

# A Static Analysis Approach for Ethereum Smart Contracts

by

**VAISHNAVI  
202111051**

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY

in

INFORMATION AND COMMUNICATION TECHNOLOGY

to

**DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY**

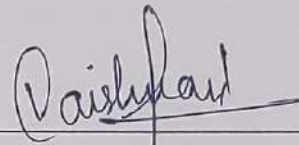


June, 2023

## Declaration

I hereby declare that

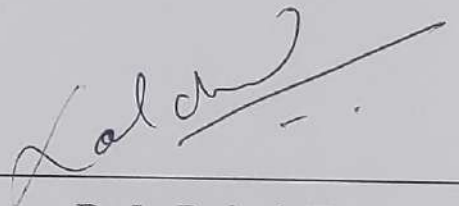
- i) The thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,
- ii) due acknowledgment has been made in the text to all the reference material used.



Vaishnavi

## Certificate

This is to certify that the thesis work entitled A Static Analysis Approach for Ethereum Smart Contracts has been carried out by Vaishnavi for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my/our supervision.



Dr. JayPrakash TL  
Thesis Supervisor

# Acknowledgments

First and foremost, I want to express my sincere gratitude to my thesis guide Dr. Prof. JayPrakash Lalchandani, who oversaw my thesis. This study endeavor has been greatly influenced by their continuous support, direction, and enormous competence. They have given me so much flexibility to pursue new interests thanks to their persistence, support, and patience.

I also want to express my appreciation to my friends and co-batchmates who supported me during this difficult journey. Their unfailing encouragement, words of support, and readiness to assist me anytime I needed it.

I also want to thank my family for all of their help and support. Their consistent confidence in my talents and ongoing support have been the pillars of strength that have helped me advance, even in the most difficult situations.

# Contents

<b>Abstract</b>	<b>vi</b>
<b>List of Principal Symbols and Acronyms</b>	<b>viii</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Figures</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Smart Contracts Security and Historical Losses. . . . .	1
1.2 Known Vulnerabilities and Tools . . . . .	3
1.3 Motivation . . . . .	4
1.4 Objective . . . . .	4
1.5 Workflow of Thesis . . . . .	5
1.6 Thesis Outline . . . . .	6
1.7 Chapter Summary . . . . .	6
<b>2 Background</b>	<b>7</b>
2.1 Vulnerability Walkthrough . . . . .	7
2.1.1 Arithmetic . . . . .	7
2.1.2 Front Running . . . . .	8
2.1.3 Reentrancy . . . . .	9
2.1.4 Time Manipulation . . . . .	11
2.1.5 Unchecked Low-Level Calls . . . . .	12
2.2 Preliminaries . . . . .	13
2.2.1 Rattle . . . . .	13
2.2.2 Satisfiability Modulo Theories (SMT Z3 Solver) . . . . .	14
2.3 Static Analysis Tools . . . . .	15
2.3.1 Symbolic Execution . . . . .	16
2.4 Dynamic Analysis Tools . . . . .	19
2.5 Comparison Between Static and Dynamic Analysis . . . . .	21

2.6	Chapter Summary . . . . .	23
<b>3</b>	<b>Literature Survey</b>	<b>24</b>
3.1	Blockchain, Ethereum, and Bitcoin . . . . .	24
3.2	Application of Blockchain . . . . .	25
3.3	Detecting Vulnerability and Security Analysis of Smart Contracts .	25
3.4	The DAO Attack and Vulnerability . . . . .	26
3.5	Static Analysis Tools . . . . .	27
3.6	Dynamic Analysis Tools . . . . .	28
3.7	Modules . . . . .	29
3.8	SmartBugs . . . . .	29
3.9	Chapter Summary . . . . .	30
<b>4</b>	<b>Proposed Method</b>	<b>31</b>
4.1	Workflow . . . . .	31
4.1.1	Algorithm for Master File . . . . .	33
4.1.2	Algorithm for Arithmetic Vulnerability . . . . .	36
4.1.3	Algorithm for Front Running(Time Order Dependency) Vul- nerability . . . . .	40
4.1.4	Algorithm for Reentrancy Vulnerability . . . . .	44
4.1.5	Algorithm for Time Manipulation Vulnerability . . . . .	47
4.1.6	Algorithm for Unchecked Low Level Calls Vulnerability . .	50
4.2	Overall Complexity . . . . .	53
4.2.1	How Complexities Can be Improved Overall . . . . .	54
4.3	Association of Files . . . . .	54
4.4	Dependencies . . . . .	55
4.5	Chapter Summary . . . . .	56
<b>5</b>	<b>Results</b>	<b>57</b>
5.1	Methodology . . . . .	57
5.2	Datasets . . . . .	58
5.2.1	SB curated Dataset Specifications . . . . .	58
5.2.2	SB wild Dataset Specifications . . . . .	58
5.3	Experimental Setup . . . . .	59
5.4	Experimental Results . . . . .	60
5.4.1	Effectiveness of Tools . . . . .	62
5.4.2	Performance of Tools . . . . .	65
5.5	Chapter Summary . . . . .	66

<b>6 Conclusion and Future Work</b>	<b>67</b>
6.1 Conclusion . . . . .	67
6.2 Future Work . . . . .	67
<b>References</b>	<b>69</b>

# Abstract

The Ethereum blockchain market has grown in prominence in recent years, enabling the daily trading of billions of dollars. Smart contracts are programs that are written in Solidity language and are executed on the Ethereum blockchain. However, the execution of smart contracts handling ether currencies has led to issues and disputes since 2016. This study focuses on the vulnerabilities of smart contracts on Ethereum. There are many tools available for detecting vulnerabilities in smart contracts. However, there is still room for research in this area, here in our study we have implemented five modules that are Arithmetic, Front-Running, Re-entrancy, Time Manipulation, and Unchecked Low-Level Calls, that are intended to find security flaws listed under the DASP10 framework. We are able to examine contract behaviour and find potential security flaws by using symbolic execution. Performance and accuracy are the two metrics taken care of throughout the study. This study tries to improve smart contract security procedures by detecting these vulnerabilities earlier before deploying them on the blockchain.

# List of Principal Symbols and Acronyms

AI	Abstract Interface
AT	Arithmetic
CFG	Content Flow Graph
CI	Code Instrumentation
CLI	Command Line Interface
CS	Constraint Solving
CT	Code Transformation
DApp	Decentralized Application
DASP	Decentralized Application Security Project
ERC20	Ethereum Request for Comments 20
ET	Execution Time
ETC	Ethereum Classic
ETC	Execution Time per Contract
ETH	Ethereum
EVM	Ethereum Virtual Machine
FR	Front Running
FT	Fuzz Testing
IR	Intermediate Representation
MBT	Model Based Testing
ML	Machine Learning



pBFT practical Byzantine Fault  
PoS Proof of Stake  
PoW Proof of Work  
RT Reentrancy  
SB Smartbugs  
SE Symbolic Execution  
SMT Satisfiability Modulo Theories  
SSA Single Static Assignment  
SWC Sweatcoin  
TA Taint Analysis  
TM Time Manipulation  
TOD Transaction Order Dependence  
ULL Unchecked Low Level Checks  
XML Extensible Markup Language

# List of Tables

2.1	Classification of Tools . . . . .	22
4.1	Comparison of Various Files . . . . .	54
5.1	Categories of Vulnerability Present in Dataset . . . . .	58
5.2	Vulnerability Categorise Division with the Criteria of Time Budget	62
5.3	Vulnerability Categorise Division with the Criteria Without Time Budget . . . . .	63
5.4	Vulnerability Categorise Division . . . . .	65
5.5	Execution Time of Each Tool for Both Datasets . . . . .	66

# List of Figures

1.1	Real World Example Of Smart Contract Execution . . . . .	2
1.2	Workflow of Thesis . . . . .	5
2.1	Example Contract for Arithmetic Vulnerability . . . . .	8
2.2	Example Contract for Front Running Vulnerability . . . . .	9
2.3	Example Contract for Reentrancy Vulnerability . . . . .	10
2.4	Example Contract for Time Manipulation Vulnerability . . . . .	11
2.5	Example Contract for Unchecked Low Level Calls Vulnerability . .	12
2.6	Register Machine Output . . . . .	13
2.7	Example Problem for Z3 Working . . . . .	14
2.8	Response of Z3 . . . . .	15
2.9	Category-wise Division of Smart Contract Analysis Tools . . . . .	15
4.1	Proposed Architecture . . . . .	32
4.2	Execution of Master File . . . . .	35
4.3	Execution of Arithmetic . . . . .	39
4.4	Execution of Front Running . . . . .	42
4.5	Execution of Reentrancy . . . . .	45
4.6	Execution of Time Manipulation . . . . .	49
4.7	Execution of ULL . . . . .	52
4.8	Flow of Execution . . . . .	55
5.1	Command Line Interface for single Processing . . . . .	59
5.2	Command Line Interface Batch Processing . . . . .	60
5.3	Text File Output . . . . .	61
5.4	Result with the Criteria of Time Budget . . . . .	63
5.5	Results with the Criteria of Execution Till Halt . . . . .	64
5.6	Results with the Criteria of Execution Till Halt . . . . .	65

## CHAPTER 1

# Introduction

Blockchain technology, particularly the Ethereum platform, has received a lot of interest in recent years as a viable option for decentralized and secure transactional systems [42] [43]. The notion of a distributed ledger lies at the heart of this technology, allowing participants to keep a synchronized and immutable record of transactions without the need for middlemen. The concept of blockchain was first introduced in 2008 as the foundation for digital currencies such as Bitcoin [13], Ethereum [35], and many more. These cryptocurrencies use blockchain to deliver safe and transparent transactions without the need for middlemen such as banks or payment processors. Blockchain provides transaction integrity, avoids duplicate spending, and allows user to verify their authenticity. Aside from cryptocurrencies, blockchain technology has multiple potential applications in health-care [26], finance and banking [23], voting systems [14], supply chain management [6] [15] etc.

One of the most important characteristics of blockchain is its decentralized nature. In contrast to typical centralized systems in which a single entity controls the data, blockchain functions on a distributed network of computers known as nodes. Each node keeps a copy of the full blockchain, and any modifications to the data must be approved by all participants. This decentralized structure has various advantages, including better security, transparency, and efficiency.

### 1.1 Smart Contracts Security and Historical Losses.

Smart contracts are programs executed across a decentralized network of nodes and written in a Turing-complete language, most often Solidity.

As shown in Fig 1.1, two users (a seller and a buyer) conduct business using an application that includes a smart contract. The transaction is completed in five

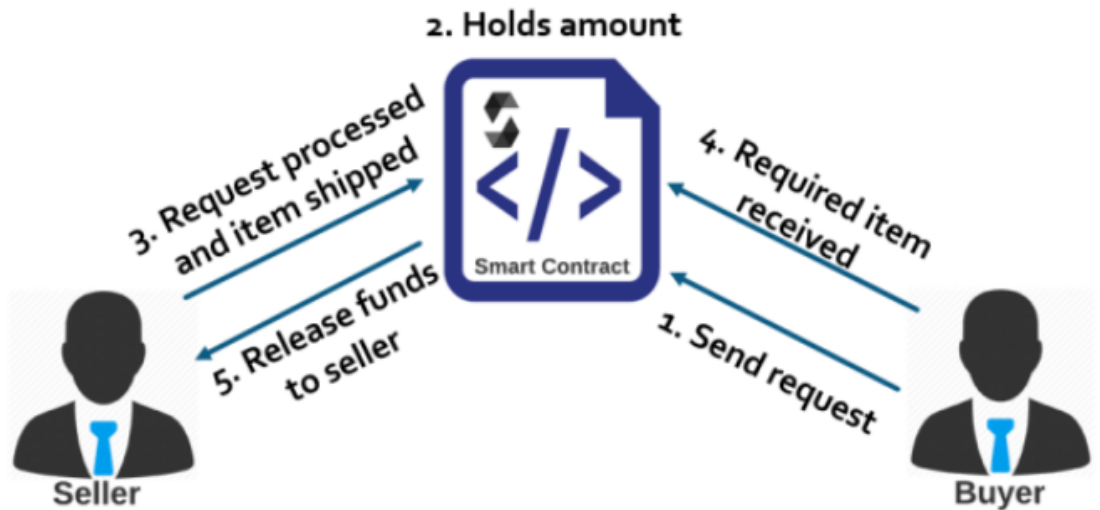


Figure 1.1: Real World Example Of Smart Contract Execution

steps that are- step 1, the buyer delivers the needed number of ethers to the smart contract's address, with the balance held in escrow by the smart contract. In step 2, the smart contract notifies the seller by causing an event that identifies the recipient of the buyer's request. In step three, the seller checks and validates the buyer's request; if it is accurate and there are sufficient ethers to pay for the desired item, the seller will ship the item and send a shipping message to the smart contract. The smart contract is updated with the delivery status at step 4 of the process. In step 5, the smart contract sends the ethers to the seller's account.

However, due to the idiosyncrasies of the EVM [2], writing secure smart contracts is challenging. Unfortunately, as the number of smart contracts grows, security concerns are arising. Smart contract code security issues will almost certainly be developed during code development. A smart contract implemented on the public blockchain is typically exposed in an open setting, leaving it vulnerable to hacking [39] [24]. Furthermore, because smart contracts are immutable and irreversible, we can only observe the cash flowing into the attacker's package and can be powerless to intervene.

Smart contracts are not free from vulnerabilities [25] [31] and Some of them were previously taken advantage of, which cost them money like the DAO attack [41] which was one of the initial attacks that happened in 2016 where the attackers drained approximately one-third of the DAO's money, which totaled more

than \$50 million at the time. This incident caused a hard split in the Ethereum blockchain, leading to the formation of two different cryptocurrencies, ETH and ETC. Another famous attack took place in the same year where an anonymous attacker exploited a weakness in the smart contract of the King of the Ether DApp on the Ethereum network, before the vulnerability was detected and patched, the attacker was able to modify the game's rules and regularly claim prizes, collecting a substantial quantity of ether. Followed there was a parity wallet freeze in 2017, resulting in the freezing of \$160 million worth of ether and these numbers are still rising.

To summarise, smart contract security issues not only result in massive money losses but also undermine the basic trust built on blockchain technologies. As a result, before smart contracts can be deployed, robust security analysis and vulnerability detection methodologies are required in order to prevent possible vulnerabilities and secure users' money and assets.

## **1.2 Known Vulnerabilities and Tools**

In this section, we will discuss the key vulnerabilities that lead to the exploitation of smart contracts, as discussed in section 1.1 a poorly coded smart contract can be hacked by someone sending certain instructions. Smart contracts, while promising decentralized automation, face numerous security challenges due to various factors [44].

Inadequate code review and the complexity of smart contracts often lead to coding errors and logical flaws. Formal verification is underutilized, leaving vulnerabilities undetected. Dependencies on external contracts and libraries can introduce weaknesses in the main contract. Insufficient testing and time constraints further exacerbate vulnerabilities. The evolving threat landscape demands continuous updates to address emerging risks. Additionally, the immutability of deployed contracts prevents easy post-deployment patching. Misuse of fallback mechanisms and security oversights also contribute to smart contract vulnerabilities. Mitigating these issues necessitates secure coding practices, rigorous testing, continuous monitoring, and adoption of formal verification methods. Staying updated with evolving security best practices is essential for developing resilient and secure smart contracts. While providing a precise figure is difficult, it is commonly acknowledged that smart contracts can be vulnerable.

Choosing a static analysis tool for smart contract security is essential to detect vulnerabilities at an early stage by analyzing the contract's source code without execution. This proactive approach minimizes the risk of exploitation by malicious actors. By automating the analysis process, these tools improve efficiency and reduce human errors in vulnerability detection. Moreover, they can uncover complex vulnerabilities that may be challenging to identify through manual code review. Comparing the effectiveness and performance of different tools helps researchers and developers choose the most suitable option for robust smart contract security.

Here, we are mainly focusing only on five vulnerabilities that are Reentrancy, Arithmetic, Front Running, Time Manipulation, and Unchecked Low-Level Calls. Further, these vulnerabilities are examined for our approach along with eight open-source static tools which are Honeybadger, Manticore, Mythril, Osiris, Oyente, Securify, Slither, and Smartcheck. Detailed discussion about the vulnerabilities and tools is covered in Chapter 2.

### **1.3 Motivation**

There has been a dramatic rise in the popularity of smart contracts. Like software programs, smart contracts may also contain bugs that can be exploited by malicious attackers for financial gains. The first smart contract assault occurred in 2016, and after that, these attacks continued. Despite many static analysis tools present in open source, there is still room for improvement in the accuracy of detecting vulnerabilities, also minimizing human error, discovering complex vulnerabilities, addressing emerging threats, etc. The motivation behind this study is to analyze how vulnerabilities affect transactions, how can we detect them, and how well static analysis tools work examining the contracts.

### **1.4 Objective**

The objective of this thesis is to develop and present a static and modular analysis approach based on symbolic execution to uncover vulnerabilities in smart contracts. The proposed approach will address the limitations of existing open-source tools by implementing five distinct modules, each designed to detect specific vulnerabilities commonly found in smart contracts. Additionally, the research will

compare the effectiveness and performance of the developed approach with other state-of-the-art open-source tools.

## 1.5 Workflow of Thesis

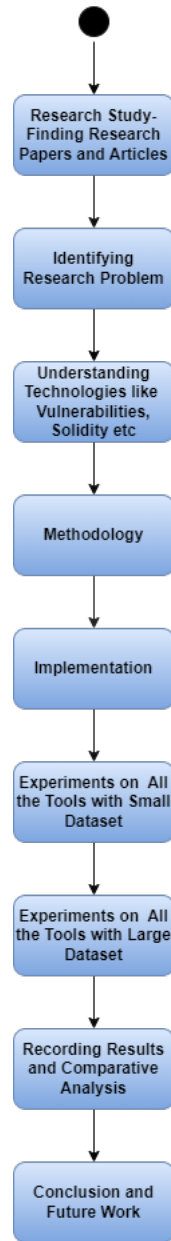


Figure 1.2: Workflow of Thesis



## 1.6 Thesis Outline

The organization of the thesis is as follows:

- Chapter 2 discusses the background of five vulnerabilities we work on. Further, static analysis and dynamic analysis tools are compared. Moving forward, symbolic execution, tools with specific versions, followed by a brief explanation of the vulnerabilities are discussed.
- Chapter 3 represents a comprehensive review of all the articles and papers that we referred to in our study.
- Chapter 4 discusses the proposed method where the workflow of the architecture is explained, along with that the implemented modules complexities are discussed later in the same chapter system dependencies are also given.
- Chapter 5 gives the results, and also describes the methodology used in this study with datasets details and experimental details.
- Chapter 6 concludes the work and presents the future scope of the thesis.

## 1.7 Chapter Summary

In this chapter, the rise of blockchain in different potential fields, how smart contracts are changing the blockchain, and how they can be exploited are discussed. The idea of this chapter was to introduce the known vulnerabilities and the tools we are going to work with. Also, the motivation and objective are discussed. Further to move ahead with the study, the background will be discussed in the next chapter.

## CHAPTER 2

# Background

This chapter discusses about a comprehensive understanding of the vulnerabilities and analysis tools that form the foundation of our research. Here we are doing an empirical study for a detailed exploration of vulnerability walkthrough, analysis tools, and a comparison between static and dynamic analysis techniques. we will also delve into the concept of Rattle and SMT Z3 Solver. These tools have proven to be invaluable in identifying and mitigating vulnerabilities in various systems. We will discuss the features, capabilities, and methodologies employed by these tools, showcasing their effectiveness in vulnerability detection and resolution.

