

# **ANALYSIS OF SMART CONTRACT VULNERABILITY AND EXTENSION OF OYENTE TOOL**

The thesis submitted in the partial fulfillment of the requirements  
for the award of the degree of

**B. Tech**  
**in**  
**Computer Science and Engineering**  
**By**

**AYUSH SHARMA (106117018)**

**MAYANK GARG (106117048)**



**DEPARTMENT OF COMPUTER SCIENCE AND  
ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY,  
TIRUCHIRAPPALLI - 620015**

**MAY 2021**

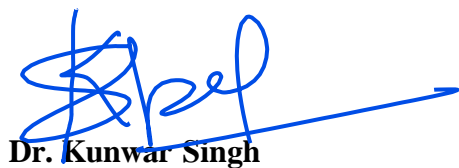
## **BONAFIDE CERTIFICATE**

This is to certify that the project titled **ANALYSIS OF SMART CONTRACT VULNERABILITY AND EXTENSION OF OYENTE TOOL** is a bonafide record of the work done by

**AYUSH SHARMA (106117018)**

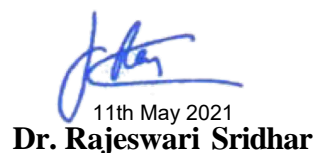
**MAYANK GARG (106117048)**

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**, during the year 2017-2021.



**Dr. Kunwar Singh**

Project Guide,  
Dept. of Computer Science  
and Engineering,  
NITT



11th May 2021  
**Dr. Rajeswari Sridhar**

Head of the Department,  
Dept. of Computer Science  
and Engineering,  
NITT

**Project viva voce held on** \_\_\_\_\_

**Internal Examiner**

**External Examiner**

# ABSTRACT

Ether is the second most valuable cryptocurrency, just after Bitcoin. The biggest difference between Bitcoin and Ethereum is the ability to write smart contracts - small programs that sit on the blockchain. As the contracts are on the blockchain, they become immutable making them attractive for various decentralized applications (or DAPP's) like e-governance, healthcare management and data provenance. However, the biggest advantage of smart contracts - their immutability also poses the biggest threat from a security standpoint. This is because any bug found in the smart contract after deployment cannot be patched. Recent attacks like the DAO attack and the Parity attack have caused massive monetary losses. In such a scenario it becomes imperative to develop and interact with smart contracts that are secure. These bugs suggest subtle gaps in the understanding of the distributed semantics of the underlying platform. With increasing studies in security of smart contracts, novel methods have been devised to detect the bugs and potential vulnerabilities. Static analysis of the contracts is one of the techniques in detecting bugs. Many tools based on static analysis of the smart contract's code have sprung up. We use a well-known tool called Oyente in detecting the bugs. We propose enhancements for the same tool. In this thesis we analyze the Ethereum Smart Contracts from a security viewpoint. We present a study of few of the security vulnerabilities observed in Ethereum smart contracts and develop a novel taxonomy for the same. We further introduced new vulnerabilities, devise algorithms for detection and implement the same to enhance the Oyente tool.

**Keywords:** Ethereum, Smart Contracts, Oyente

## ACKNOWLEDGMENTS

On the very outset of this report, we would like to express our sincere thanks and profound gratitude to **Dr. Kunwar Singh**, Assistant Professor, Department of Computer Science and Engineering, National Institute of Technology, Tiruchirappalli who provided us with this opportunity and for the conscientious guidance and encouragement through the course of the project.

Our hearty thanks to all members of Departmental Project Evaluation Committee Members **Dr. K. Viswanathan Iyer, Dr. Kunwar Singh, and Dr. R. Bala Krishnan** for their precious inputs and constructive criticism during project reviews.

Finally, we wish to thank the scholars and all staff members of the Department of Computer Science and Engineering and extend our gratitude to all teaching and non-teaching staff, our friends and classmates for their help and encouragement during the execution of this project work.

