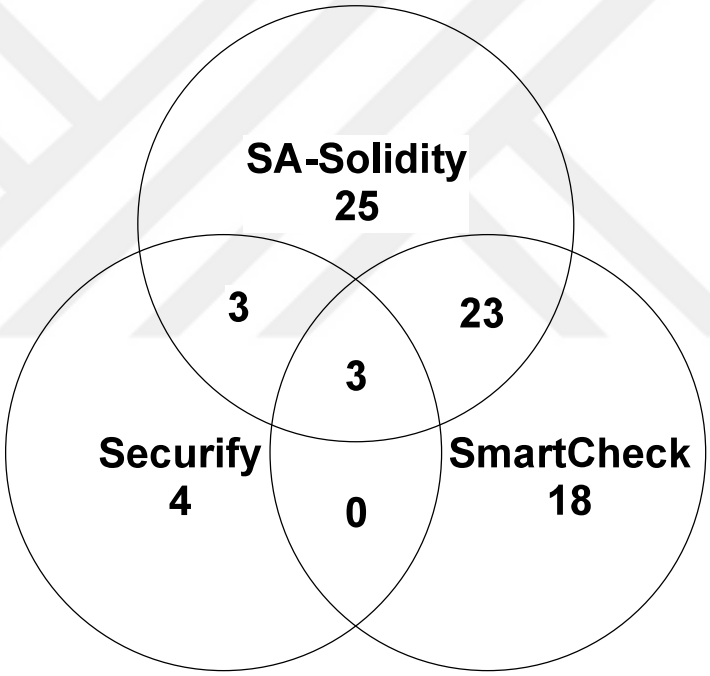

Figure 5.1. Venn Diagram of the Rule Violations Checked by MLBP-Solidity, Securify, and SmartCheck

5.2.2. Evaluation of MLBP-Solidity

In this section, we trained a bug prediction model for MLBP-Solidity. Since we did not compare the bug detection tools, we used all 22,725 files in SmartEmbed's dataset. Table 5.2 shows the number of buggy files reported by all three bug detectors for all files in the dataset. We labeled each file in SmartEmbed's dataset as buggy if all bug detectors that could process that file reported it as buggy.

Otherwise, we labeled the file as not-buggy. We use 5-fold cross validation to divide the labeled dataset into training and test sets. In every fold, we used SMOTE [38] to eliminate imbalances in the training data. We picked the imbalanced-learn [39] package for implementing SMOTE.

For every fold, we trained seven classifiers: Decision Tree, Random Forest, Multi-Layered Perceptron, Naive Bayes, Support Vector Machines, K-Nearest Neighbor, and Stochastic Gradient Descent. We calculated the statistics in Equations (5.1) - (5.5) and took the average of all five folds to measure the classifier performance.

$$\text{Recall(R)} = \frac{\text{tp}}{\text{fn} + \text{tp}} \tag{5.1}$$

$$\text{Precision(P)} = \frac{\text{tp}}{\text{fp} + \text{tp}} \tag{5.2}$$

$$\text{Accuracy(A)} = \frac{\text{tn} + \text{tp}}{\text{tn} + \text{tp} + \text{fn} + \text{fp}} \tag{5.3}$$

$$\text{Positive Prevalence} = \frac{\text{tp} + \text{fp}}{\text{tp} + \text{fp} + \text{tn} + \text{fn}} \tag{5.4}$$

$$\text{F1-Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.5}$$

5.2.3. Experimental Results

Table 5.3 shows the experimental results for MLBP-Solidity with all seven classifiers and all seven bug types (A - G). We evaluated the recall and precision statistics of the classifiers first. Both the recall and precision had to be high to decide a classifier was better than others. According to our results, Random Forest outperformed all the other classifiers by achieving the highest precision score over all bug types. However, it achieved the highest recall for only bug type B. Thus, we evaluated positive prevalence and f1-score statistics to determine the best classifier for MLBP-Solidity. The best classifier had to obtain the lowest positive prevalence and highest f1-score values. According to our results, Random Forest had the lowest Positive Prevalence for the most bug types (A, C, D, E, and F) and clearly outperformed all the other classifiers over all the bug types again. This is why we decided Random Forest is the best classifier for MLBP-Solidity among the seven classifiers. This conclusion was similar to the results Koroglu et. al.'s study [43], wherein Random Forest outperformed the other classifiers researched.