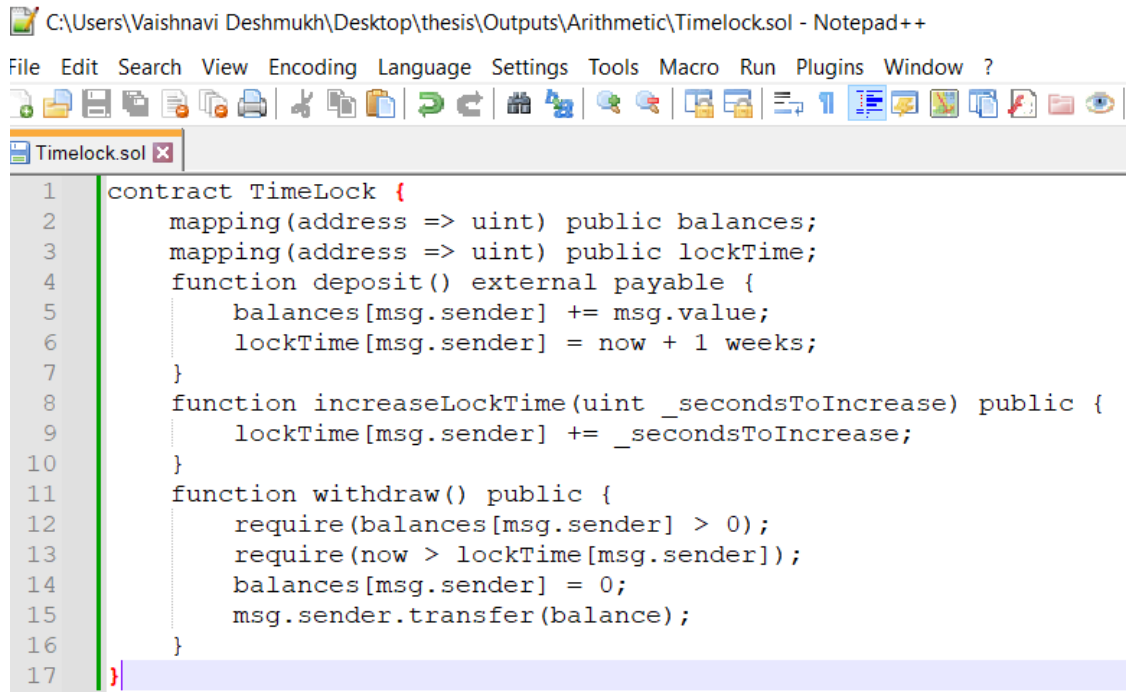
## 2.1 Vulnerability Walkthrough

As part of our current focus on enhancing smart contract security, we prioritize the analysis of five key vulnerabilities, which fall within the DASP Top 10 category.

### 2.1.1 Arithmetic

The integer type in Solidity has a constrained range. During arithmetic operations, variable values may go above the upper or lower bound. The value will warp to the other side of the bound if this happens. The actual value is the real value less the upper bound if the real value exceeds the upper bound. Older versions of Solidity do not issue a warning for this anomaly. Attackers might manipulate particular integer values to cause the smart contract to behave abnormally. Floating points are not yet supported by Solidity. In order to represent floating numbers in smart contracts, integer types must be used. For example, decimal is used to express the number of digits following the decimal point in the design of ERC20 (a token standard). Solidity's division always rounds off to the lower integer [40].

For example, if we consider the contract shown in Fig. 2.1 Users can deposit ether into the contract, and it will be locked there for at least a week. This contract is intended to function as a time vault. However, once deposited, the user may be confident that their ether is locked in safely for at least a week, as this contract implies. The user may choose to prolong the wait time to longer than 1 week. A con-



```
1 contract TimeLock {
2     mapping(address => uint) public balances;
3     mapping(address => uint) public lockTime;
4     function deposit() external payable {
5         balances[msg.sender] += msg.value;
6         lockTime[msg.sender] = now + 1 weeks;
7     }
8     function increaseLockTime(uint _secondsToIncrease) public {
9         lockTime[msg.sender] += _secondsToIncrease;
10    }
11    function withdraw() public {
12        require(balances[msg.sender] > 0);
13        require(now > lockTime[msg.sender]);
14        balances[msg.sender] = 0;
15        msg.sender.transfer(balance);
16    }
17 }
```

Figure 2.1: Example Contract for Arithmetic Vulnerability

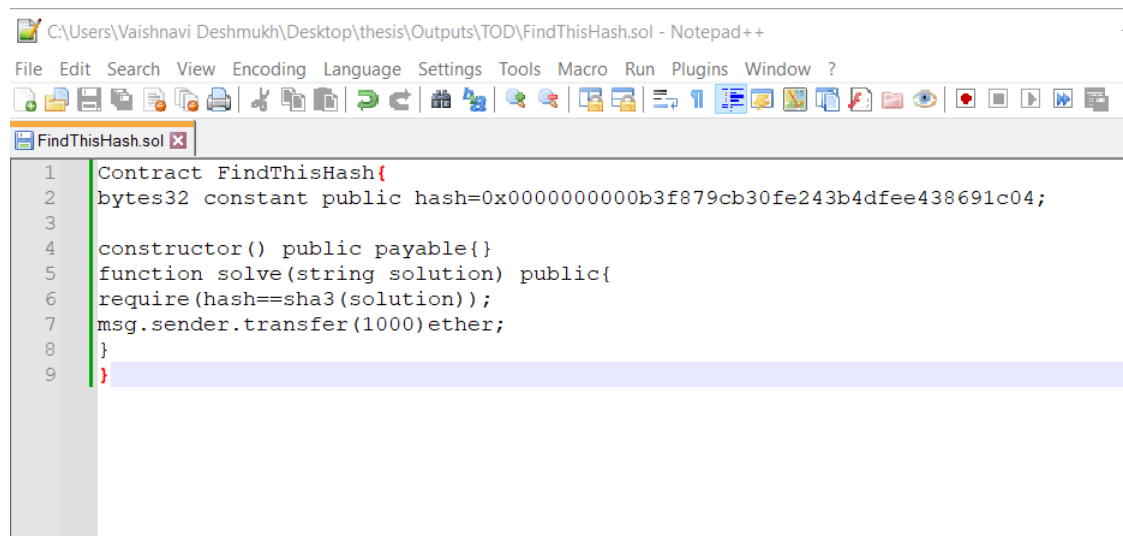
tract like this could come in handy if a user is required to hand over their private key to make sure their ether is unavailable for a brief period of time. However, if a user had entered 100 ether into this contract and given the attacker their keys, the attacker could utilize overflow to extract the ether from the user. The lockTime() for the address for which they now possess the key is a public variable. Thus, the attacker might find it. They could then execute the increaseLockTime() and send the number  $2^{256}$ , userLockTime as input after referring to this value as userLockTime(). The amount would produce an overflow when added to the current userLockTime(), setting lockTime[msg.sender] to 0. In order to get their reward, the attacker might then only call the withdrawal method.

### 2.1.2 Front Running

Transaction Order Dependence (TOD) is another name for this issue, according to article [4]. When two transactions (T1 and T2) are present and each of them invokes the same contract, it occurs. If transaction T2 happens first and results in

state 1, then transaction T1 can happen in state 1 or in state 0. A miner chooses this sequence, but malicious users can profit from it. For instance, if a user submits a solution to a puzzle, a malevolent user may be able to view the answer and then initiate a new transaction using the solution and pay a much higher price (also known as gas) to perform the transaction in order to be mined first and win the prize. Since miners choose how transactions are processed in a given order, there is currently no known mechanism to address this kind of vulnerability.

To understand this vulnerability in detail let us consider a contract shown in Fig 2.2 here an attack on this smart contract can happen in a way where the first user will send the hash solution to the smart contract in a transaction. This transaction will reach the mempool where it will be waiting to be added to the following block. An attacker is keeping an eye on the mempool and waiting for a transaction to obtain the solution. After that, the attacker sends the smart contract with the same value but with a greater Gas value. Hence, the attacker transaction will be the first to be completed, making him the winner.



```
1 Contract FindThisHash{
2   bytes32 constant public hash=0x0000000000b3f879cb30fe243b4dfee438691c04;
3
4   constructor() public payable{}
5   function solve(string solution) public{
6     require(hash==sha3(solution));
7     msg.sender.transfer(1000)ether;
8   }
9 }
```

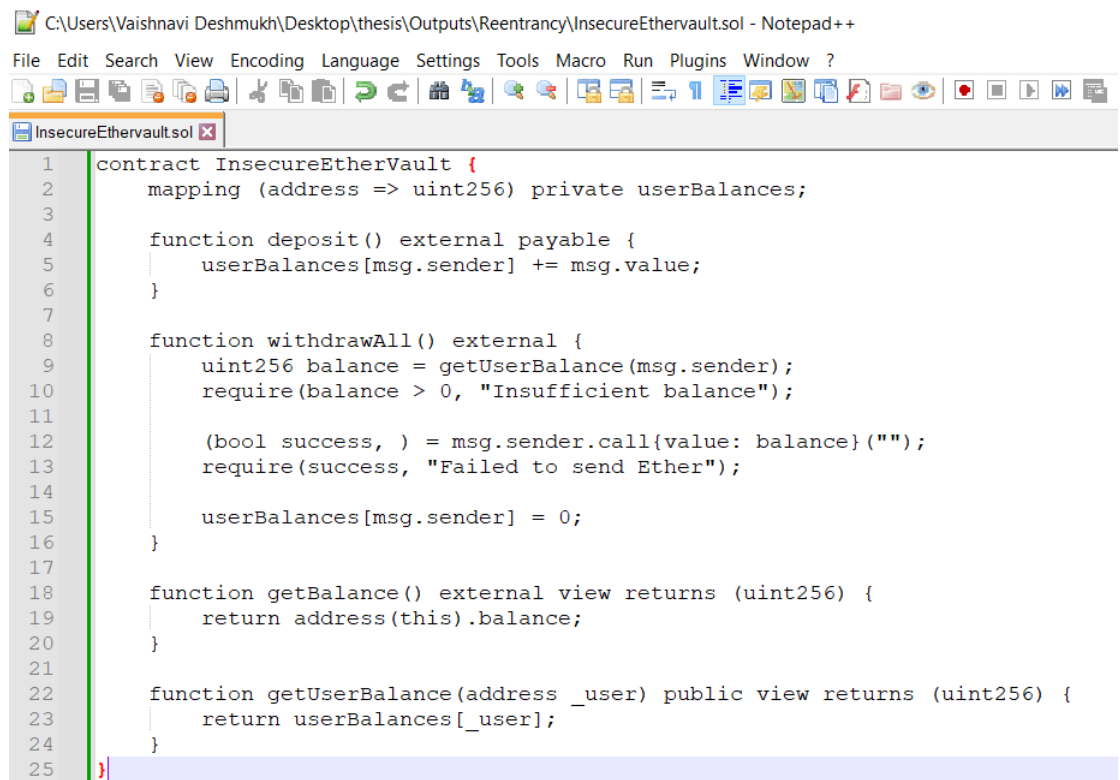
Figure 2.2: Example Contract for Front Running Vulnerability

### 2.1.3 Reentrancy

Reentrancy [7] refers to a circumstance in which contract A contacts contract B, who may call A back and perform A's call again. This condition is caused by Solidity's fallback mechanism. When calls from other contracts fail to discover a matching function, the fallback function is called. When the caller invokes the call function without providing a function signature, the callee's fallback function is

invoked. To re-enter the caller, this function might call the caller's function. In rare cases, this method may induce unexpected and uncontrolled ether transfers.

Reentrancy vulnerability can be explained with an example shown in Fig 2.3, here the smart contract InsecureEtherVault can be attacked as the reentrancy begins in line 10 in the withdrawAll function. The amount of ethers indicated by the balance variable would be transmitted to the user wallet or external contract as soon as the low-level function call is carried out. The attacker can attack the contract which can do the reentrancy by repeatedly invoking the withdrawAll function to drain out all Ethers locked in the InsecureEtherVault contract. Because the call function is called prior to setting the withdrawer's balance to 0 (`userBalances[msg.sender] = 0`), the attack is successful in this case. As a result, the attack contract has the ability to perform the loop calls to the withdrawAll function in the middle of the control flow. The attack contract has the ability to steal all ethers because the withdrawAll function would still keep track of the balance before the modification.



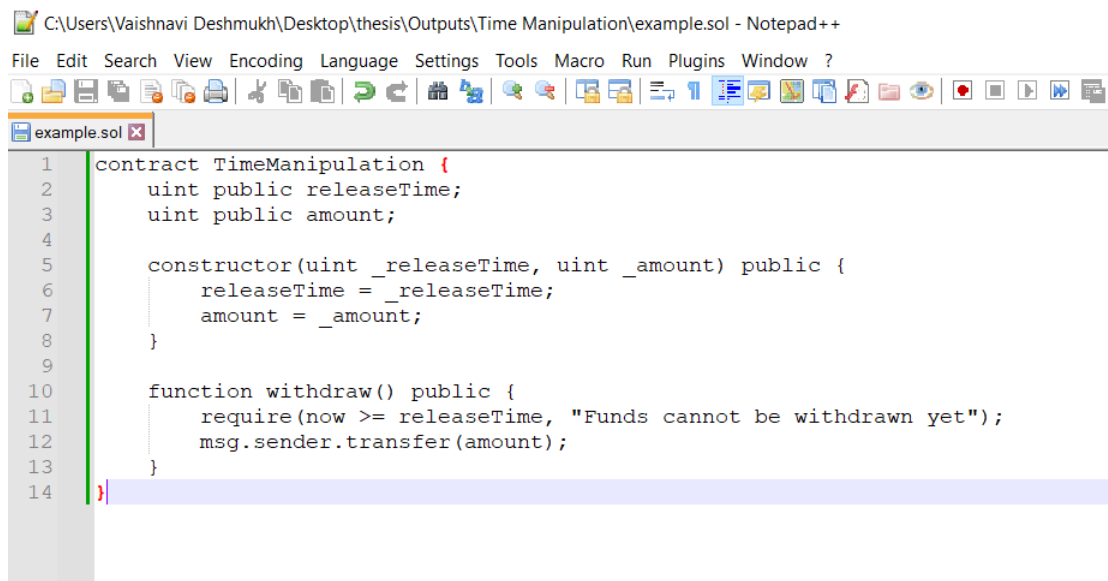
```
1 contract InsecureEtherVault {
2     mapping (address => uint256) private userBalances;
3
4     function deposit() external payable {
5         userBalances[msg.sender] += msg.value;
6     }
7
8     function withdrawAll() external {
9         uint256 balance = getUserBalance(msg.sender);
10        require(balance > 0, "Insufficient balance");
11
12        (bool success, ) = msg.sender.call{value: balance}("");
13        require(success, "Failed to send Ether");
14
15        userBalances[msg.sender] = 0;
16    }
17
18    function getBalance() external view returns (uint256) {
19        return address(this).balance;
20    }
21
22    function getUserBalance(address _user) public view returns (uint256) {
23        return userBalances[_user];
24    }
25 }
```

Figure 2.3: Example Contract for Reentrancy Vulnerability

## 2.1.4 Time Manipulation

The contracts must have up-to-date information, which can be acquired through a variable called a block. Simply put, the variable timestamp reads the data found in the block where the transaction is located. However, because miners hold this information, developers should avoid exploiting it. There are no restrictions on this value in the Ethereum specification (yellow paper), which merely states that it must be higher than the preceding block. Both the Parity and Geth implementations on Ethereum reject timings that are off by 15 seconds. Since the miners control this information, there is currently no known method to mitigate this type of vulnerability. The only solution is for developers to refrain from using blocks with the timestamp prefix.

In Fig 2.4 the TimeManipulation contract in this example accepts two inputs, first the amount which represents the value to be released, and second releaseTime which specifies the timestamp at which funds may be withdrawn. A user can only withdraw money via the withdraw function if the releaseTime is greater than or equal to the current time. However, a spoofing attack could be used by an attacker to provide the impression that more time has passed than it actually has. The attackers can spoof the release time, adds one hour to the contract's releaseTime, and then prematurely withdraw the funds. An attacker who seizes control of the blockchain node that verifies network transactions could launch this attack.



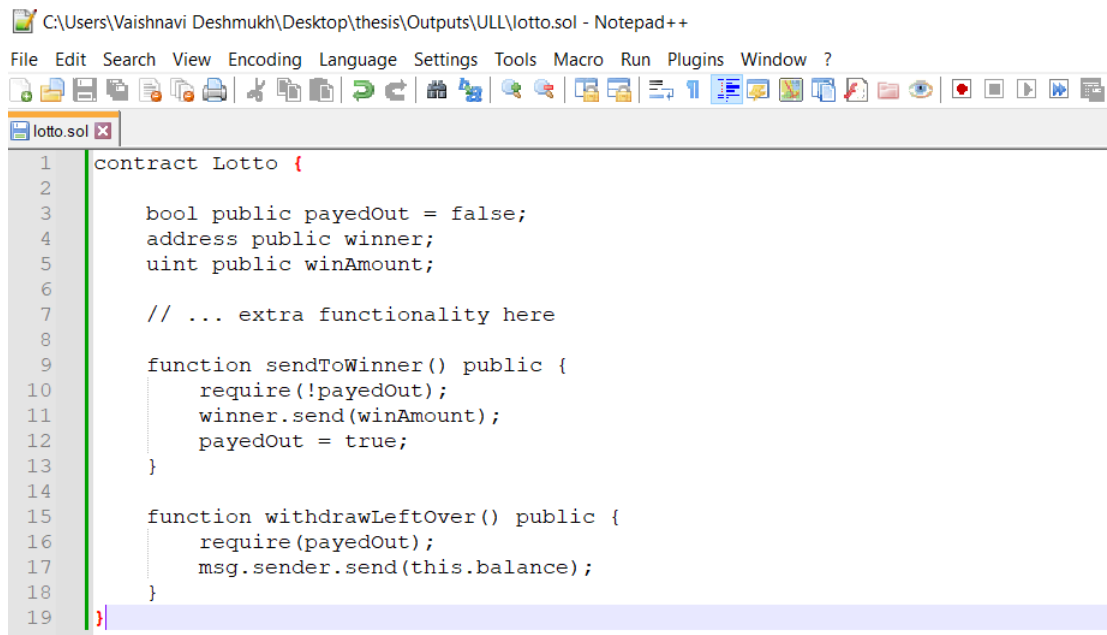
```
C:\Users\Vaishnavi\Deskto\thesis\Outputs\Time Manipulation\example.sol - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example.sol
1 contract TimeManipulation {
2     uint public releaseTime;
3     uint public amount;
4
5     constructor(uint _releaseTime, uint _amount) public {
6         releaseTime = _releaseTime;
7         amount = _amount;
8     }
9
10    function withdraw() public {
11        require(now >= releaseTime, "Funds cannot be withdrawn yet");
12        msg.sender.transfer(amount);
13    }
14 }
```

Figure 2.4: Example Contract for Time Manipulation Vulnerability

## 2.1.5 Unchecked Low-Level Calls

A contract can call another contract in Ethereum in a variety of ways using a variety of commands, such as CALL, CALLCODE, DELEGATECALL, and STATICCALL. These directives are regarded as low-level directives. In order to call another contract when writing a contract, the user will use higher-level methods like send or transfer. Send() only returns false when an exception occurs and does not propagate exceptions. The caller execution will proceed as if nothing happened if the callee execution is interrupted due to an exception and the user calls the send function without inspecting the return value. It is advised to examine the return value when using low-level instructions directly, the transfer function, or other functions to help mitigate this type of vulnerability.

The example shown in Fig2.5 demonstrates the unchecked low level call, here when a send() is used without examining the response, a problem occurs even if ether was sent or not, payedOut() can be set to true if a winner's transaction fails. In this instance, the withdrawLeftOver() can be used by the general public to withdraw the winner's rewards.



```
C:\Users\Vaishnavi\Deskmukh\Desktop\thesis\Outputs\ULL\lotta.sol - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
lotta.sol
1 contract Lotto {
2
3     bool public payedOut = false;
4     address public winner;
5     uint public winAmount;
6
7     // ... extra functionality here
8
9     function sendToWinner() public {
10         require(!payedOut);
11         winner.send(winAmount);
12         payedOut = true;
13     }
14
15     function withdrawLeftOver() public {
16         require(payedOut);
17         msg.sender.send(this.balance);
18     }
19 }
```

Figure 2.5: Example Contract for Unchecked Low Level Calls Vulnerability

## 2.2 Preliminaries

Most of the smart contracts are written in Solidity. In this study, as an input, we are taking a Solidity file or the byte code corresponding to it. Generally, Solidity files have common features like contract definition, state variables, functions, modifiers, constructors, Inheritance, and Libraries.

### 2.2.1 Rattle

In our research for intermediate representation(IR), we are taking support of the Rattle tool, this will elevate the bytecode to an IR and change the instructions' form from a stack to a register and into a single static assignment form. It will also be built to house the CFG. Functions are recovered and split off. Additionally, function arguments, memory locations, and storage locations are recovered. The following instruction shown in the listing would produce an output like shown in Fig 2.6.

```
python3 rattlecli.py -input inputs/kingofether/KingOfTheEtherThrone.bin O
```

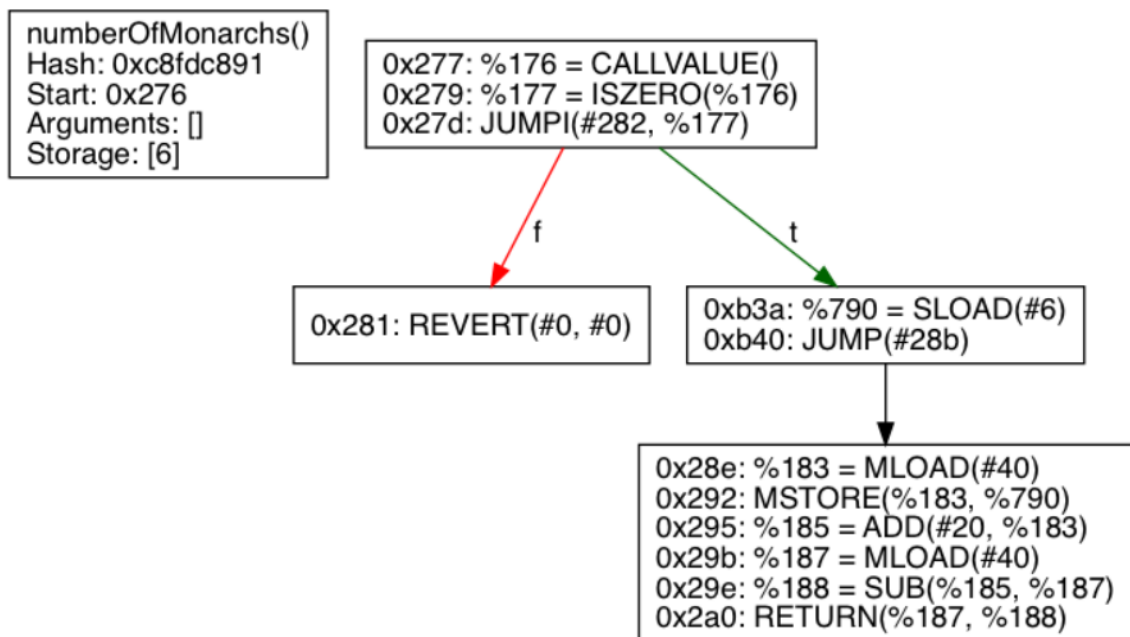


Figure 2.6: Register Machine Output



## 2.2.2 Satisfiability Modulo Theories (SMT Z3 Solver)

The SMT Z3 solver is a powerful automated theorem prover developed by Microsoft Research. It is widely used in the field of formal verification and software analysis to solve logical formulas involving constraints from different theories, such as integer arithmetic, real arithmetic, arrays, and bit-vectors. Let's proceed with a simple example to illustrate these concepts. We can query an SMT solver such as Z3 to determine whether the expression  $x + y = 5$  can be satisfied. In the context of integers, e.g. there are integer values  $x$  and  $y$ , that add up to value 5. We will express the problem in the SMT-LIB language Fig 2.7. When we run the

```
; this is a comment - it is ignored by solvers

; declare x as integer
(declare-const x Int)

; declare y as an integer
(declare-const y Int)

; express the problem - e.g. add the formula to the list of formulas we are trying to
prove
(assert (= (+ x y) 5))

; query the solver to determine if the SMT problem is satisfiable
(check-sat)

; if it is, show one possible solution
(get-model)
```

Figure 2.7: Example Problem for Z3 Working

Z3 SMT solver, we obtain the response as shown in Fig 2.8. Here sat means that the solver determined that the formula can be satisfied. It identified one solution (model), with  $y = 0$  and  $x = 5$  that satisfies the problem as stated. This was the basic understanding of how Z3 works.

```
sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    5)
)
```

Figure 2.8: Response of Z3

## 2.3 Static Analysis Tools

Static analysis is a technique in software development that allows you to evaluate code without running it. It entails inspecting a program's source code, bytecode, or built binary for potential flaws like as bugs, security vulnerabilities, or coding mistakes. Static analysis examines the code's structure, grammar, and semantics rather than its runtime behaviour [12]. There are broadly three categories of how the tools are divided, as seen in Fig 2.9, static analysis tools contribute 75%. In our study, we take into account static analysis tools only.

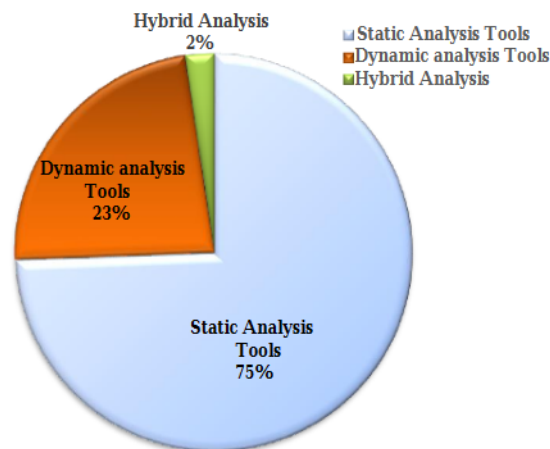


Figure 2.9: Category-wise Division of Smart Contract Analysis Tools [16]

### 2.3.1 Symbolic Execution

Symbolic execution is a method used in software analysis and testing to investigate a program's various routes and behaviors. It entails running a program using symbolic values rather than physical inputs. Symbolic execution enables the methodical examination of multiple execution routes, finding probable program behaviors and potential vulnerabilities by considering inputs as symbolic variables [18].

The control flow of the program is analyzed during symbolic execution, and restrictions are constructed depending on the conditions and branching statements encountered. The links between symbolic inputs and program states are represented by these limitations. The symbolic execution engine then attempts to meet these requirements by employing constraint-solving approaches such as SMT solvers. In this subsection, we present eight symbolic execution tools that we are working on for comparative study:

#### **Honeybadger [33]**

Honeybadger was developed in the year 2019 by a group of researchers at the University of Luxembourg. The authors conducted a largescale analysis of over 2 million smart contracts. They identified 690 honeypots Smart with his contracts in circulation and his 240 victims, totaling over \$90,000 in profits for the honeypot creator. Manual verification shows that 87% of reported contracts are actually honeypots. There are three implications of their study First, honeypots are a real and growing threat to the Ethereum ecosystem. Second, HoneyBadger is a valuable honeypot detection tool. Third, developers should be aware of the honeypot threat and take steps to protect their contracts.

**Current Version:** ff30c9a

**Input:** Solidity file.

#### **Manticore [21]**

This tool is a contribution of TrailOfBits, it is a symbolic execution that is also used to uncover execution pathways in EVM bytecode that lead to reentrancy vulnerabilities and accessible self-destruct actions. This tool combines static and dynamic analysis. However, it can be challenging to utilize static analysis to detect defects that are only triggered by particular inputs. Static analysis can find bugs that are not reachable during runtime. While dynamic analysis can reveal faults that can

be fixed at runtime, it can also be laborious and inefficient. Static analysis is used by Manticore to identify potential bug locations, while dynamic analysis is used to investigate the state space surrounding those spots. Some of Manticore's salient characteristics are, even for users who are unfamiliar with symbolic execution, Manticore is made to be simple to use. Manticore is adaptable and may be used to examine many different types of software, including binaries, smart contracts, and bytecode. Manticore is a powerful tool that has been used to identify bugs and security flaws in a range of software.

**Current Version:** 0.3.7

**Input:** Solidity file.

### **Mythril [28]**

Mythril open-source analytical tool was created in 2017 by ConsenSys, analyses Ethereum blockchain-based smart contracts but also works for other blockchain platforms. Mythril uses three approaches for analyzing smart contracts: symbolic execution, SMT solving, and taint analysis. It can also be used in combination with other tools. There are a number of limitations to Mythril, such as its reliance on symbolic execution, which can be computationally expensive. There are a number of limitations to Mythril, such as its reliance on symbolic execution, which can be computationally expensive.

**Current Version:** 0.23.15

**Input:** Solidity file, byte code.

### **Osiris [32]**

Osiris was created in 2018 and is an extension of the Oyente tool that is freely accessible and developed in Python. It comprises a mix of two techniques—symbolic execution and taint analysis—to find mathematical weaknesses. Arithmetic, truncation, and signedness issues are three forms of integer vulnerabilities that it may identify. The tool is evaluated with 1.2 million or more smart contracts. The tool was able to identify 42,108 contracts with integer issues, they discovered. 1,272 of these contracts had previously disclosed known vulnerabilities. Twelve new vulnerabilities were also discovered by Osiris, one of which was critical and present in a contract that was being used on the Ethereum network.

Osiris can identify a wider range of integer issues than previous techniques. Compared to other instruments, Osiris has a reduced rate of false positives. Osiris can be used to identify vulnerabilities, both known and unknown. A useful tool for

enhancing the security of Ethereum smart contracts is Osiris.

**Current Version:** d1ecc37

**Input:** Solidity file.

### **Oyente [20]**

It is one of the earliest tools that analyze smart contracts using symbolic execution and statically examines the program code route using a path.CFG constructor, Explorer, Core analyzer, and Validator are its four key architectural components. The Z3 bit-vector solver is used by the Explorer and Validator to delete traces that are demonstrably impossible. This tool examines nearly 19k smart contracts out of which it flags 8k contracts and finds 1682 are distinct and were able to collect source code for only 175 contracts to confirm the tool's correctness.

**Current Version:** 480e725

**Input:** Solidity file.

### **Securify [34]**

Security analysis is done in two phases. The contractual dependency graph is symbolically executed in the first stage, and relevant semantic data is extracted ie Securify's approach is based on automatic inference of semantic program facts followed by checking of compliance and violation security patterns over these facts. The second stage determines whether or not a property is secure by looking for patterns of compliance and violations. Securify uses compliance and violation patterns to guarantee that certain behaviors are safe and, respectively, unsafe. The remaining behaviors are reported as warnings (to avoid missing errors), and more than 18k smart contracts in the real world.

**Current Version:** Securify 2.0

**Input:** Solidity file.

### **Slither [10]**

This study introduces Slither, a static analysis system created to offer comprehensive data on Ethereum smart contracts. It operates by transforming Solidity smart contracts into a translation format called SlithIR. SlithIR uses Static Single Assignment (SSA) form and a condensed instruction set to simplify analysis implementation while retaining semantic data that would be lost if Solidity were converted to bytecode. Dataflow and taint tracking are two frequently used program analysis approaches that can be applied using Slither. Four key use cases comprise our

framework: (1) automated vulnerability discovery, (2) automated code optimization opportunity discovery, (3) enhanced user knowledge of the contracts, and (4) help with code review. This tool's open-source version may identify around 20 issues, including Reentrancy, suicidal contracts, locked ether, and arbitrary ether sending. The authors assessed Slither's bug-finding abilities by contrasting its performance on reentrancy bugs with that of other readily available cutting-edge tools.

**Current Version:** v0.9.3

**Input:** Solidity file.

### **Smartcheck [30]**

Lexical and syntactic analysis is the method used by SmartCheck to examine the smart contract. ANTLR (a parser generator) and a unique Solidity grammar are used to create an XML parse tree as an intermediate representation. For the purpose of processing intermediate representation and identifying vulnerability patterns, XPath queries are employed. It recognizes about 20 different sorts of vulnerabilities, including implicit visibility level, compiler version not fixed, division by zero, and violation of style guides. Reentrancy vulnerabilities, transaction order dependence vulnerabilities, integer overflow vulnerabilities, and other security flaws are all detectable by SmartCheck in Ethereum smart contracts. For identifying potential security flaws in Ethereum smart contracts, SmartCheck is a helpful tool. The creators of SmartCheck intend to continue developing the program in the future, adding new capabilities and enhancing the tool's precision.

**Current Version:** v2.0

**Input:** Solidity file.

## **2.4 Dynamic Analysis Tools**

### **ContractFuzzer [17]**

It was created in 2018 and is an open-source dynamic analysis tool that is freely accessible. It is a tool for fuzzing Ethereum smart contracts to find security flaws. For the purpose of creating fuzzing inputs, it employs smart contract ABI standards. To find security flaws, it categorizes test oracles. By instrumenting the Ethereum virtual computer, the behavior of smart contracts during their runtime is recorded. Finally, these logs are examined to identify security flaws.

### **ContractGuard [37]**

It was created in 2019 and is a JavaScript-based dynamic analysis tool. It makes use of an effective anomaly-based intrusion detection system methodology. When it notices any unusual behaviour, it alerts the administrators and restores the smart contract's status to its prior secure state.

### **ContractLarva [22]**

It was created in 2017 and is a freely accessible open-source dynamic analysis tool written in Haskell and TeX. It operates on Solidity code and is a runtime verification tool. This utility implements event triggering and monitoring logic for the Ethereum smart contract. Dynamic event automata are used to specify the attributes needed to monitor the events. Both control flow events and data flow events are recorded by the tool.

### **EthBMC [45]**

The dynamic analysis tool was created in 2020. This tool accepts input for EVM byte code analysis. It is a symbolic execution-based automated vulnerability detector. It investigates the potential state space that a programme can access. It uses some constraints to encrypt the attackers' intended outcome, and then the SMT solver is used to resolve that constraint. When dealing with parity bug vulnerabilities, it is effective.

### **Etherolic [3]**

It is a Rust-based dynamic analysis tool. The etherolic analysis approach uses console testing and dynamic taint tracking to examine the byte code of the Ethereum virtual machine. It both locates weaknesses and produces exploits to set off unforeseen mistakes. It is capable of detecting short addresses, denial of service, bad randomness, locked ether, re-entrancy, unhandled exceptions, integer overflow/underflow, and bad randomness.

### **Ethlint [1]**

It was created in 2016 and is a freely accessible open-source JavaScript dynamic analysis tool. Its previous name was Solium. It looks for style and security issues in the Solidity code. It borrows concepts from Solidity Parser and ESLint, a static analyzer for JavaScript code.

### **Harvey [5]**

The invention of this dynamic analysis tool took place in the year 2020. Harvey is a grey-box fuzzer for smart contracts, which is nothing more than a quick way to create tests to find security flaws. Assertion violations, as specified in SWC 110, and memory access errors, as described in SWC 124, are the two main sorts of defects that Harvey mostly finds.

### **ModCon [19]**

It was created in 2020 and is a JavaScript-based dynamic analysis tool. It is a model-based testing framework that can be used with both permissioned and permission-less blockchain platforms. It defines test oracles using user-defined models. The front end of ModCon is web-based, and the back end is powered by JavaScript. The user must provide both a test model specification and a smart contract as inputs.

### **Solitor [29]**

It was created in 2018 and is a dynamic analysis tool that is implemented in Java. Solidity monitor is abbreviated as Solitor. Using annotations, the user of this tool can define the behaviour. These annotations can be used to test whether or not certain properties hold at runtime.

### **Vultron [36]**

It was created in 2019 and is a freely accessible, open-source, JavaScript dynamic analysis tool. It suggests a strategy for developing a mechanism to distinguish between irregular transactions and regular ones. This tool can enable a wide range of downstream analysis techniques like testing, fuzzing, verification, and symbolic execution. ContraMaster is one of its other names.

## **2.5 Comparison Between Static and Dynamic Analysis**

Static analysis examines the code structure without running it, allowing for early vulnerability detection. Dynamic analysis, on the other hand, comprises running the code and analysing its behaviour in real-world contexts. The key distinctions between static and dynamic analysis are as follows:



1. Static analysis tools examine the source code or bytecode. To find any potential problems, they look at the grammar, control flow, and code structure. whereas dynamic tools focus on the actual program’s inputs, memory use, and network communications.
2. Static analysis is often carried out during the development phase. It permits the early detection of weaknesses and coding mistakes. runtime overhead is incurred by dynamic analysis since it necessitates code execution. This might not be appropriate for large-scale systems analysis or time-sensitive applications due to its performance implications.

Some of the well know static analysis tools that we included in our study are listed in section 2.3.1. A relative analysis of popular static and dynamic analysis tools is shown in table 2.1 below:

Table 2.1: Classification of Tools ( AI- Abstract Interpretation, CS- Constraint Solving, CT- Code Transformation, CI- Code Instrumentation, FT- Fuzz Testing, MBT- Model-Based Testing, SE-Symbolic Execution, TA- Taint Analysis, ML-Machine Learning. )

Tool	Static	Dynamic	Code	CLI	Year	Platform	Approach
ContractFuzzer [17]		Yes	Yes	Yes	2018	Go	FT
ContractGuard [37]		Yes		Yes	2019	JavaScript	ML
ContractLarva [22]		Yes	Yes	Yes	2017	Haskell, Tex	CI
EthBMC [45]		Yes	Yes	Yes	2020	Rust	SE
Etherolic [3]		Yes		Yes	2020	Rust	FT & TA
Ethlint [1]		Yes		Yes	2016	JavaScript	CI
Harvey [5]		Yes		Yes	2020	-	FT
Honeybadger [33]	Yes		Yes	Yes	2019	Python	SE & CS
Manticore [21]	Yes		Yes	Yes	2017	Python	SE
ModCon [19]		Yes		Yes	2020	JavaScript	MBT
Mythril [28]	Yes		Yes	Yes	2017	Python	SE & CS
Osiris [32]	Yes		Yes	Yes	2018	Python	SE
Oyente [20]	Yes		Yes	Yes	2016	Python	SE
Securify [34]	Yes		Yes	Yes	2018	JAVA	AI
Slither [10]	Yes		Yes	Yes	2018	Python	CT & CS
SmartCheck [30]	Yes		Yes	Yes	2019	C++	CT
Solitor [29]		Yes	Yes	Yes	2018	-	CI
Vultron [36]		Yes	Yes	Yes	2019	JavaScript	FT

## 2.6 Chapter Summary

Summarizing this chapter, we learned about different vulnerability types constituting modules that are implemented. For this study later we also discussed static and dynamic analysis and also why we choose static analysis. Lastly, the tools associated with the study are described.

## CHAPTER 3

# Literature Survey

This chapter provides an in-depth exploration of blockchain technology, Ethereum, and Bitcoin. It also covers the architecture of blockchain, consensus mechanisms, privacy challenges in Bitcoin, and techniques to enhance privacy within the Bitcoin ecosystem. The survey also investigates the applications of blockchain technology beyond Bitcoin in various industries, examines vulnerabilities and security analysis of smart contracts, and discusses notable attacks such as the DAO attack. Furthermore, it presents static and dynamic analysis tools used for detecting vulnerabilities in smart contracts. The survey serves as a foundation for further research in this field.