Overall, Table 5.3 shows that MLBP-Solidity achieves $90 - 99\%$ accuracy in predicting bugs that all three bug detectors would report as buggy. Thus, developers/testers should use MLBP-Solidity to quickly eliminate the most common bugs from smart contracts.

Table 5.3. Experimental results

Statistics-Bug Type / Method	Recall							Precision						
	A	B	C	D	E	F	G	A	B	C	D	E	F	G
Decision Tree	0.71	0.89	0.77	0.62	0.62	0.75	0.36	0.52	0.90	0.53	0.41	0.44	0.60	0.14
Random Forest	0.67	**0.91**	0.76	0.61	0.61	0.75	0.49	**0.70**	**0.91**	**0.65**	**0.64**	**0.67**	**0.70**	**0.53**
Multi-Layer Perceptron	**0.80**	0.84	0.50	0.66	0.57	**0.90**	**0.77**	0.16	0.86	0.18	0.04	0.11	0.23	0.01
Naive Bayes	0.33	0.43	0.51	0.39	0.47	0.41	0.56	0.17	0.83	0.21	0.06	0.15	0.37	0.01
Support Vector Machines	0.53	0.66	0.67	0.56	0.55	0.70	0.67	0.15	0.84	0.19	0.04	0.13	0.39	0.06
K-Nearest Neighbor	0.78	0.90	**0.85**	**0.80**	**0.74**	0.85	0.75	0.37	0.89	0.39	0.23	0.25	0.50	0.08
Stochastic Gradient Descent	0.53	0.52	0.68	0.73	0.48	0.71	0.62	0.16	0.85	0.17	0.04	0.17	0.36	0.02
Statistic-Bug Type / Method	**Positive Prevelance**							**F1-Score**						
	A	B	C	D	E	F	G	A	B	C	D	E	F	G
Decision Tree	0.11	0.52	0.09	0.03	0.06	0.16	**0.01**	0.60	0.89	0.63	0.50	0.52	0.67	0.26
Random Forest	**0.08**	0.54	**0.07**	**0.02**	**0.04**	**0.14**	0.02	**0.68**	**0.90**	**0.70**	**0.63**	**0.64**	**0.73**	**0.46**
Multi-Layer Perceptron	0.45	0.52	0.30	0.21	0.21	0.61	0.24	0.26	0.85	0.35	0.07	0.20	0.35	0.01
Naive Bayes	0.17	**0.27**	0.16	0.08	0.13	**0.14**	0.42	0.22	0.57	0.29	0.10	0.23	0.39	0.01
Support Vector Machines	0.30	0.43	0.23	0.16	0.19	0.23	0.26	0.24	0.75	0.30	0.07	0.20	0.50	0.01
K-Nearest Neighbor	0.18	0.53	0.14	0.15	0.12	0.22	0.01	0.50	**0.90**	0.53	0.28	0.38	0.63	0.14
Stochastic Gradient Descent	0.34	0.33	0.26	0.16	0.13	0.26	0.06	0.22	0.63	0.27	0.10	0.25	0.47	0.04

Statistic-Bug Type / Method	Accuracy						
	A	B	C	D	E	F	G
Decision Tree	0.91	0.89	0.94	0.97	0.94	0.90	**0.99**
Random Forest	**0.94**	**0.90**	**0.95**	**0.98**	**0.96**	**0.93**	**0.99**
Multi-Layer Perceptron	0.60	0.84	0.69	0.78	0.79	0.49	0.76
Naive Bayes	0.79	0.65	0.83	0.91	0.86	0.83	0.58
Support Vector Machines	0.70	0.75	0.79	0.83	0.80	0.82	0.73
K-Nearest Neighbor	0.86	0.89	0.90	0.95	0.89	0.87	0.98
Stochastic Gradient Descent	0.66	0.69	0.75	0.84	0.87	0.80	0.93