### 3.1 Blockchain, Ethereum, and Bitcoin

This section talks about the architecture of blockchain and various consensus mechanisms used in blockchain networks, and also how to examine the privacy challenges in Bitcoin, and explores techniques to improve privacy within the Bitcoin ecosystem. Zheng [42] [43] explains the architecture of a blockchain in detail, what is the structure of blocks their transactions, and the decentralized network. It also discusses different blockchain designs, such as consortium, private, and public blockchains. Additionally, it describes PoW, PoS, and PBFT consensus algorithms, highlighting their benefits, drawbacks, and security issues.

Blockchain has a very famous application which is Bitcoin, which was implemented in the year 2008 by a group of unknown people. Herrera [13] introduces Bitcoin and discusses its decentralized nature. It addresses the misconception of total anonymity in Bitcoin and explores challenges related to privacy, including linkability, traceability, and the lack of built-in identity. It also discusses anonymity techniques such as coinjoin, stealth addresses, and which aim to enhance privacy within the Bitcoin ecosystem.

Ethereum is a decentralized blockchain with smart contract functionality. Ether is the native cryptocurrency of the platform. Among cryptocurrencies, ether is second only to Bitcoin in market capitalization. It is open-source software. Ethereum was conceived in 2013 by programmer Vitalik Buterin. Vujicic [35] presents the Ethereum platform and provides a brief comparison between Bitcoin and Ethereum. It emphasizes that while both platforms leverage blockchain technology, Ethereum offers a platform for DApps and smart contracts, whereas Bitcoin primarily focuses on digital money transactions.

## **3.2 Application of Blockchain**

We explored the diverse applications of blockchain beyond Bitcoin, discussing its implementation in various industries and sectors. Hsaed [26], Palihapitiya [23], Jafar [14], Chen [6], Kakkar [15] collectively discuss the applications of blockchain in different domains, highlighting its potential beyond Bitcoin. Each paper focuses on a specific industry or sector where blockchain technology is being utilized. Hsaed [26] explores the use of blockchain in healthcare, discussing how the industry is changing due to this technology. It addresses the challenges faced and highlights developments in this domain. Palihapitiya [23] specifically discusses the use of blockchain in the finance and banking industry, highlighting its potential benefits and applications in this sector. Jafar [14] explores the use of blockchain in voting systems, discussing its potential to enhance transparency, security, and efficiency in the voting process.

## **3.3 Detecting Vulnerability and Security Analysis of Smart Contracts**

There are many vulnerabilities but for this study, we only focus on five categories. Ashizawa [2] the author describes the importance of detecting vulnerabilities in these contracts to mitigate security risks. It discusses the challenges of accurately identifying vulnerabilities, especially in complex smart contracts. As bugs are found in smart contracts it is very important to ensure security for smart contracts, explore different types of vulnerabilities and attacks, and discuss various

security analysis methods and tools. In this study, the authors have mentioned 6 security patterns which consist of Checks-Effects-Interaction, Emergency Stop (Circuit Breaker), Speed Bump Pattern, Rate Limit Pattern, Mutex, and Balance Limit Pattern. They emphasize the need for these security methods and their limitations and their solutions.

P. Praitheeshan [24] explores techniques such as static analysis, dynamic analysis, system execution, and formal verification for smart contracts. The authors also discuss various tools and frameworks available for the security analysis of smart contracts. The efficiency of various security analysis techniques and tools is assessed and contrasted in this study based on their capabilities, constraints, and the kinds of vulnerabilities they may identify. Tsankov [39] highlights the importance of securing smart contracts and preventing vulnerabilities that can lead to attacks. similar work is covered by the authors in Qian [25], C. F Torres [31], Zhou [44].

### **3.4 The DAO Attack and Vulnerability**

Zhao [41] discusses the infamous DAO attack, which resulted in the theft of a significant amount of funds. It highlights the implications of the attack and the paradoxes it revealed in blockchain-based systems. The author emphasizes the importance of conducting rigorous security analysis and verification in decentralized systems. The paper also suggests lessons learned from the attack to enhance the security and robustness of future decentralized systems. The vulnerability that was associated with the Dao attack is Reentrancy. For this, an article, where Chaturvedula [4] explores reentrancy attacks in blockchain systems, explains the mechanism of reentrancy attacks, discusses countermeasures to prevent such attacks, and presents detection techniques. The paper emphasizes the importance of understanding and mitigating reentrancy attacks to ensure the security and integrity of blockchain systems. Yellu [40] focuses on arithmetic vulnerabilities in blockchain systems and provides countermeasures to mitigate such vulnerabilities. The author also discusses the two conditions for such vulnerabilities that is Integer Overflow or Integer Underflow. It also addresses the specific challenges associated with arithmetic vulnerabilities and presents techniques to enhance the security of blockchain systems in this regard.

### 3.5 Static Analysis Tools

The primary goal of static analysis is to catch software defects early in the development process before the code is executed or deployed. By detecting potential issues before runtime, static analysis helps improve code quality, reliability, and security. It can also help identify areas for code optimization and adherence to coding standards.

Ghaleb and Pattabiraman [12] focus on static analysis tools to find bugs in smart contracts. Authors use the bug injection technique where the bugs are injected into smart contracts in order to measure the tool's ability to detect the vulnerabilities and also define evaluation metrics such as true positive, false positive, false negatives, true negatives, and execution time. They conclude by emphasizing the importance of evaluating these tools using realistic scenarios to enhance their effectiveness and reliability. There are various methods in static analysis among which the Symbolic execution is a part. Liew [18] emphasizes symbolic execution and how applying symbolic execution optimizes the results.

Torres [33] discusses the Honeybadger tool, which utilizes symbolic execution to identify vulnerabilities in smart contracts. It highlights how symbolic execution can optimize the results of vulnerability detection. Mossberg [21] introduces the Manticore framework, which enables users to perform symbolic execution on smart contracts. It discusses the capabilities and applications of Manticore, emphasizing its use in analyzing smart contracts. Sharma [28] presents the Mythril tool, which employs symbolic execution and relies on the Z3 solver. It highlights the tool's usage of symbolic execution techniques, discusses limitations and challenges, and demonstrates a vulnerability discovered by the tool. Torres [32] introduces Osiris, a static analysis tool designed to detect potential vulnerabilities and risks in Ethereum smart contracts. It specifically focuses on identifying integer bugs and highlights the key contributions of the Osiris tool. Luu [20] discusses vulnerabilities such as reentrancy, transaction order dependency, and timestamp dependency in smart contracts. It introduces the Oyente tool, which utilizes intermediate representation and symbolic execution techniques to analyze smart contracts. Tsankov [34] presents the Securify tool, which leverages static analysis techniques to analyze smart contracts. It evaluates the precision, recall, and false positive rates of the tool and provides insights into its performance. Feist [10] introduces the Slither tool, which applies static analysis techniques to smart

contracts. It focuses on detecting vulnerabilities and provides an overview of the tool's capabilities. Tikhomirov [30] presents the SmartCheck tool, which utilizes static analysis to analyze Ethereum smart contracts. It discusses the tool's approach and features, emphasizing its ability to process Solidity files.

### 3.6 Dynamic Analysis Tools

Dynamic analysis tools are essential software security assessment techniques that analyze program behavior by monitoring variables, function calls, and data flow at runtime. Jiang et al. [17] offer ContractFuzzer, a tool for the dynamic analysis of smart contracts. They emphasize the use of fuzzing techniques to find smart contract vulnerabilities and talk about how well ContractFuzzer can spot potential security risks. Azzopardi et al. [22], present Contractlarva, a dynamic monitoring tool for smart contracts. They highlight the importance of monitoring smart contracts in real-time to ensure their proper behavior and discuss the challenges and future directions in this area.

Wang et al. [37], proposes Contractguard, a dynamic analysis tool for Ethereum smart contracts that incorporates embedded intrusion detection capabilities. The authors address the security challenges associated with smart contracts and introduce Contractguard as a solution to detect and defend against potential intrusions. They discuss the architecture and functionality of Contractguard, emphasizing its effectiveness in enhancing the security of Ethereum smart contracts. Liu et al. [19] the authors present Mod-Con, a model-based testing platform specifically designed for smart contracts. They discuss its capabilities and how it can be used for effective testing and validation of smart contracts. Frank et al. [45] introduce ETHBMC, a bounded model checker designed for analyzing smart contracts. The authors highlight the importance of security analysis for smart contracts and present ETHBMC as a tool that can detect vulnerabilities by exploring various execution paths within a bounded context. Similarly, Ashouri [3] talks about Etherolic, and Raghava introduces Ethlint [1] which are similar to the other dynamic tools.

## 3.7 Modules

Williams [38] developed the Rattle tool where he explains how the tool is used for data mining and data analysis. The tool comes with a graphical user interface (GUI) that allows users to perform various data mining tasks, such as data pre-processing, visualization, modeling, and evaluation. Rattle is primarily designed to work with the R programming language, which is widely used for statistical computing and data analysis. In our study, we are using rattle as an IR for implementation.

Moura and Bjørner [8] introduce Z3 and its key feature for solving SMT problems. The authors also explain how Z3 is designed to be easily integrated into various software tools and frameworks. It provides interfaces in multiple programming languages, including C++, C, Python, and Java, enabling developers to use Z3 as a library within their applications. This flexibility allows Z3 to be seamlessly integrated into existing software pipelines and workflows. According to the authors, Z3 can be used to perform symbolic execution, generates test cases, and be used in formal verification to demonstrate the correctness of hardware and software systems. Additionally, it is used in program analysis for program synthesis, program repair, and bug identification. Z3 is used in automated planning, constraint resolution, and reasoning problems in artificial intelligence. Z3 is an open-source version of Solver and is freely available to use.

## 3.8 SmartBugs

This study works close with SmartBugs, Ferreira [11] proposes SmartBugs a framework that talks about collecting smart contracts from etherscan.io which is injected with annotated vulnerabilities, that dataset contained 69 contracts with tagged vulnerabilities. This paper analyses 9 static analysis tools and determines their true positives, true negatives, and performance of the tools. compares all the tools on these metrics and records results.

Durieux [9] discusses collecting 47k smart contracts which are filled with 97% vulnerabilities in it, here 9 analysis tools were taken and this dataset is run on various tools, the results are recorded for every tool, it has come to the notice of



the authors that by combining all the tools together <50% vulnerabilities are detected by the tool and author further conclude by saying that there is still the room for more analytical tools.

### **3.9 Chapter Summary**

In this chapter, we categorize the research papers into 8 categories where the first section talks about Blockchain, Ethereum, and Bitcoin followed by different applications of blockchain other than Bitcoin. Then it also describes the detection of vulnerabilities and security analysis of smart contracts which is followed by another section explaining the famous DAO attack and the vulnerability associated with it. Moving further the static and dynamic analysis tools are discussed, proceeding with the module section which describes the Rattle and Z3 solver, and lastly talks about a framework for smart contracts which we are closely working with.

## CHAPTER 4

# Proposed Method

In this chapter, we go over our strategy, which makes use of the modular approach as seen in Fig 4.1. The fundamental concept is to use symbolic execution along with it rattle [38] is used as an IR tool, the SMT Z3 solver [8] is used for program analysis verification after which the category of vulnerability is discovered, and finally, the line number and vulnerability types are recorded in the form of a text file.

### 4.1 Workflow

The interaction with our approach can be done with a command line interface(CLI). Researchers are experimenting with translating smart contract source code or bytecode into an IR containing highly semantic data to analyze smart contracts. By examining the intermediate representation of the contract as input, as well as the contract's related bytecode or Solidity source code, they can identify security flaws. When source code is provided, the best compiler is used to compile it. We are using the same concept where for the IR we are employing is the rattle module for which only Solidity written source code is currently given. The rattle [38] module will get the bytecode.

The bytecode will be elevated to an IR and instructions will be transformed into Single Static Assignment(SSA) form in the Rattle module. In SSA each variable is assigned only once, at its definition point. This property makes the control flow and data dependencies in the program explicit. It will also be built to house the CFG. When a variable is assigned a new value, instead of modifying the existing variable, a new variable with a different name is created. This ensures that previous values of the variable are preserved and can be referred to in the control flow graph.

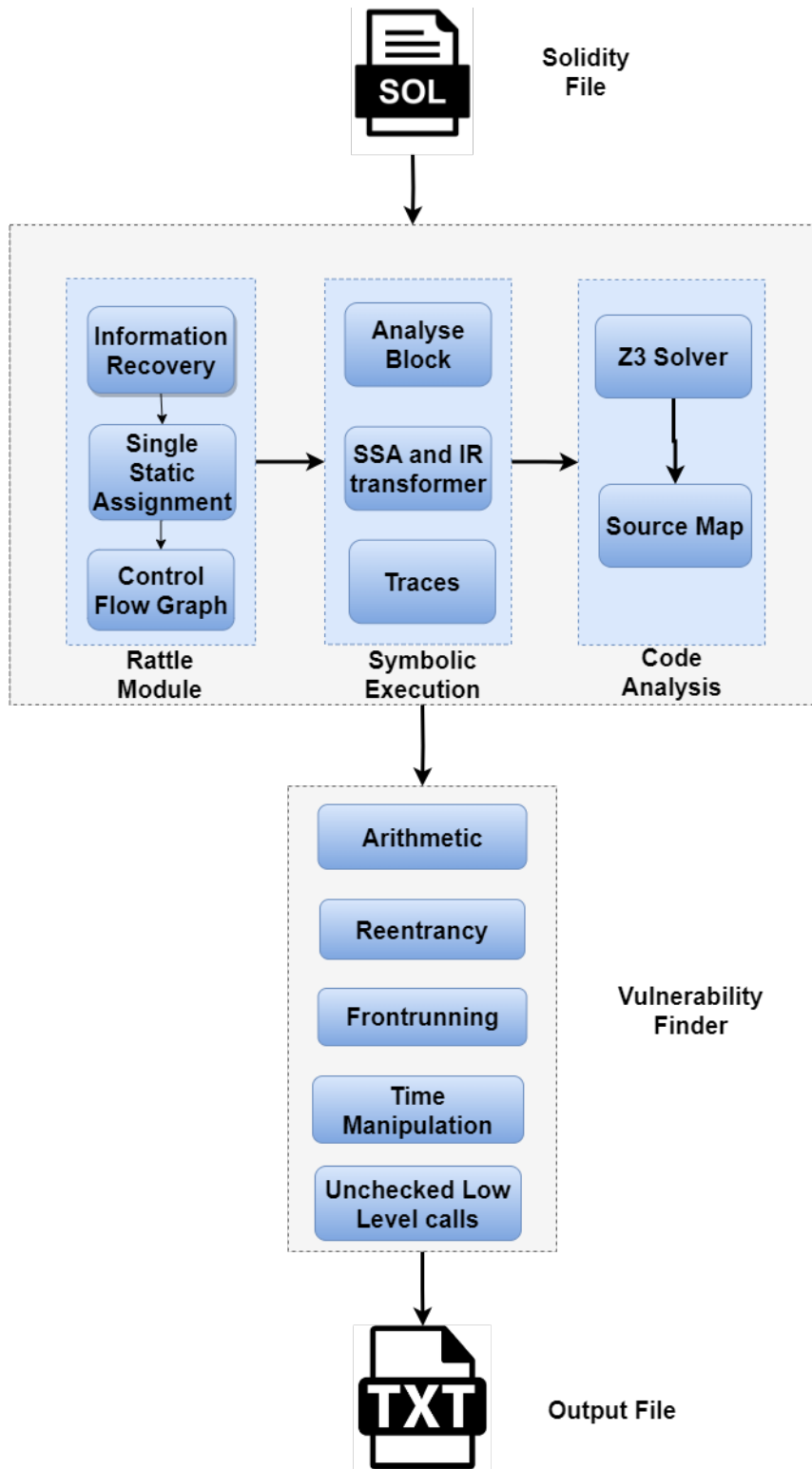


Figure 4.1: Proposed Architecture

Moving further the CFG is traversed in the symbolic execution phase, the core concept behind symbolic execution is to symbolize variables in the program code, which by symbolizing program input, maintains a set of restrictions for all execution routes. The constraint solver is used in symbolic execution to resolve constraints and identify the input reason for execution. Last but not least, programmers can utilize the constraint solver Z3 [8] to obtain a fresh test input to determine whether the symbol value has a possible weakness or not.

The following phases are used to categorize the symbolic execution method used to discover smart contract vulnerabilities:

1. Define the contract variable values.
2. Specify each step of the execution program's instructions.
3. Search all executable routes, update the execution status, and gather path restrictions.

Here in our approach to support symbolic execution the constraint solver used is the SMT Z3 solver. Finally, this complete information is used to detect the vulnerability type and line number as an output in the form of a text file.

#### 4.1.1 Algorithm for Master File

This algorithm performs symbolic execution and vulnerability analysis on Ethereum virtual machine (EVM) bytecode or Solidity source code. It utilizes data structures like dictionaries, CFG, execution traces, and custom classes to facilitate the analysis process. Here is a brief explanation of what the algorithm does and the key data structures used.

1. Importing multiple libraries for various functions, such as **argparse** for CLI parsing, logging for message logging, **solcx** for dealing with the Solidity compiler, **Solidity\_parser** for parsing Solidity source code, and other custom modules.
2. A **logger** class instance from the logging module that is used to record messages. An instance of **argparse.ArgumentParser** for parsing command-line arguments and options.
3. The command-line arguments are defined by **argparse.ArgumentParser**. The input file (EVM bytecode or Solidity source code file), verbosity level, vulnerability type, maximum recursion depth, and other settings are among

---

**Algorithm 1** Algorithm for master file

---

```
1: Input: SolidityFile, Verbosity, VulnType, MaxDepth, FindAllVulnerabilities, Timeout
   Output: Detected vulnerabilities
2: Initialize configurations and dependencies such as logging, solcx, solidity_parser.
3: Parse command-line arguments using argparse package.
4: Configure logging based on the provided verbosity level.
5: Read the input file and determine its type (Solidity file or bytecode).
6: if SolidityFile then
7:     Retrieve the solc version from the Solidity file's pragma directive.
8:     Install and set the solc version.
9:     Compile the Solidity file and extract the runtime bytecode with
       solcx.compile_files() function.
10: else if then
11:     Read the bytecode from the input file.
12:     for each bytecode do
13:         Perform Sym_Exec() on the bytecode to generate constructor traces.
14:         Initialize the VulnerabilityFinder( ) module with the traces, functions, name,
           source map, and FindAllVulnerabilities flag.
15:         Analyze vulnerabilities using the specified vulnerability types.
16:         Store the detected vulnerabilities.
17:     end for
18:     Output the detected vulnerabilities
```

---

the arguments. The parameter parsing and help messages are automatically generated by the **argparse** package.

4. The verbosity argument that is supplied by the script determines the logging level that will be used. Messages are logged in accordance with the log level that has been established for the **logger**.
5. The script determines the Solidity version used in the file and sets it as the active version using the **solcx** library if the input file is a Solidity source code file. Then, a dictionary of compiled contracts is returned by the function **solcx.compile\_files()** function, which compiles the Solidity source code.
6. The script runs symbolic execution and vulnerability analysis on each compiled contract. The control flow graph (CFG) is recovered using the Recover class from the rattle module, and bytecode optimization is also done. The symbolic execution on the CFG is carried out using the **SymExec** class from the **sym\_exec** package, and execution traces are obtained. The execution traces, contract functions and other data are instantiated with the **VulnerabilityFinder** class from the **vuln\_finder module**. It actually conducts the vulnerability analysis. With the given vulnerability categories to investigate, the method **VulnerabilityFinder.analyse\_only()** is invoked. The script

logs and outputs details about any vulnerabilities it discovers along with the model values they are linked with.

7. The entry point for the script is defined as the **main()** function. It does the vulnerability analysis, configures logging, and parses the command-line inputs. In the event that the script is run directly (`__name__ == '__main__'`), the `main()` method is invoked.

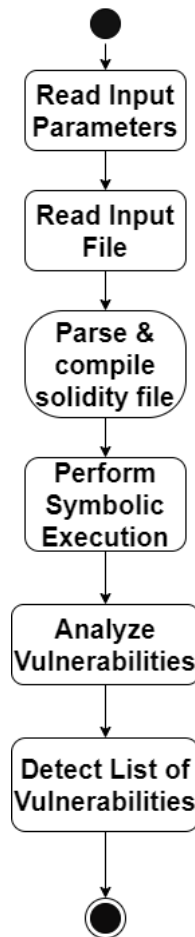


Figure 4.2: Execution of Master File

### Time and Space Complexity

This parsing `solc.compile_files()` operation has a time complexity of  $O(n)$ , where  $n$  is the size of the Solidity file. If the input file is a Solidity file, the algorithm proceeds to compile it. The time complexity of the compilation process depends on the complexity of the Solidity code and the number of contracts within the file. The compilation step typically has a complexity of  $O(k)$ , where  $k$  is the number of contracts in the file.

The `get_constructor_traces` function performs symbolic execution on the bytecode to generate constructor traces. The time complexity of symbolic execution depends on the complexity of the bytecode and the number of paths explored during execution. It can vary, but it generally ranges from  $O(n)$  to  $O(n^2)$ , where  $n$  is the size of the bytecode.

The `VulnerabilityFinder` module analyzes the traces and performs vulnerability analysis based on the specified vulnerability types. The time complexity of vulnerability analysis depends on the complexity of the traces and the number of vulnerability types being analyzed. It typically has a complexity of  $O(m * n)$ , where  $m$  is the number of traces and  $n$  is the number of vulnerability types. Total Time Complexity is given as  $O(n + k + n^2 + m * n)$ .

For Analyzing the space complexity of the algorithm, the parsing file depends on the size of the Solidity file which can be given in  $O(n)$ . During Solidity compilation, memory usage depends on the complexity of the Solidity code and the number of contracts being compiled. The space complexity for compilation is typically  $O(k)$ , where  $k$  is the number of contracts in the Solidity file.

The space complexity of the symbolic execution `SymExec` class depends on the memory required to represent the program state, constraints, and execution paths. As symbolic execution explores different paths, it accumulates additional memory usage. The space complexity of symbolic execution is typically  $O(n)$ , where  $n$  is the size of the bytecode being executed.

The space complexity of vulnerability analysis `VulnerabilityFinder.analyse_only()` is determined by the memory required to store the traces, functions, and other data structures used during the analysis which is typically  $O(m)$ , where  $m$  is the number of traces generated during symbolic execution. Total Space Complexity is  $O(n + k + n + m)$ .

### 4.1.2 Algorithm for Arithmetic Vulnerability

This algorithm provides an overview of the steps involved in analyzing arithmetic vulnerabilities. It focuses on looping over blocks and traces, classifying instructions, and checking for potential integer overflow and underflow vulnerabilities

in arithmetic operations such as addition, multiplication, and subtraction.

---

**Algorithm 2** Algorithm for arithmetic file

---

- 1: Input: Trace, find\_all, solcx, solidity. Output: Detected arithmetic vulnerabilities.
  - 2: Create an empty set called all\_vulns to store all the vulnerabilities found during the analysis.
  - 3: **for** program traces **do**
  - 4:     Skipping any that are reversed (signaling an error or improper execution).
  - 5: **end for**
  - 6: Create an empty set called analyzed\_blocks to keep track of the analyzed blocks to avoid redundancy.
  - 7: Repeat the analysis of the blocks, ignore blocks that have previously undergone analysis.
  - 8: Set a boolean variable was\_mul\_with\_256 to false to track if there was a multiplication operation with the value 256.
  - 9: Set a boolean variable was\_exp\_with\_256 to false to track if there was an exponentiation operation with the value 256.
  - 10: Set a boolean variable next\_trace to false to indicate whether to move to the next trace or not.
  - 11: **for** each instruction **do**
  - 12:     For the ADD and MUL commands, determine whether there is a chance of an integer overflow.
  - 13:     Create a vulnerability object and add it to the list of vulnerabilities if an overflow happens.
  - 14:     For SUB instructions, make sure there isn't a chance of an integer underflow.
  - 15:     If there is an underflow then it will be added to the \_handle\_sub\_underflow.
  - 16:     A possible overflow is indicated by an exponent of 256 for EXP instructions, which should be checked and handled by \_handle\_add\_overflow and \_handle\_mul\_overflow.
  - 17:     Add the current analyzed block to the analyzed\_blocks set.
  - 18: **end for**
  - 19: Add the vulnerabilities to all\_vulns set.
- 

1. Data structures that are used are **Trace** Represents an execution trace including details about the blocks, states, and constraints that were really executed. **Vulnerability** Represents a vulnerability that has been found, along with information on its nature, the block that was analyzed, the offset of the susceptible instruction, the model (values) of the variables at that location, and other pertinent details. The **Z3** library offers this tool, which is used for constraint solving and model extraction. **set** is used to store the analyzed blocks (**analyzed\_blocks**) and the discovered vulnerabilities (**all\_vulns**). For extracting values, maintaining constraints, and handling arithmetic operations, various utility functions and data structures imported from other modules (**sym\_exec.trace**, **sym\_exec.utils**, **vuln\_finder**, **z3**, **z3.z3util**)



2. The **find\_all** flag tells the **arithmetic\_analyse** function whether to find all vulnerabilities or to stop at the first one discovered. It accepts a list of **Trace** objects.
3. To prevent repeated analysis, the function initializes a set **analyzed\_blocks** to keep track of the analyzed blocks and an empty set **all\_vulns** to hold the discovered vulnerabilities.
4. Each **Trace** object in the given list of traces is iterated over.
5. It moves on to the next trace if the trace is reversed.
6. It cycles over the block's instructions for each analyzed block in the trace.
7. If the instruction is an addition or multiplication, it calls the appropriate handler routines, **\_handle\_add\_overflow**, and **\_handle\_mul\_overflow**, to check for potential integer overflow vulnerabilities. These operations take the instruction's parameters and extract them, calculate the operation's result, determine whether an overflow occurred, and retrieve the instruction's parameters.
8. If an overflow is found, a Vulnerability object is created and added to the **all\_vulns** set.
9. If the **find\_all** flag is not set, the function returns the set of vulnerabilities immediately.
10. The **\_handle\_sub\_underflow** function is called if the instruction is a subtraction, which checks for potential integer underflow issues.
11. A Vulnerability object is constructed and added to the **all\_vulns** set if, an underflow is discovered.
12. The function immediately returns the list of vulnerabilities, if the **find\_all** flag is not set.
13. The flag **was\_exp\_with\_256** is set if the instruction is exponentiation and the base (a) is 256 to signal that subsequent subtractions may be connected to a prior multiplication by 256.
14. The currently analyzed block is added to the set of **analyzed\_blocks** after the instructions in it have been examined.

15. The next trace should be processed without looking at the remaining instructions in the current trace if the flag **next\_trace** is set, indicating that the current trace has a specific pattern relating to multiplication or exponentiation with 256. After handling the following trace, the flag is reset.
16. Finally, the function returns the set of all vulnerabilities (**all\_vulns**).

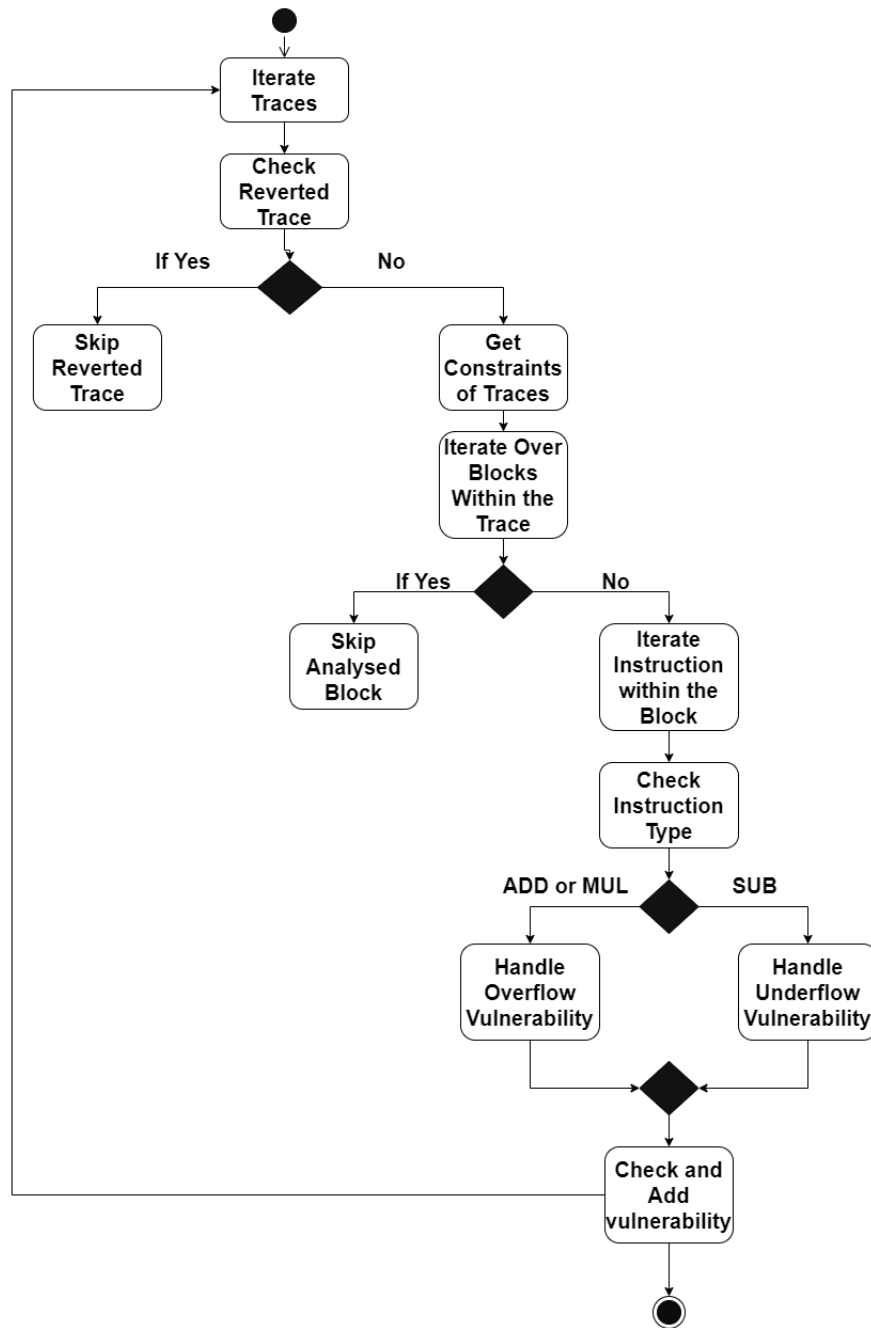


Figure 4.3: Execution of Arithmetic

## Time and Space Complexity

Time complexity depends on the size of the input traces and the number of instructions from each analyzed block. If we consider  $n$  number of traces and  $m$  number of instructions in total. The time complexity of parsing and initializing necessary data structures (`sym_exec.trace`, `sym_exec.utils`, `vuln_finder`, `z3`, `z3.z3util`) is considered to be  $O(1)$ .

The outer loop iterates through the traces, and the nested loop iterates through the analyzed blocks within each trace with the function `Trace`. These results are stored in `(all_vulns)` set in a total of  $O(n * m)$  iterations. Depending on the type of instruction, specific operations are carried out within each analyzed block. Although the complexity of examining each instruction can differ, it typically entails straightforward arithmetic calculations and constraint solving with the Z3 solver library invoked by `arithmetic_analyse` function. The complexity of individual instruction analysis can be considered constant or logarithmic. Total time complexity is  $O(n * m)$ .

The space required to store traces and instructions for analyzed blocks is analyzed by the function `arithmetic_analyse` takes a constant amount of time and is calculated as  $O(n + m)$ . The space complexity of constraints can vary based on the complexity of the analyzed code and the number of instructions. In the worst case, if all instructions generate new constraints, the space complexity can be considered as  $O(m)$ . Assuming each vulnerability object occupies a constant amount of space, then the space complexity is typically considered to be  $O(k)$ , where  $k$  is the number of vulnerabilities found. Total space complexity is given as  $O(n + m + k)$ .

### 4.1.3 Algorithm for Front Running(Time Order Dependency) Vulnerability

In order to locate `CALL` instructions with symbolic storage values, it iteratively searches through traces and analyzed blocks. It defines a constant string `TRANSACTION_ORDERING_DEPENDENCE_TYPE` to represent this vulnerability type. The algorithm includes a function to extract symbolic storage values from `CALL` instructions and another function called `tod_analyse` to analyze traces and detect vulnerabilities. It keeps track of processed blocks, and interesting storage values, and stores the found vulnerabilities in sets and lists for further analysis. The al-

gorithm evaluates each saved symbolic storage value against storage states from different traces to identify potential vulnerabilities.

---

**Algorithm 3** Algorithm for time Order Dependence File

---

- 1: Input: Trace, find\_all, solcx, solidity.
  - 2: Output: Detected timeorder dependence vulnerabilities.
  - 3: Import the necessary functions and classes.
  - 4: Define constant tuple CALL\_WITH\_VALUES that involves values for the CALL instructions.
  - 5: Define a constant string TRANSACTION\_ORDERING\_DEPENDENCE\_TYPE to represent the type of vulnerability as "Transaction Ordering Dependence".
  - 6: Define a function \_get\_symbolic\_storage\_value\_of\_call to take a CALL instruction and retrieve the symbolic storage value from it.
  - 7: Define the function tod\_analyse that takes a list of traces and a flag to determine if all vulnerabilities should be found or only the first one.
  - 8: Create an empty set called all\_vulns to store all the vulnerabilities found during the analysis.
  - 9: Create an empty set called analyzed\_blocks to keep track of the analyzed blocks to avoid redundancy.
  - 10: Create an empty list called interesting\_values\_in\_call to store the interesting storage values encountered during call instructions. Create an empty list called interesting\_storages to store the storage values encountered during the analysis.
  - 11: **for** each trace **do**
  - 12:     Skip traces with a reversed label.
  - 13:     Iterate over each block in the trace that has been analyzed.
  - 14:     Ignore blocks that have previously been handled.
  - 15:     Repeat the block's instructions one after another.
  - 16:     Verify if the command is a CALL instruction with values are stored in interesting\_values\_in\_call().
  - 17:     If so, obtain and extract the symbolic storage value.
  - 18:     Include the examined block among the processed blocks.
  - 19:     Save the trace's storage state in interesting\_storages().
  - 20:     **for** each symbolic storage value that has been saved **do**
  - 21:         Evaluate each value against the storage states gleaned from various traces.
  - 22:     Return the list of vulnerabilities to all\_vulns set.
- 

1. **Trace** represents an execution trace including details about the blocks, states, and constraints that were really executed. **Vulnerability** represents a vulnerability that has been discovered, together with information on its nature, the analyzed block, the offset of the vulnerable instruction, and other pertinent details. The names of the instructions that use calls with values are listed in the tuple **CALL\_WITH\_VALUES**. The vulnerability type, "Transaction Ordering Dependence", is represented by the variable **TRANSACTION\_ORDERING\_DEPENDENCE\_TYPE**. **set** is used to store the analyzed blocks (**analyzed\_blocks**) and the discovered vulnerabilities (**all\_vulns**).

The list is used to keep track of the interesting storage values accessible during calls and the interesting storage values from various traces. The data structure **dict** is used to hold interesting\_values\_in\_call's storage value together with the block and instruction information that goes with it. Various utility functions and data structures (such as **vuln\_finder.vulnerability** and **sym\_exec.utils**) imported from other modules are used.

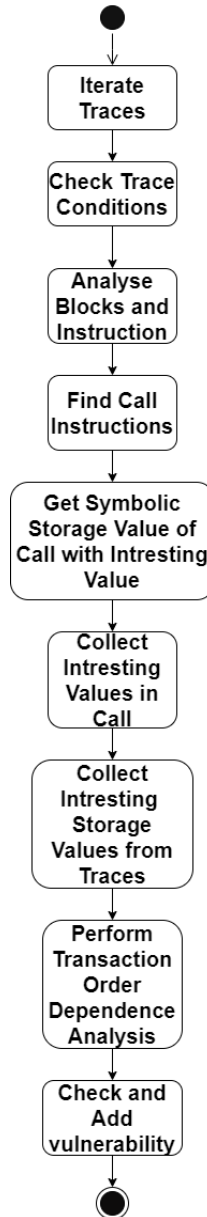


Figure 4.4: Execution of Front Running

2. The flag **find\_all** tells the **tod\_analyse** function whether to find all vulnerabilities or to stop at the first one discovered. It accepts a list of **Trace** objects.

3. The function initializes an empty list **interesting\_values\_in\_call** to store the interesting storage values accessed in calls, an empty list **interesting\_storages** to store the interesting storage values from various traces, an empty list **interesting\_values\_in\_call** to track the interesting storage values analyzed in calls, and an empty set **all\_vulns** to store the found vulnerabilities.
4. Each **Trace** object in the given list of traces is iterated over.
5. It moves on to the next trace if the trace is reversed.
6. It cycles over the block's instructions for each analyzed block in the trace.
7. The `_get_symbolic_storage_value_of_call` function is called if the instruction is a call (CALL or CALLCODE) to retrieve the symbolic storage value retrieved during the call. If a storage value is discovered, it and the accompanying block and instruction are added to the **interesting\_values\_in\_call** list.
8. The **analyzed\_blocks** set receives the newly added analyzed block.
9. The **interesting\_storages** list is expanded to include the storage values from the current trace.
10. The function then checks for vulnerabilities related to transaction ordering dependencies after examining all traces.
11. It contrasts the value with the matching storage values from various traces (**interesting\_storages**) for each storage value received from a call (**interesting\_values\_in\_call**).
12. If a different storage value is found for the same storage position, indicating a potential transaction ordering dependence vulnerability, a vulnerability object is created and added to the **all\_vulns** set.
13. The function immediately returns the list of vulnerabilities if the **find\_all\_flag** is not set.
14. Finally, the function returns the set of all vulnerabilities (**all\_vulns**).

### Time and Space Complexity

The time complexity of initializing the data structures takes a constant amount of time. The outer loop iterates over the traces with the function **Trace**, which

has a time complexity of  $O(n)$ , where  $n$  is the number of traces. Cycling over the block's instructions in this step, let's assume there are  $m$  instructions in each block. The time complexity of this step is  $O(m)$ . Checking vulnerabilities and comparing storage values depends on the size of the **interesting\_values\_in\_call** list and the **interesting\_storages** list. Here let us assume that there are  $k$  values in **interesting\_values\_in\_call** and  $l$  values in **interesting\_storages**. The worst-case time complexity of this step is  $O(k * l)$ . The overall time complexity can be given as  $O(n * m * k * l)$ .