5.3. Discussion

The bug detectors mentioned in this thesis identify different bugs because their main concerns are different and it is too time-consuming to check for all bugs simultaneously. Securify focuses on bugs that can be checked for with formal analysis techniques. Its main concern is vitally important security issues. Meanwhile, SmartCheck focuses on gas usage problems and ERC-20 standards and their effects on different versions of Solidity language. However, SA-Solidity focuses on security issues, gas usage problems, up-to-date implementation features of Solidity language, object-oriented programming rules, and positive patterns.

In this study, we picked the bug types checked by at least two bug detectors out of the three under evaluation. Note that Securify labels its overflow/underflow findings as missing input validation, but we labeled these findings as missing input validation bugs for the sake of consistency.

All bug detectors under evaluation produce false positives. For the Locked Money bug, Securify only checks if the money can be locked. However, sometimes the user is not allowed to send any money. SmartCheck and SA-Solidity check if the user can also send the money, so they do not produce this particular false positive. Regarding the Reentrancy bug, SmartCheck does not control it, while Securify checks whether this situation may occur. However, SA-Solidity checks whether the state-changing order is correct. For the Missing Input Validation bug, SA-Solidity and SmartCheck suggest using the SafeMath library, whereas Securify only labels the multiplication process as buggy. Meanwhile, SmartCheck and SA-Solidity checks for all kinds of arithmetic operations. Concerning the Restricted Write problem, SmartCheck does not control it, but SA-Solidity checks if a state variable can be accessed and assigned from a parameter in any kind of function. However, Securify uses the formal analysis technique to determine whether state variable is reachable from the parameter. For the Revert Usage problem, Securify does not check for since it is an unharmful problem; SmartCheck controls revert usage, whereas SA-Solidity checks revert usage in if blocks. For the Transaction Ordering Dependency bug, SmartCheck does not check it, whereas SA-Solidity misses many true positives. Securify's output for this bug is more accurate. For the Unhandled Exception bug, Securify rules out cases wherein the exception is unreachable via symbolic execution, while SA-Solidity and SmartCheck do not have this capability. Therefore, for this bug, SA-Solidity and SmartCheck produce false positives where Securify does not. We were able to reduce the false positives in MLBP-Solidity by predicting the bugs reported by all tools.

Unlike Efendioglu et. al.'s previous work [44], we could not find the bug-fix history. Therefore, we could not collect any process metrics for bug prediction. Adding process metrics has the possibility of increasing the bug prediction accuracy. However, MLBP-Solidity has the advantage of being applicable to smart contracts with no version history. Thus, testers/developers can immediately use it on the first version of the smart contract under test.

6. CONCLUSION

In this thesis, we examined Ethereum smart contract bugs and developed SA-Solidity and MLBP-Solidity to detect them. In the beginning, we described blockchain systems and EVM. Then, we provided a brief background on Solidity language. Next, we discussed previous works on smart contract bug checking and demonstrated our knowledge base on Solidity smart contract bugs. Thus, we showed that smart contracts are vulnerable to attacked by cyber-pirates and proposed our methodology to find these bugs. Our first tool, SA-Solidity, uses static analysis techniques to find bugs, and we proved that it can check more rules than known tools Securify and SmartCheck. We also showed that bug detectors focus on different violations. Our second tool, MLBP-Solidity, uses machine learning algorithms to detect seven different bugs in smart contracts using SA-Solidity, Securify, and SmartCheck's results in SmartEmbed's experimental database as ground truth. It is a practical tool for developers/testers as the current state-of-the-art bug detectors cannot check for all bugs and may produce false positives. The tool test results demonstrated that MLBP-Solidity accomplishes 90-99% accuracy. The kind of predicted bug affects the percentage. In future work, based on the new developments in Solidity language, we plan to expand the rule set and data metrics for our bug detector, SA-Solidity. We will also evaluate ensemble classifiers such as majority voting between Random Forest, Multi-Layer Perceptron, and K-Nearest Neighbor.