The space complexity of this algorithm is  $O(n)$  uses sets and lists to store analyzed blocks, interesting values in the call, interesting storages, and vulnerabilities. The total space complexity is computed to  $O(n)$

#### 4.1.4 Algorithm for Reentrancy Vulnerability

---

**Algorithm 4** Algorithm for Reentrancy File

---

- 1: Input: Trace, find\_all, solcx, solidity.
- 2: Output: Detected reentrancy vulnerabilities
- 3: Define a constant string REENTRANCY\_TYPE to represent the type of vulnerability as "Reentrancy".
- 4: Define constant strings CALL\_INSTRUCTION and SSTORE\_INSTRUCTION to represent the instructions' names "CALL" and "SSTORE".
- 5: Define a helper function \_find\_instruction, \_get\_storage\_position, \_get\_storage\_var, \_get\_storage\_position,
- 6: Create an empty set called "all\_vulns" to hold the discovered flaws.
- 7: **for** each trace **do**
- 8:     Skip trace states that have been reversed.
- 9:     Go through each trace's analyzed blocks one more time.
- 10:     Repeat the block's directives iteratively.
- 11:     Move on to the following instruction if the current instruction is not CALL.
- 12:     Analyse Constraints after the CALL command.
- 13:     If vulnerabilities are present, use a \_get\_solver function to look for vulnerabilities.
- 14:     Add any discovered vulnerabilities to the "all\_vulns" list.
- 15:     Track down SSTORE instructions that follow the CALL command.
- 16:     Analyse Constraints prior to the CALL command.
- 17:     If vulnerabilities are present, use a \_get\_solver to look for vulnerabilities.
- 18:     Add any discovered vulnerabilities to the "all\_vulns" list.
- 19:     Add the current analyzed block to the analyzed\_blocks set.
- 20: **end for**
- 21: Return the list of vulnerabilities.

---

This algorithm checks for **CALL** instructions and analyzes constraints before and after the **CALL** as you iteratively go through traces and analyzed blocks. On

the basis of the identified constraints, use a solver to look for vulnerabilities and send back a list of discovered vulnerabilities.

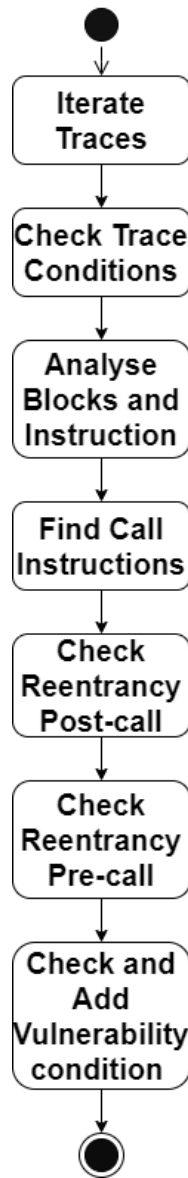


Figure 4.5: Execution of Reentrancy

1. **Trace** represents an execution trace including details about the blocks, states, and constraints that were really executed. **Vulnerability** Represents a vulnerability that has been discovered, together with information on its nature, the analyzed block, the offset of the vulnerable instruction, and other pertinent details.

A basic block in the control flow graph is represented by the **SSABasicBlock** class, together with any associated instructions and constraints. Check the satisfiability of constraints using a **Z3** solver. **set** is used to store the analyzed



blocks (**analyzed\_blocks**) and the discovered vulnerabilities (**all\_vulns**). the **list** is used to keep track of the interesting storage values accessible during calls and the interesting storage values from various traces. **dict** is used to hold interesting\_values\_in\_call's storage value together with the block and instruction information that goes with it. Various utility functions and data structures (such as **vuln\_finder.vulnerability** and **sym\_exec.utils**) imported from other modules are used.

2. The flag **find\_all** tells the **reentrancy\_analyse** function whether to find all vulnerabilities or to stop at the first one discovered and accepts a list of **Trace** objects.
3. The function initializes an empty set **all\_vulns** to store the discovered vulnerabilities, a boolean variable **block\_analyzed** to store the most recent analyzed block, a boolean variable **exist\_constraints** to track the existence of constraints, a boolean variable **offset** to track the offset of the vulnerable instruction, and a boolean variable **instruction\_offset** to track its offset.
4. The function iterates over each **Trace** object in the provided list of traces.
5. It moves on to the next trace if the trace is reversed.
6. It determines if a block has already been analyzed for each one that is present in the trace of analysis. If so, it moves on to the following block being examined.
7. It determines whether each instruction in the block being analyzed is a **CALL** instruction. If not, it moves to the next instruction.
8. To check for potential flaws associated with reentrant calls made before the weak **CALL** instruction, the **\_reentrancy\_pos\_call** function is called.
9. The **\_has\_vulnerability** function is used to determine whether restrictions result in a vulnerability if constraints are discovered. A vulnerability object is produced and added to the **all\_vulns** set if a vulnerability is discovered. The function immediately returns the list of vulnerabilities if the **find\_all** flag is not set.
10. The analyzed block is added to the **analyzed\_blocks** set.
11. If constraints were analyzed but no vulnerabilities were found, a vulnerability is assumed to exist, and a vulnerability object is created based on the

last analyzed block, offset, and instruction offset. The object is added to the `all_vulns` set. If the `find_all` flag is not set, the function returns the set of vulnerabilities immediately.

12. Repeat steps 4 to 11.

13. Finally, the function returns the set of all vulnerabilities (**`all_vulns`**).

### Time and Space Complexity

The first step for initializing the data structure takes  $O(1)$ . The function **Trace** function iterates over the given traces list, which contains  $n$  elements computes to a complexity of  $O(n)$ . For each block for reentrancy vulnerability The time complexity of this step depends on the number of analyzed blocks (**`analyzed_blocks`**) it iterates over the analyzed blocks set, which contains at most  $m$  elements (where  $m$  is the number of unique analyzed blocks across all traces). The worst time complexity is given as  $O(m)$ .

Now for each **CALL** instruction extract values from instructions, create constraints, solve the constraints using a solver, etc. The time complexity of this step is  $O(k)$  assuming there are  $k$  instructions in each block. Adding the analyzed block to the **`analyzed_blocks set`** will take a time complexity of this step depending on the implementation of the **`set`** data structure. Assuming an average constant time complexity, it can be considered as  $O(1)$  Total time complexity is  $O(n * m * k)$ .

The space complexity depends on the number of unique analyzed blocks ( $m$ ), the number of vulnerabilities found ( $v$ ), and the maximum size of the constraints and other data structures.

Total space complexity is  $O(m + v + k)$ .

### 4.1.5 Algorithm for Time Manipulation Vulnerability

To briefly describe the algorithm it repeats the analysis of the blocks and traces. Examine the analyzed block to see if any restrictions or return values are based on the timestamp variable. Create a Vulnerability object and include it in the list of vulnerabilities discovered if a time manipulation vulnerability is discovered and gives the detected vulnerabilities. It defines constants like "TIME\_MANIPULATION\_VAR" (e.g., "timestamp") and "TIME\_MANIPULATION\_TYPE" (e.g., "TIME MANIPULATION"). The algorithm iterates over each block and trace, examining if any restrictions or return data depend on the "TIME\_MANIPULATION\_VAR." If a

---

**Algorithm 5** Algorithm for Time Manipulation File

---

- 1: Input: Traces, find\_all, solidity, solcx.
  - 2: Output: Detected time manipulation vulnerabilities.
  - 3: Define constants like "TIME\_MANIPULATION\_VAR"(eg."timestamp") and "TIME\_MANIPULATION\_TYPE"(eg. TIME MANIPULATION)
  - 4: **for** each block and trace **do**
  - 5:   Examine each trace to see whether any restrictions or return data are depending on the variable "TIME\_MANIPULATION\_VAR." If a vulnerability of type 'TIME\_MANIPULATION\_TYPE' is discovered, it should be included along with the related object of type 'Analyzed Block' and instruction offset.
  - 6:   Iterate over the instructions for each analyzed block and look for special instructions that deal with time manipulation (such as SHA3, SSTORE, etc.). Add a vulnerability of type 'TIME\_MANIPULATION\_TYPE' associated with the analyzed block and instruction offset if any of these instructions use arguments depending on the variable 'TIME\_MANIPULATION\_VAR'.
  - 7: **end for**
  - 8: Return the list of vulnerabilities.
- 

"Time Manipulation" vulnerability is detected, it includes the related object, the analyzed block, and the instruction offset. The method analyses SHA3 and SSTORE operation limitations, return values, and instructions to find temporal manipulation flaws.

1. These data structures are used in this algorithm, **Trace** represents an execution trace including details about the blocks, states, and constraints that were really executed. **Vulnerability** gives a vulnerability that has been discovered, together with information on its nature, the analyzed block, the offset of the vulnerable instruction, and other pertinent details. **set** is used to store the analyzed blocks (**analyzed\_blocks**) and the discovered vulnerabilities (**all\_vulns**). Various utility functions and data structures for vulnerability finding (**vuln\_finder.vulnerability** and **sym\_exec.utils**) imported from other modules are used.
2. The flag **find\_all** tells the **time\_manipulation\_analyse** function whether to find all vulnerabilities or to stop at the first one discovered and accepts a list of **Trace** objects.
3. The function initializes an empty set **all\_vulns** to store the discovered vulnerabilities, a boolean variable **block\_analyzed** to store the most recent analyzed block.
4. The function iterates over each **Trace** object in the provided list of traces.
5. It moves on to the next trace if the trace is reversed.

6. To determine whether any restrictions in the trace or the return data are based on the timestamp variable, `_analyse_return_value_and_constraints` function is used.
7. If vulnerabilities are found, they are added to the `all_vulns` set. If the `find_all` flag is not set, the function returns the set of vulnerabilities immediately.
8. For each analyzed block in the trace, it checks if the block has already been analyzed. If so, it continues to the next analyzed block.

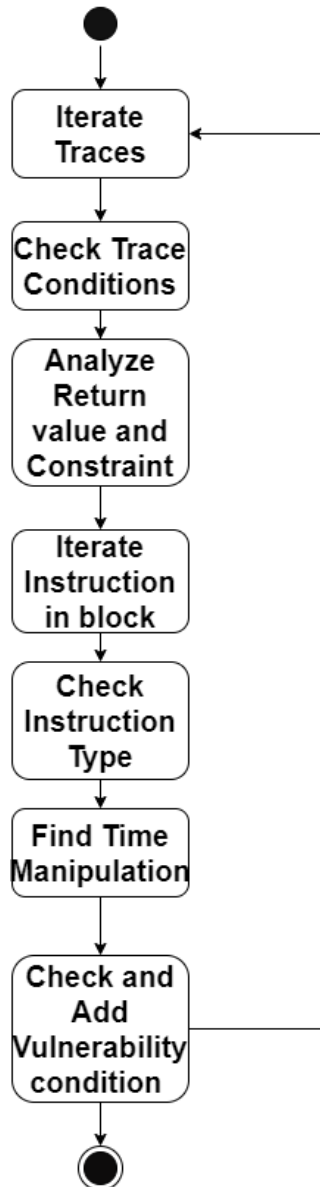


Figure 4.6: Execution of Time Manipulation

9. For each instruction in the analyzed block, it checks the instruction name.

10. If the instruction is a **SHA3** instruction, then the **analyze\_sha3** function is invoked to check for any potential time manipulation issues. A vulnerability is added to the **all\_vulns** set if it is discovered. The function immediately returns the list of vulnerabilities if the **find\_all** flag is not set.
11. The **analyze\_sstore** function is called to examine the instruction for potential time-related vulnerabilities if it is an SSTORE instruction. A vulnerability is added to the **all\_vulns** set if it is discovered. The function immediately returns the list of vulnerabilities if the **find\_all** flag is not set.
12. The analyzed block is added to the analyzed\_blocks set.
13. Repeat steps 8 to 12.
14. Finally, the function returns the set of all vulnerabilities (**all\_vulns**).

### Time and Space Complexity

Initializing the data structure will take a constant amount of time. The function iterates over the given traces list, which contains  $n$  elements. For each trace, it performs the following operations, Calls the analyze return value and constraints function, which checks constraints and return values for time manipulation vulnerabilities. Iterates over the analyzed blocks set, which contains at most  $m$  elements (where  $m$  is the number of unique analyzed blocks across all traces) and finally stores the discovered vulnerabilities to **all\_vulns** set which is given by the time complexity of  $O(m)$  For each analyzed block, iterates over the block's instructions and calls either the **analyze\_sha3** or **analyze\_sstore** function, depending on the instruction for  $k$  vulnerabilities the time complexity is given as  $O(k)$ .

Total time complexity is  **$O(n * (m + k))$** .

The function maintains several data structures, including sets (analyzed blocks, all vulns), lists (vulns), and individual variables (offset, length, etc.). The space complexity depends on the number of unique analyzed blocks ( $m$ ), the number of vulnerabilities found ( $v$ ), and the maximum size of the constraints and other data structures.

Total space complexity is  **$O(m + v + k)$** .

### 4.1.6 Algorithm for Unchecked Low Level Calls Vulnerability

It starts by iterating over the traces, then checks the conditions for each trace to proceed with the analysis. Within each trace, it iterates over the analyzed blocks

and instructions, checks the instruction type, handles the low-level call, checks the remaining constraints, and determines if a vulnerability exists. If a vulnerability

---

**Algorithm 6** Algorithm for Unchecked Low Level Calls File

---

- 1: Input: Traces, find\_all, solidity, solcx.
  - 2: Output: Detected unchecked low level calls vulnerabilities.
  - 3: Define constants such as 'UNCHECKED\_LL\_CALL\_TYPE' (e.g., "Unchecked Low Level Call") and 'instructions\_to\_check' (e.g., ('CALL', 'CALLCODE', 'DELEGATECALL', 'STATICCALL')).
  - 4: Go through the traces and analyzed blocks.
  - 5: If the state is reversed or the depth exceeds the maximum permitted depth, skip that trace.
  - 6: **for** every analyzed block. **do**
  - 7:   Verify whether there are any leftover restrictions that were not examined in the block if the instruction is one of the instructions to check (based on "instructions\_to\_check").
  - 8:   Add a vulnerability of type 'UNCHECKED\_LL\_CALL\_TYPE' linked to the analyzed block and instruction offset if there are no more restrictions.
  - 9:   If there are still limitations, handle the low-level call by determining if any of them pertain to the call's return value. Add a vulnerability of type associated with the analyzed block 'UNCHECKED\_LL\_CALL\_TYPE' and instruction offset if there are no restrictions on the return value.
  - 10:   Add the analyzed block to the set of analyzed blocks.
  - 11: **end for**
  - 12: Return the set of detected vulnerabilities.
- 

is found, it is added to the set of vulnerabilities. The algorithm repeats these steps until all traces and blocks have been analyzed. Finally, it returns the set of vulnerabilities.

1. The data structures associated with the algorithm are **Trace** Represents an execution trace including details about the blocks, states, and constraints that were really executed. **Vulnerability** Represents a vulnerability that has been discovered, together with information on its nature, the analyzed block, the offset of the vulnerable instruction, and other pertinent details. **set** is used to store the analyzed blocks (**analyzed\_blocks**) and the discovered vulnerabilities (**all\_vulns**). Various utility functions and data structures for vulnerability finding (such as **vuln\_finder.vulnerability** and **sym\_exec.utils**) imported from other modules are used.
2. The flag **find\_all** tells the **unchecked\_low\_level\_calls\_analyse** function whether to find all vulnerabilities or to stop at the first one discovered and accepts a list of **Trace** objects.
3. The function initializes an empty set **all\_vulns** to store the discovered vul-

nerabilities, a boolean variable `block_analyzed` to store the most recent analyzed block.

4. The function iterates over each Trace object in the provided list of traces.

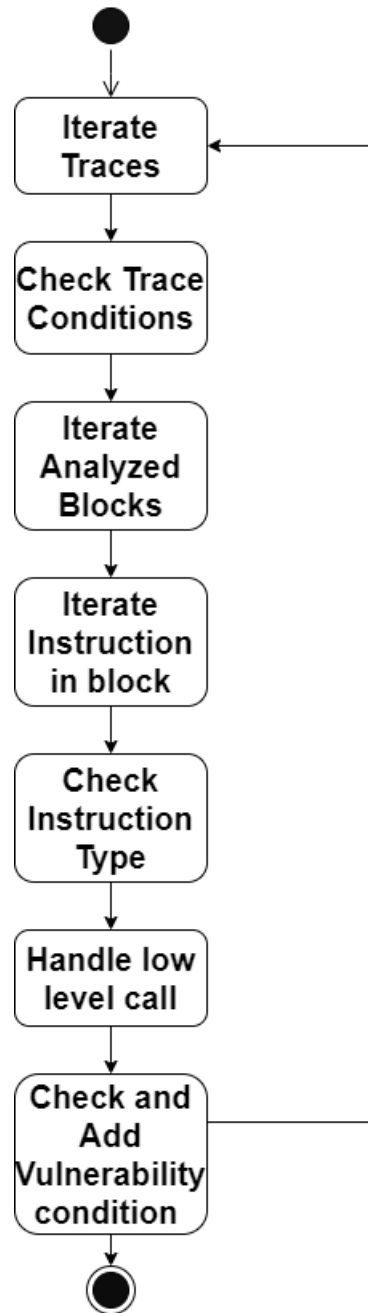


Figure 4.7: Execution of ULL

5. It moves on to the next trace if the trace is reversed.
6. It determines if the name of each instruction in the analysed block is one of the low-level calls that needs to be examined.

7. If the instruction matches one of the low-level calls, the remaining constraints in the trace that have not yet been analysed are found by extracting the analysed constraints from the analysed block.
8. A vulnerability object is created and added to the **all\_vulns** set if there are no more constraints, which denotes an unchecked lowlevel call vulnerability. The function immediately returns the list of vulnerabilities if the **find\_all** flag is not set.
9. The `_handle_low_level_call` function is called to examine whether the return value of the low-level call was appropriately checked if there are any remaining limitations. The function immediately returns the list of vulnerabilities if the **find\_all** flag is not set.
10. The analyzed block is added to the `analyzed_blocks` set.
11. Repeat steps 6-10.
12. Finally, the function returns the set of all vulnerabilities (**all\_vulns**).

### Time and Space Complexity

The function iterates over the given traces list, which has a size of  $n$ . This loop has a linear time complexity of  $O(n)$ . Within each trace, the function iterates over the analyzed blocks of the trace, which has a size of  $m$  (the number of analyzed blocks in the trace). This loop also has a linear time complexity of  $O(m)$ . For each analyzed block, the function iterates over the instructions of the block. The number of instructions per block is typically small and can be considered constant, denoted as  $k$ . Inside the instruction loop, there is a check to determine if the instruction name is in the instructions to check, which has a constant lookup time.

Total Time Complexity is given as  **$O(n * m * k)$** .

The function uses several data structures to store information, including sets to store vulnerabilities (`all_vulns`), analyzed blocks (`analyzed_blocks`), and remaining constraints. The space complexity of these data structures depends on the number of vulnerabilities found and the size of the analyzed blocks and constraints.

Total Space Complexity is  **$O(v)$** .

## 4.2 Overall Complexity

The complexities of five modules along with a master file can be seen in table 4.1, here the argument sizes, input file size, data structures, and other parameters are



independent of particular files.

Table 4.1: Comparison of Various Files

Name of File	Total Time Complexity	Total Space Complexity
Master File	$O(n + k + n^2 + m * n)$	$O(n + k + n + m)$
Arithmetic File	$O(n * m)$	$O(n+m+k)$
Front Running File	$O(n * m * k^1)$	$O(n)$
Reentrancy File	$O(n * m * k)$	$(m + v + k)$
Time Manipulation File	$O(n * (m + k))$	$O(m + v + k)$
ULL File	$O(n * m * k)$	$O(v)$

### 4.2.1 How Complexities Can be Improved Overall

- If the goal is to find a single vulnerability, terminating the analysis and return it as soon as a vulnerability is found. This can save unnecessary iterations and improve the overall time complexity.
- By set a maximum depth for the symbolic execution analysis to prevent analyzing traces that exceed a certain depth. This can help control the time and space complexity by avoiding excessively deep traces.
- Parallelizing the analysis of multiple traces or analyzed blocks. This can leverage multi-core systems and potentially reduce the overall execution time.
- Reviewing the data structures used in the function and ensuring that only the essential information to store for analysis. By avoiding redundant or excessive storage of intermediate results, and release any resources or data structures that are no longer required in this way space complexity can be reduced.

## 4.3 Association of Files

As shown in Fig 4.8 we run a Solidity file on the master file which invokes the vul\_finder script which has five Python scripts associated with it, those five after processing the output will be displayed to the Command line interface.

For example, let us take the reentrancy vulnerability, first of all, the command `$ python3 master.py -vt reentrancy -s example2.sol`, this command executes the

Python script file of **master.py** with vulnerability reentrancy and the file name here is **example2.sol**, the **master.py** is a master file that calls vulnerability finder file, where different Python scripts are written for different vulnerabilities. From this **reentrancy.py** is invoked and this script gives the name of the vulnerability, its location and finally, it gives the time taken to run a **.sol** file. In the same way, other modules are also run. For finding and vulnerability we must need a master file. whereas the vulnerability scripts are independent of each other.

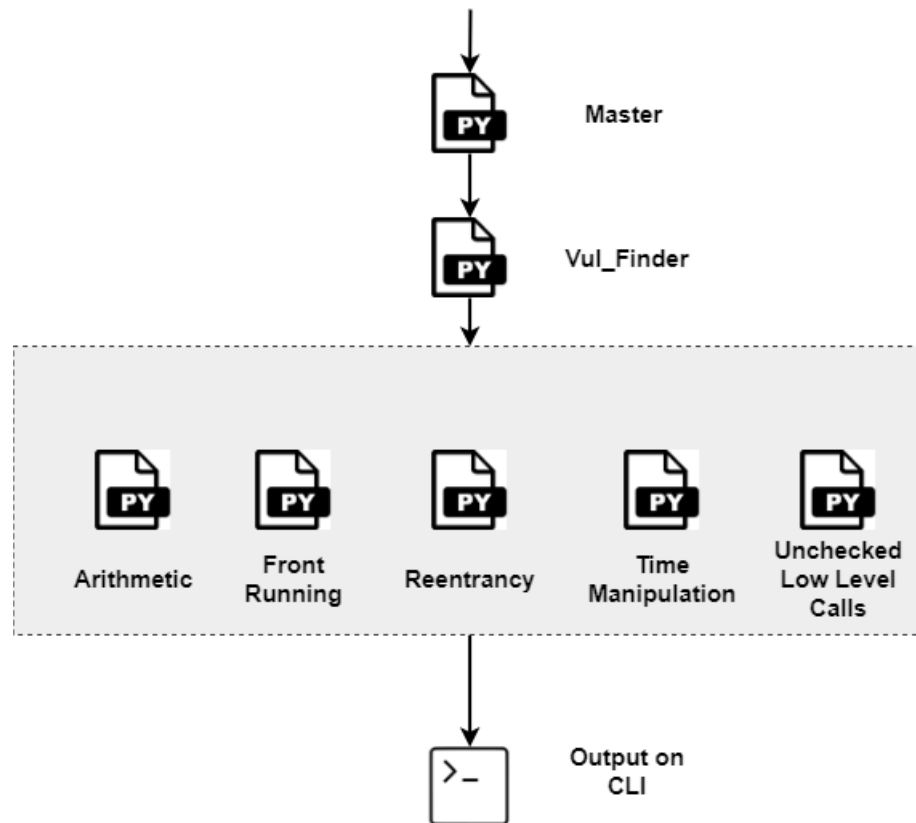


Figure 4.8: Flow of Execution

## 4.4 Dependencies

Python3 is used to implement the five modules in our study, as well as the Python libraries and packages indicated below:

- pyevmasm is a Python package that provides a set of tools for working with the EVM. It allows you to disassemble, assemble, and analyze EVM bytecode.
- py-solc-x is a Python package that provides an interface to the Solidity compiler. It allows you to compile Solidity source code to EVM bytecode.

- Z3 solver is a Python package that provides an interface to the Z3 theorem prover. Z3 is an open-source theorem prover that is designed for automated reasoning about software and hardware systems.
- Solidity parser is a Python package that provides a parser for Solidity source code. It allows you to generate an Abstract Syntax Tree (AST) of a Solidity program, which can be used for further analysis and transformation.

## 4.5 Chapter Summary

In this chapter, we discussed the system architecture, the master file along with the five modules that we implemented. Here we even emphasize the time and space Complexity of all the Algorithms and how they can be improved, also gives the association and close working of all the modules together.

## CHAPTER 5

# Results

In this chapter, we compare our method and other static tools, for analysis of tools we have a few choices, the conventional approach, where each tool produces its own dataset by downloading contracts from [27] between specified dates, runs all tools on it, and then presents the results. Another way could be getting in touch with the authors and asking for the datasets. However, we believe that this is not the best course of action, thus we use a publicly accessible framework to make sure that the dataset is comparable across all tools. We swiftly adapted our strategy where we decided to use the available Smartbugs dataset [11] and started conducting analysis using 9 different tools. The SB curated [11] and SB wild [9] databases are both part of SmartBugs. Since the vulnerabilities were manually annotated. Firstly we analyzed all the tools on the smaller dataset and then on the larger one.

### 5.1 Methodology

The selection of all the tools was based on the following factors:

1. Open source availability of the tools.
2. Python environment is used in all the tools.
3. The input format of the tools is either the Solidity file or Byte code.
4. All the tools were either using system execution or IR techniques.

The methodology that is followed:

1. Configure the tools with the necessary environment and parameters, and execute the tools on the dataset to perform vulnerability analysis.
2. Extraction of vulnerabilities to Text files or JSON files as compatible with different tools.
3. Review each detected vulnerability and determine its appropriate category based on its characteristics and impact.

4. Determine the coverage and effectiveness of the tools by examining which vulnerability category is detected and to what extent.

## 5.2 Datasets

### 5.2.1 SB curated Dataset Specifications

According to Smartbugs [11] this dataset contained 69 smart contracts out of which 115 labeled vulnerabilities were present the main motive to use this dataset was the contracts here were real contracts or vulnerable annotated contracts and the vulnerabilities were labeled with the line numbers and categories.

The objective of this dataset is to have a collection of contracts that are known to be vulnerable, each annotated with the location and type of vulnerability. This dataset can be used to assess how well smart contract analysis tools are at spotting weaknesses. However, after reviewing the SmartBugs dataset, we came to the conclusion that there are certain vulnerabilities that exist but are not documented. We carefully reviewed the contracts and discovered that, to our knowledge, there are a total of 124 vulnerabilities. Although this number may rise, we took into account 124 vulnerabilities when estimating the tools.

Table 5.1: Categories of Vulnerability Present in Dataset

<b>Vulnerability Type</b>	<b>No. of Contracts</b>	<b>No. of Vulnerabilities</b>
Access Control	17	24
Arithmetic	14	23
Denial of service	8	14
Front running	4	7
Reentrancy	7	34
Time manipulation	5	7
Unchecked low levels	11	10
others	3	5
Total	69	124

### 5.2.2 SB wild Dataset Specifications

This dataset from SmartBugs [9] was a big dataset with nearly 47k smart contracts fetched from Ethereum blockchain using Etherscan, for this dataset the set of vul-

nerabilities were unknown unlike the SB curated dataset(refer section5.2.1) However, this information may be used to find actual contracts that contain vulnerabilities and have an indicator of how frequently a certain issue occurs. Additionally, it may be used to compare analytic tools using measures like performance, true positive, false positive, true negative, false negative, accuracy, etc.

### 5.3 Experimental Setup

For this experiment we followed two approaches on the smaller dataset SB curated where the first observation was made in a manner where we set a time budget of 30 minutes for each tool, if the allotted time has passed, we halt the execution and gather the execution’s partial results. The second way was running the tools to their complete execution time and the results were recorded. multiple files were run parallelly to gather the results for different vulnerabilities and batch processing was also done, but as this data set was small so we didn’t take that into account. An example of single processing where the time for running the contract, line number, and type of vulnerability can be examined as seen in Fig 5.1.

```
(venv) daict@admin:~/Downloads/conkas-master$ time python3 master.py -vt reentrancy -s BadContract.sol
Analysing BadContract.sol:Authorization...
Analysing BadContract.sol:Ballv...
Nothing to analyse
Analysing BadContract.sol:SafeMath...
Analysing BadContract.sol:StandardToken...
Vulnerability: Reentrancy. Maybe in function: approve(address,uint256). PC: 0xbf8. Line number: 219.
Analysing BadContract.sol:Token...
Nothing to analyse
Analysing BadContract.sol:TokenFactory...
PHI instruction need arguments but 0 was given

Traceback (most recent call last):
  File "conkas.py", line 110, in main
    traces = sym_exec.execute()
  File "/home/daict/Downloads/conkas-master/sym_exec/symbolic_executor.py", line 51, in execute
    new_traces = self.__sym_exec_traces(traces_to_execute)
  File "/home/daict/Downloads/conkas-master/sym_exec/symbolic_executor.py", line 64, in __sym_exec_traces
    new_blocks = self.__sym_exec_block(block_to_analyse, trace.state)
  File "/home/daict/Downloads/conkas-master/sym_exec/symbolic_executor.py", line 93, in __sym_exec_block
    new_blocks = self.__sym_exec_instruction(instruction, state)
  File "/home/daict/Downloads/conkas-master/sym_exec/symbolic_executor.py", line 116, in __sym_exec_instruction
    return func(instruction, state)
  File "/home/daict/Downloads/conkas-master/sym_exec/instructions/rattle_instructions.py", line 58, in inst_phi
    raise Exception
Exception
Analysing BadContract.sol:TokenRecipient...
Nothing to analyse
Analysing BadContract.sol:XPAAssetToken...

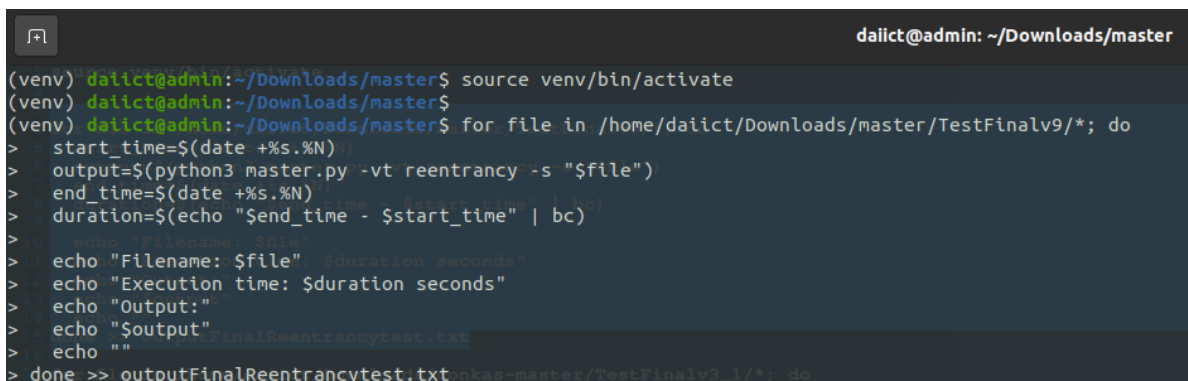
real    0m9.987s
user    0m9.880s
sys     0m0.112s
(venv) daict@admin:~/Downloads/conkas-master$
```

Figure 5.1: Command Line Interface for Single Processing

From the bigger dataset of 47k contracts we randomly chose around 3k smart contracts to be precise 3031 smart contracts are audited on different tools, for our approach we divided these 3k contracts into 10 files of different sizes which con-

tained variable numbers of smart contracts as we were limited with the system. For running these smart contracts on different tools we used to run a particular file of size, suppose a file which contains 400 smart contracts which will be run on all the five types of vulnerability with a simple loop command which can be seen in Fig 5.2.

Here the command is a loop where every .sol file in the directory is iterated over a loop. It keeps track of the start time for every file before running the main script. The current file is used as input and the master script is executed with the settings supplied ("-vt reentrancy -s"). The end time is recorded following the script execution. By deducting the start time from the finish time, it determines the length of the script execution. The script output and the execution time are collected in the text file and can be seen in Fig 5.3.

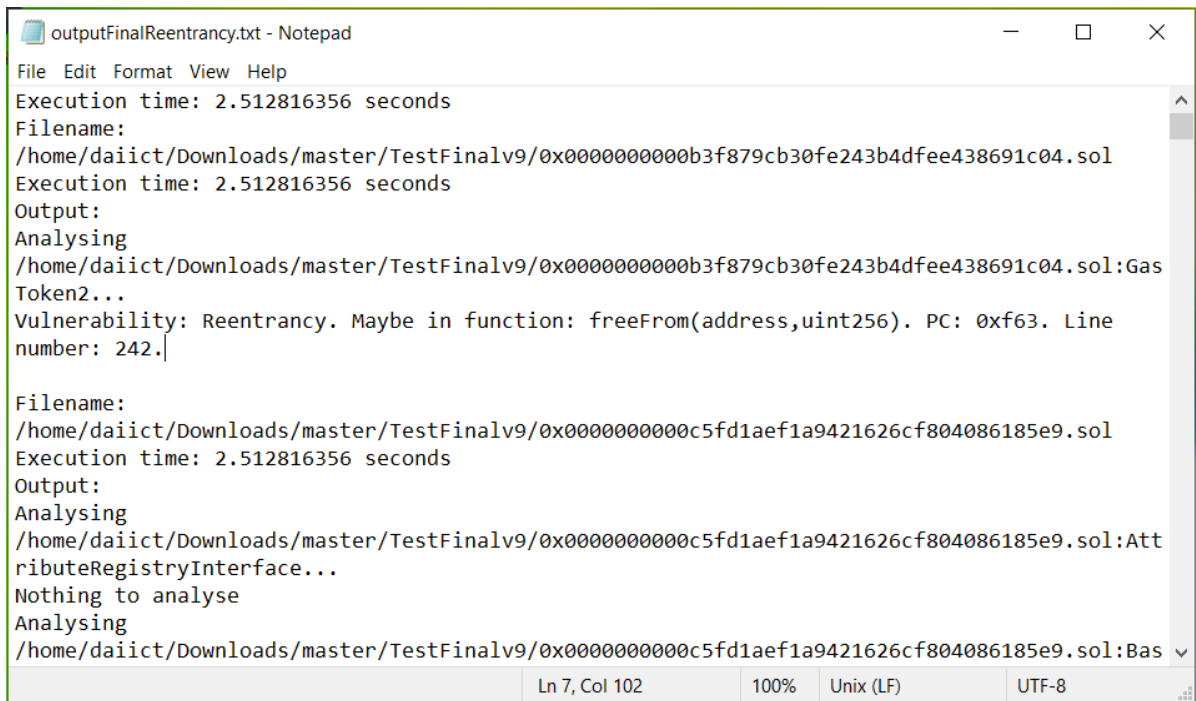


```
daict@admin: ~/Downloads/master
(venv) daict@admin:~/Downloads/master$ source venv/bin/activate
(venv) daict@admin:~/Downloads/master$
(venv) daict@admin:~/Downloads/master$ for file in /home/daict/Downloads/master/TestFinalv9/*; do
> start_time=$(date +%s.%N)
> output=$(python3 master.py -vt reentrancy -s "$file")
> end_time=$(date +%s.%N)
> duration=$(echo "$end_time - $start_time" | bc)
> echo "Filename: $file      duration: $duration seconds"
> echo "Execution time: $duration seconds"
> echo "Output:"
> echo "$output"
> echo ""
done >> outputFinalReentrancytest.txt
```

Figure 5.2: Command Line Interface for Batch Processing

## 5.4 Experimental Results

The results of our study focus on the true positive rate and performance of all the tools, we adopted the approach of the paper Smartbugs [11]. Here the authors are considering the 9 best tools that were run on a dataset containing 69 smart contracts with 115 tagged vulnerabilities(SB curated), they also mentioned that the vulnerability count can be varied as they may not be able to detect. Whereas in our study we considered 8 tools along with our approach where we ran all the tools on the same dataset but while running the tools we found out that few of the tools were able to detect a few vulnerabilities at the same location which we took into account for evaluating tools also for those vulnerabilities, so after sufficient



```
outputFinalReentrancy.txt - Notepad
File Edit Format View Help
Execution time: 2.512816356 seconds
Filename:
/home/daiict/Downloads/master/TestFinalv9/0x000000000b3f879cb30fe243b4dfee438691c04.sol
Execution time: 2.512816356 seconds
Output:
Analysing
/home/daiict/Downloads/master/TestFinalv9/0x000000000b3f879cb30fe243b4dfee438691c04.sol:Gas
Token2...
Vulnerability: Reentrancy. Maybe in function: freeFrom(address,uint256). PC: 0xf63. Line
number: 242.

Filename:
/home/daiict/Downloads/master/TestFinalv9/0x000000000c5fd1aef1a9421626cf804086185e9.sol
Execution time: 2.512816356 seconds
Output:
Analysing
/home/daiict/Downloads/master/TestFinalv9/0x000000000c5fd1aef1a9421626cf804086185e9.sol:Att
ributeRegistryInterface...
Nothing to analyse
Analysing
/home/daiict/Downloads/master/TestFinalv9/0x000000000c5fd1aef1a9421626cf804086185e9.sol:Bas
Ln 7, Col 102    100%    Unix (LF)    UTF-8
```

Figure 5.3: Text File Output

attempts we came to this numbers that this dataset contained 69 smart contracts and 124 vulnerabilities and the number may rise. In this particular study, we are only focusing on five types of vulnerabilities as of now, whereas in the paper [11] nine types of vulnerabilities are considered.

About the larger dataset as mentioned in the paper [9] the authors analyzed 47,518 contracts that took them approximately 564 days and 3 hours to run in total with all the 9 tools they considered.

Here, we randomly chose 3031 smart contracts from these 47k smart contracts. In our study we were not focusing on the quantity but quality as we wanted to examine how our approach is performing with respect to other tools so after considering this set of smart contracts we run them on all 8 tools along with our approach which took us roughly 25 to 26 days considering tool setup time also as we were limited to only a single machine.



## 5.4.1 Effectiveness of Tools

### Results of SB curated Dataset

A true positive identifies the appropriate vulnerability category and the line number in the source code where the vulnerability exists. Here we proceeded with the steps taken in paper [11] the tools are run with two criteria. For SB curated dataset as mentioned in section 5.2.1 the time budget of 30 minutes and execution till halt is measured differently as seen in Fig 5.4 and Fig 5.5. Though the dataset has access control, denial of service, and other types of vulnerabilities present for our study, we are only focusing on five vulnerabilities that were discussed in section 2.1.

From Fig 5.4 it is clearly seen that Our approach is able to detect 59 vulnerabilities in total. Tools such as Slither and Smartcheck were able to detect 39 and 38 vulnerabilities in total, whereas the tool Oyente was able to detect 34. Osiris performed in fifth position with 25 vulnerabilities, and Mythril and Securify were able to detect 20 vulnerabilities. Honeybadger is not able to detect any vulnerability because of the time constraint of half an hour. If we talk about the individual vulnerability category as seen in table 5.2, Slither is performing best in all the categories, whereas Smartcheck is performing well in unchecked low-level calls (ULL). To conclude from our analysis our approach is able to detect a maximum of vulnerabilities overall in this criteria.