REFERENCES

1. Sultan, K., U. Ruhi and R. Lakhani, "Conceptualizing Blockchains: Characteristics Applications", *ArXiv*, Vol. abs/1806.03693, 2018.

2. CoinMarketCap, *Cryptocurrency Market Capitalizations*, 2019, `https://coinmarketcap.com`, accessed in January 2019.

3. Antonopoulos, A. M. and G. Wood, *Mastering Ethereum: Building Smart Contracts and Dapps*, O'reilly Media, 2018.

4. Buterin, V., "A next-generation smart contract and decentralized application platform", *White Paper*, Vol. 3, No. 37, 2014.

5. Norman, A. T., *Blockchain Technology Explained: The Ultimate Beginners Guide About Blockchain Wallet, Mining, Bitcoin, Ethereum, Litecoin, Zcash, Monero, Ripple, Dash, IOTA And Smart Contracts*, CreateSpace Independent Publishing Platform, 2017.

6. DAO, *DAO Contract*, 2018, `https://bit.ly/2x8LNYB`, accessed in December 2019.

7. Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl and M. A. Vouk, "On the value of static analysis for fault detection in software", *IEEE Transactions on Software Engineering*, Vol. 32, No. 4, pp. 240–253, April 2006.

8. Ball, T., "The Concept of Dynamic Analysis", O. Nierstrasz and M. Lemoine (Editors), *Software Engineering ESEC/FSE '99*, pp. 216–234, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

9. Nakamoto, S., *Bitcoin: A Peer-to-Peer Electronic Cash System*, Tech. rep., Manubot, 2019.

10. Buterin, V., *Ethereum White Paper*, 2013, http://ethereum.org, accessed in January 2021.

11. Wood, G., "Ethereum: A Secure Decentralised Generalised Transaction Ledger", *Ethereum Project Yellow Paper*, Vol. 151, No. 2014, pp. 1–32, 2014.

12. Dannen, C., *Introducing Ethereum and Solidity*, Vol. 1, Springer, 2017.

13. Ethereum Foundation, *Accounts*, 2016, http://bit.ly/3roDRg2, accessed in December 2019.

14. Vitaki, A., *Ranking Ethereum Smart Contracts*, 2018, http://bit.ly/2Mzu3Rm, accessed in June 2019.

15. Ethereum Foundation, *Solidity*, 2016, http://bit.ly/2YRpswn, accessed in January 2021.

16. Wan, Z., D. Lo, X. Xia and L. Cai, "Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study", *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 413–424, 2017.

17. Atzei, N., M. Bartoletti and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts SoK", *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, p. 164–186, Springer-Verlag, Berlin, Heidelberg, 2017, https://doi.org/10.1007/978-3-662-54455-6$_8$.

18. Luu, L., D.-H. Chu, H. Olickel, P. Saxena and A. Hobor, "Making Smart Contracts Smarter", *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pp. 254–269, ACM, New York, NY, USA, 2016, http://doi.acm.org/10.1145/2976749.2978309.

19. Mueller, B., J. Honig and N. Parasaram, *Introducing Mythril: A framework for bug hunting on the Ethereum blockchain*, 2020, `https://github.com/ConsenSys/mythril`, accessed in January 2021.

20. Sen, K., "Concolic Testing", *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pp. 571–572, ACM, New York, NY, USA, 2007, `http://doi.acm.org/10.1145/1321631.1321746`.

21. Fontein, R., "Comparison of static analysis tooling for smart contracts on the evm", *28th Twente Student Conference on IT*, 2018.

22. Nikolić, I., A. Kolluri, I. Sergey, P. Saxena and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale", *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 653–663, 2018.

23. Brent, L., A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz and B. Scholz, "Vandal: A Scalable Security Analysis Framework for Smart Contracts. CoRR abs/1802.08660 (2018)", *arXiv preprint arXiv:1809.03981*, 2018.

24. Tsankov, P., A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts", *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pp. 67–82, ACM, New York, NY, USA, 2018, `http://doi.acm.org/10.1145/3243734.3243780`.

25. Tikhomirov, S., E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts", *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, p. 9–16, Association for Computing Machinery, New York, NY, USA, 2018, `https://doi.org/10.1145/3194113.3194115`.

26. Gao, Z., V. Jayasundara, L. Jiang, X. Xia, D. Lo and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding", *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 394–397, IEEE, 2019.

27. Huang, T. H., "Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks", *CoRR*, Vol. abs/1807.01868, 2018, `http://arxiv.org/abs/1807.01868`.

28. Chen, T., X. Li, X. Luo and X. Zhang, "Under-Optimized Smart Contracts Devour Your Money", *CoRR*, Vol. abs/1703.03994, 2017, `http://arxiv.org/abs/1703.03994`.

29. Parizi, R. M., A. Dehghantanha, K.-K. R. Choo and A. Singh, "Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains", *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, p. 103–113, IBM Corp., USA, 2018.

30. Mavridou, A. and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach", *Financial Cryptography and Data Security*, pp. 523–540, Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

31. Hegedüs, P., "Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts", *Technologies*, Vol. 7, p. 6, 01 2019.

32. Ethereum Foundation, *ERC-20 Token Standard*, 2020, `http://bit.ly/36QPO68`, accessed in February 2020.

33. OpenZeppelin, *OpenZeppelin Library*, 2017, `https://cutt.ly/FjcyhYY`, accessed in February 2020.

34. OpenZeppelin, *Safe Math Library*, 2017, `http://bit.ly/3rtcUYu`, accessed in February 2020.

35. TheWeakestHodler, *TheWeakestHodler Contract*, 2019, `https://bit.ly/3juFztA`, accessed in January 2021.

36. ConsenSys, *EIP20 Contract*, 2018, `http://bit.ly/2N4UChr`, accessed in February 2020.

37. Bond, F., *Antlr Parser*, 2017, `http://bit.ly/3tuW5hL`, accessed in December 2019.

38. Bowyer, K. W., N. V. Chawla, L. O. Hall and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique", *CoRR*, Vol. abs/1106.1813, 2011, `http://arxiv.org/abs/1106.1813`.

39. Lemaitre, G., F. Nogueira and C. K. Aridas, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning", *CoRR*, Vol. abs/1609.06570, 2016, `http://arxiv.org/abs/1609.06570`.

40. Alpaydin, E., *Introduction to Machine Learning*, MIT Press, Cambridge, MA, 3 edn., 2014.

41. Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, "Scikit-learn: Machine Learning in Python ", *Journal of Machine Learning Research*, Vol. 12, pp. 2825–2830, 2011.

42. Gao, Z., L. Jiang, X. Xia, D. Lo and J. Grundy, *SmartEmbed*, 2019, `https://cutt.ly/HgSmjkd`, accessed in February 2020.

43. Koroglu, Y., A. Sen, D. Kutluay, A. Bayraktar, Y. Tosun, M. Cinar and H. Kaya, "Defect Prediction on a Legacy Industrial Software: A Case Study on Software with Few Defects", *2016 IEEE/ACM 4th International Workshop on Conducting Empirical Studies in Industry (CESI)*, pp. 14–20, May 2016.

44. Efendioglu, M., A. Sen and Y. Koroglu, "Bug Prediction of SystemC Models Using Machine Learning", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 38, No. 3, pp. 419–429, March 2019.