Table 5.2: Vulnerability Categorise Division with the Criteria of Time Budget(AT- Arithmetic, FR- Front Running, RT- Reentrancy, TM- Time Manipulation, ULL- Unchecked Low Level Calls)

Tools	AT	FR	RT	TM	ULL	Total
OurApproach	19	2	30	5	4	59
Honeybadger [33]	0	0	0	0	0	0
Manticore [21]	1	0	2	2	0	5
Mythril [28]	7	1	15	0	2	20
Osiris [32]	13	0	21	1	0	25
Oyente [20]	16	2	28	0	0	34
Securify [34]	0	2	14	0	4	20
Slither [10]	0	0	33	2	4	39
Smartcheck [30]	1	0	30	1	6	38

The results shown above could be better as some of the tools could still perform better so in order to have a fair comparison we modified the criteria. Fig 5.5 shows

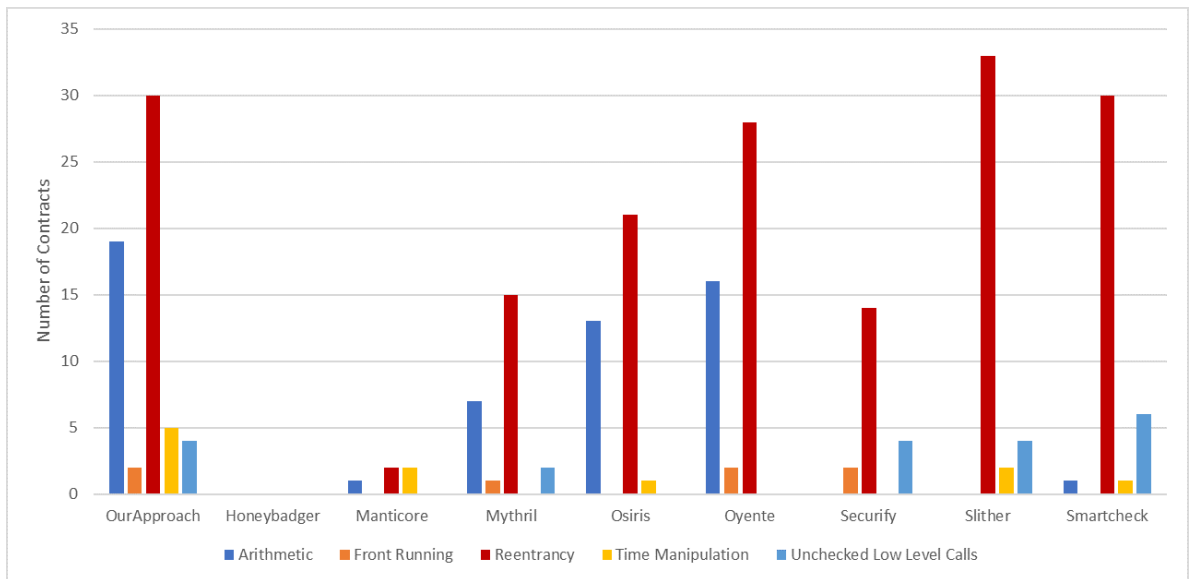


Figure 5.4: Result with the Criteria of Time Budget

the results with the criteria where no limit on time is kept ie execution of tools are done till they halt, here the performance of the tool honey badger is increased and is able to detect a total of 19 vulnerabilities, also the performance of tools such as Mythril, Oyente, and Slither is increased to 51, 48, and 42. As we can see in the detailed category vulnerability detection in table5.3 we still find that our tool is able to detect the maximum vulnerabilities with a number of 61 vulnerabilities in total.

Table 5.3: Vulnerability Categorise Division with the Criteria Without Time Budget(AT- Arithmetic, FR- Front Running, RT- Reentrancy, TM- Time Manipulation, ULL- Unchecked Low Level Calls)

Tools	AT	FR	RT	TM	ULL	Total
OurApproach	19	2	30	7	4	61
Honeybadger [33]	0	0	19	0	0	19
Manticore [21]	13	0	15	4	2	34
Mythril [28]	16	2	25	0	8	51
Osiris [32]	13	0	21	2	0	36
Oyente [20]	18	2	28	0	0	48
Securify [34]	0	2	14	0	6	22
Slither [10]	0	0	33	3	6	42
Smartcheck [30]	1	0	30	2	8	38

After analyzing both scenarios we came to the conclusion that our tool is able to detect the maximum vulnerabilities with an average of 48 % which is 7% more

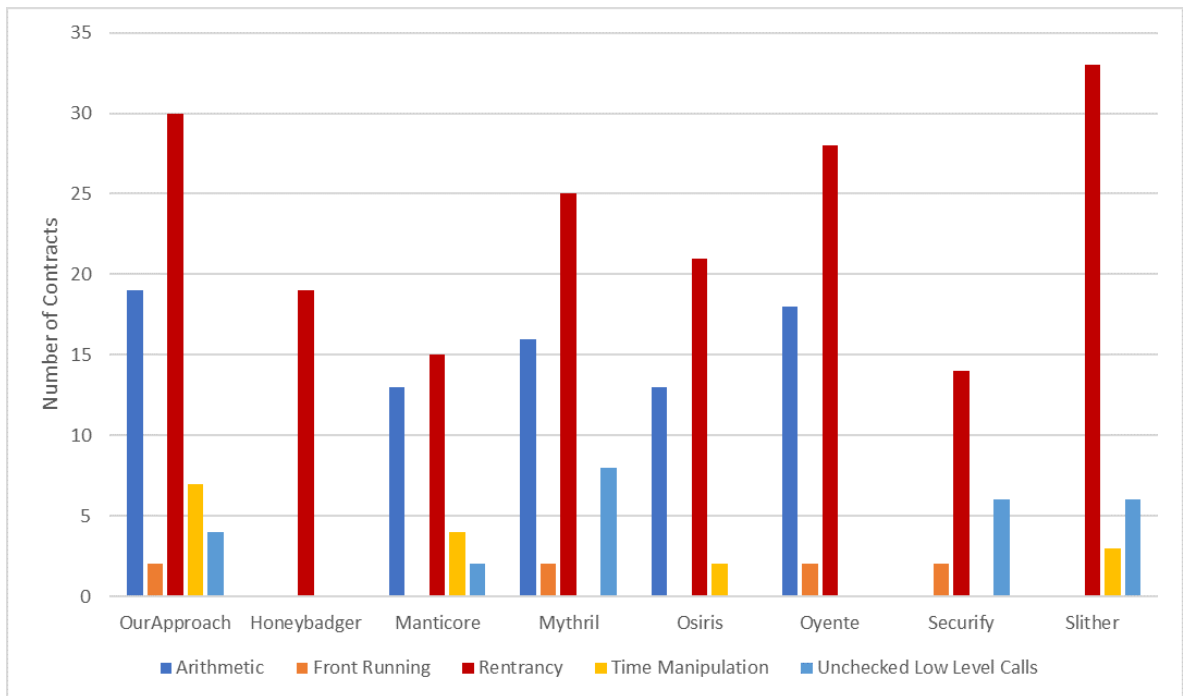


Figure 5.5: Results with the Criteria of Execution Till Halt

than all the tools, Mythril is able to detect 41%, Oyente at 39%, with 36% Smartcheck, and Slither are the tools that are closest. Smartcheck is a tool that finds more categories than any other that have an advantage over others.

These results are only limited to these five categories mentioned, we can't promise the performance of our tools will remain the same for other categories of vulnerabilities or not. The other tools might work exceptionally well for other vulnerability types, but for this study we are only examining these five categories of vulnerability.

### Results of SB wild Dataset

As discussed in section 5.2.2 only 3 thousand smart contracts are examined out of 47 thousand, here no criteria of time budget were kept. Fig 5.6 shows the results of each tool, where our tool is able to detect 2311 vulnerabilities. Whereas the tool Oyente is able to detect 2195 vulnerable contracts. From the results, we can clearly state that our approach is able to detect 2% more vulnerability than the tool Oyente, which is also performing equally well. In particular vulnerability category Slither is still performing well in the Reentrancy category and is able to detect 998 vulnerable contracts. The more detailed category-wise division is seen in table 5.4, to summarize we can say that for this five-category even on the big

dataset of 3031 smart contracts where the vulnerabilities are not known still our approach is performing better than the other tools by 2%.

Table 5.4: Vulnerability Categorise Division (AT- Arithmetic, FR- Front Running, RT- Reentrancy, TM- Time Manipulation, ULL- Unchecked Low Level Calls)

Tools	AT	FR	RT	TM	ULL	Total
OurApproach	1015	32	849	401	14	2311
Honeybadger [33]	151	9	673	30	5	868
Manticore [21]	874	16	589	275	8	1762
Mythril [28]	998	29	732	62	13	1834
Osiris [32]	824	19	716	240	9	1808
Oyente [20]	947	34	813	389	12	2195
Securify [34]	178	27	174	59	18	456
Slither [10]	253	11	998	358	18	1638
Smartcheck [30]	367	25	813	114	20	1339

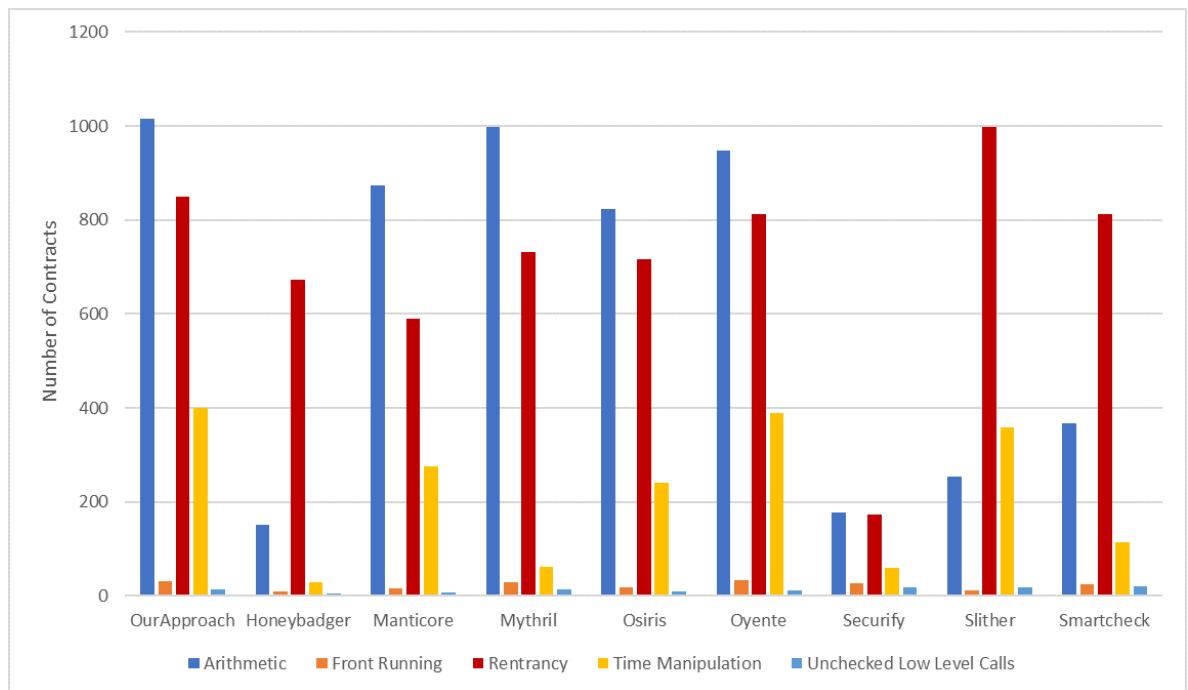


Figure 5.6: Results with the Criteria of Execution Till Halt

## 5.4.2 Performance of Tools

The performance of the tools is an important consideration since if a tool takes a long time to analyze a contract, the user will not be happy. The average and overall times spent using each tool are shown in table 5.5.

Table 5.5: Execution Time of Each Tool for Both Datasets

No.	Tool	Avg ETC	ET(SB curated )	ET(SB wild )
1	Our Approach	00:00:32	1:14:37	1day, 13:18:30
2	Honeybadger [33]	00:01:12	02:49:03	3 days, 20:13:46
3	Manticore [21]	00:12:53	1day, 06:15:28	10 days,06:16:08
4	Mythril [28]	00:00:58	02:16:21	2 days, 20:10:30
5	Osiris [32]	00:00:21	00:50:25	1 day, 01:12:30
6	Oyente [20]	00:00:05	00:12:35	0 days, 07:20:20
7	Securify [34]	00:02:06	04:56:13	6 days 04:06:30
8	Slither [10]	00:00:04	00:09:56	0 days 05:47:40
9	Smartcheck [30]	00:00:15	00:35:23	0 days 17:41:30

It takes into account the tools contract-based execution, including result compilation, analysis, and average parsing. For both the dataset the time of execution of total smart contracts and the average execution time for a contract is taken into account. In table 5.5 the average execution time does not reflect the complete picture of the performance of a tool. When running many tools at once, Honeybadger and Manticore presented challenges because they are difficult to parallelize and could only support four and ten parallel executions, respectively. Manticore is also by far the slower tool. Slither performs best, just taking 4 seconds on average per contract. Our approach is in front of Mythril but behind Oyente.

## 5.5 Chapter Summary

This chapter gives the results based on two datasets over two metrics which are the effectiveness and performance of tools. Further, the single and batch processing is also discussed in the experimental setup section.

## CHAPTER 6

# Conclusion and Future Work

### 6.1 Conclusion

Our study offers a static analysis method that makes use of the Rattle module, the acclaimed SMT Z3 solver, and symbolic execution to maximize performance. We used two different datasets for assessment as we set out to compare our technique to eight popular tools. The first collection included 69 contracts that had been labeled with 124 vulnerabilities. A remarkable 3031 contracts were included in the second collection, which also had vulnerabilities that were regrettably untagged. We implemented five modules designed to identify different types of vulnerabilities. True positives (the quantity of accurately identified vulnerabilities) and overall performance were the two key parameters we used to evaluate the tools' performance. The outcomes we got are impressive and encouraging. Our technique performed exceptionally well in terms of True Positives. Notably, our method was able to find almost 9% more flaws than any other tool in the comparison on the smaller dataset. Our method nevertheless worked well even on the bigger dataset, where the sheer number of contracts offered a tougher difficulty. From the estimation, we observed that our approach performed almost equivalent to the tool oyente but was still able to detect 2% more vulnerabilities. In terms of tool performance, our strategy placed fourth in this category, but tools like Slither and Oyente are quite quick. Therefore with these results, our research indicates that combining our strategy with the Oyente tool may produce even more impressive outcomes.

### 6.2 Future Work

This study sets the groundwork for future improvements in smart contract security analysis. Currently, our approach is only able to detect five types of vulnerabilities, we can add modules to detect more vulnerability categories also the

module for unchecked low-level calls vulnerability is presently not functioning efficiently, but this may be addressed in future work. For future work improving the symbolic engine can be done and different path exploration strategies can be adapted for better results.

## References

- [1] Ethlint, dec. 2021, [online] available:<https://github.com/duaragha>.
- [2] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura. Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, 2021.
- [3] M. Ashouri. Etherolic: A practical security analyzer for smart contracts. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20*, page 353–356, New York, NY, USA, 2020. Association for Computing Machinery.
- [4] C. Chaturvedula, N. Bang, N. Rastogi, and S. Kumar. Price manipulation, front running and bulk trades: Evidence from india. *Emerging Markets Review*, 23, 06 2015.
- [5] H. Chen, G. Whitters, M. J. Amiri, Y. Wang, and B. Loo. Declarative smart contracts, 07 2022.
- [6] W. Chen, Z. Xu, S. Shi, Y. Zhao, and J. Zhao. A survey of blockchain applications in different domains. pages 17–21, 12 2018.
- [7] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura. Ra: A static analysis tool for analyzing re-entrancy attacks in ethereum smart contracts. *J. Inf. Process.*, 29:537–547, 2021.
- [8] L. de Moura and N. Bjørner. Z3: an efficient smt solver. volume 4963, pages 337–340, 04 2008.
- [9] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 530–541, 2019.



- [10] J. Feist, G. Grieco, and A. Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019.
- [11] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu. Smartbugs: A framework to analyze solidity smart contracts. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1349–1352, 2020.
- [12] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020*, page 415–427, New York, NY, USA, 2020. Association for Computing Machinery.
- [13] J. Herrera-Joancomartí. Research and challenges on bitcoin anonymity. volume 8872, 09 2014.
- [14] U. Jafar, M. J. A. Aziz, and Z. Shukur. Blockchain for electronic voting system—review and open research challenges. *Sensors*, 21(17), 2021.
- [15] B. Kakkar, P. Johri, and A. Kumar. Blockchain applications in various sectors beyond:bitcoin. In *2021 International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)*, pages 469–473, 2021.
- [16] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022.
- [17] A. Li, J. A. Choi, and F. Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 438–453, New York, NY, USA, 2020. Association for Computing Machinery.
- [18] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zahl, and K. Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 601–612, 2017.
- [19] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan. Modcon: a model-based testing platform for smart contracts. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

- [20] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.
- [21] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, 07 2019.
- [22] G. Pace, J. Ellul, and S. Azzopardi. Monitoring smart contracts: Contractlarva and open challenges beyond. 11 2018.
- [23] T. Palihapitiya. Blockchain in banking industry. 10 2020.
- [24] P. Praitheeshan, L. Pan, J. Yu, J. K. Liu, and R. R. M. Doss. Security analysis methods on ethereum smart contract vulnerabilities: A survey. *ArXiv*, abs/1908.08605, 2019.
- [25] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang. Smart contract vulnerability detection technique: A survey. *ArXiv*, abs/2209.05872, 2022.
- [26] H. Saeed, H. Malik, U. Bashir, A. Ahmad, S. Riaz, M. Ilyas, W. Bukhari, and M. Khan. Blockchain technology in healthcare: A systematic review. *PLOS ONE*, 17:e0266462, 04 2022.
- [27] A. W. services website. <https://etherscan.io/> = "".
- [28] N. Sharma and S. Sharma. A survey of mythril, a smart contract security analysis tool for evm bytecode. 13:51003–51010, 12 2022.
- [29] L. Stegeman. Solitor : runtime verification of smart contracts on the ethereum network. 2018.
- [30] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB '18*, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery.
- [31] C. F. Torres, M. Baden, R. Norvill, B. B. F. Pontiveros, H. L. Jonker, and S. Mauw. gis: Shielding vulnerable smart contracts against attacks. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

- [32] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.
- [33] C. F. Torres and M. Steichen. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *USENIX Security Symposium*, 2019.
- [34] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [35] D. Vujičić, D. Jagodic, and S. Randić. Blockchain technology, bitcoin, and ethereum: A brief overview. pages 1–6, 03 2018.
- [36] H. Wang, Y. Li, S. Lin, L. Ma, and Y. Liu. Vultron: Catching vulnerable smart contracts once and for all. In *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering*, Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2019, pages 1–4, United States, May 2019. Institute of Electrical and Electronics Engineers Inc. 41st IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2019 ; Conference date: 25-05-2019 Through 31-05-2019.
- [37] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung. Contractguard: Defend ethereum smart contracts with embedded intrusion detection, 11 2019.
- [38] G. Williams. Rattle: a data mining gui for r. *The R Journal*, 1:45–55, 12 2009.
- [39] M. Wöhler and U. Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018.
- [40] P. Yellu, M. R. Monjur, T. Kammerer, D. Xu, and Q. Yu. Security threats and countermeasures for approximate arithmetic computing. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 259–264, 2020.
- [41] X. Zhao, Z. Chen, X. Chen, Y. Wang, and C. Tang. The dao attack paradoxes in propositional logic. pages 1743–1746, 11 2017.

- [42] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. An overview of blockchain technology: Architecture, consensus, and future trends. *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, 2017.
- [43] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. Blockchain challenges and opportunities: a survey. *Int. J. Web Grid Serv.*, 14:352–375, 2018.
- [44] H. Zhou, A. M. Fard, and A. Makanju. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy*, 2022.
- [45] T. Zhou, K. Liu, L. Li, Z. Liu, J. Klein, and T. Bissyandé. Smartgift: Learning to generate practical inputs for testing smart contracts. pages 23–34, 09 2021.