**AYUSH SHARMA** (106117018)

**MAYANK GARG** (106117048)

# TABLE OF CONTENTS

Title	Page
<b>ABSTRACT</b> .....	i
<b>ACKNOWLEDGMENTS</b> .....	ii
<b>TABLE OF CONTENTS</b> .....	iii
<b>LIST OF TABLES</b> .....	vi
<b>LIST OF FIGURES</b> .....	vii
 <b>CHAPTER 1 INTRODUCTION</b>	 1
 <b>CHAPTER 2 LITERATURE REVIEW</b>	 4
2.1 BITCOIN .....	4
2.1.1 BITCOIN INTRODUCTION .....	4
2.1.2 BITCOIN FEATURES .....	5
2.1.3 CONSENSUS PROTOCOL.....	6
2.2 SMART CONTRACTS IN BLOCKCHAIN .....	7
2.3 OPERATIONAL SEMANTICS OF ETHEREUM .....	9
2.4 SECURITY VULNERABILITIES IN ETHEREUM SMART CONTRACTS .....	11
 <b>CHAPTER 3 REQUIREMENT ANALYSIS</b>	 15
3.1 FUNCTIONAL REQUIREMENTS .....	15
3.2 NON-FUNCTIONAL REQUIREMENTS .....	15
3.2.1 USER INTERFACE .....	15
3.2.2 HARDWARE .....	15
3.2.3 SOFTWARE.....	15
3.2.3.1 TOOLS .....	16
3.2.3.2 PACKAGES NEEDED .....	16

3.2.3.3	IMPORTANT LIBRARIES USED .....	17
3.2.4	PERFORMANCE .....	17
3.3	CONSTRAINTS AND ASSUMPTIONS .....	17
3.3.1	CONSTRAINTS .....	17
3.3.2	ASSUMPTIONS .....	17
3.4	TOOLS AVAILABLE FOR ETHEREUM SMART CONTRACTS .	17
3.4.1	SECURITY TOOLS .....	18
3.4.2	VISUALISATION TOOLS .....	18
3.4.3	DISASSEMBLER AND DECOMPILER .....	18
3.4.4	LINTER .....	18
3.5	METHODS EMPLOYED BY THE SECURITY TOOLS .....	18
3.5.1	STATIC ANALYSIS .....	18
3.5.2	SYMBOLIC EXECUTION .....	18
<b>CHAPTER 4 SYSTEM DESIGN</b>		19
4.1	OYENTE TOOL AND ARCHITECTURE .....	19
4.1.1	DESIGN OVERVIEW .....	19
4.1.2	IMPLEMENTATION .....	20
4.2	OUR WORK .....	21
4.2.1	EXTERNAL CONTRACT REFERENCING .....	22
4.2.2	EFFICIENT TRANSACTION ORDERING DEPENDENCE.....	25
4.2.3	TX_ORIGIN .....	27
4.3	ALGORITHMS DESIGNED TO DETECT VULNERABILITIES ..	29

<b>CHAPTER 5 IMPLEMENTATION AND RESULTS</b>	<b>30</b>
5.1 IMPLEMENTATION .....	30
5.2 RESULTS .....	38
<b>CHAPTER 6 SUMMARY AND CONCLUSION</b>	<b>40</b>
6.1 SUMMARY .....	40
6.2 CONCLUSION .....	40
6.3 SCOPE FOR FUTURE WORK .....	41
<b>REFERENCES</b>	<b>42</b>

## LIST OF TABLES

Table	Title	Page
4.1	Vulnerabilities Detection in Old and New Tools .....	22
5.1	Comparison of performance between old and enhanced tools .....	38
5.2	Recall of the newly detected vulnerabilities .....	39



## LIST OF FIGURES

<b>Fig No.</b>	<b>Title</b>	<b>Page</b>
2.1	Blockchain Design in Cryptocurrencies like Ethereum and Bitcoin .....	6
2.2	Graph of Number of Smart contracts in Ethereum with Time .....	8
2.3	Semantics for Proposing and Accepting a Block in Ethereum .....	9
2.4	The Activation Record Stack Definition .....	10
2.5	Rules for Transaction Execution .....	11
2.6	Double spending attack .....	12
2.7	Mechanism of Re-entrancy attack .....	13
4.1	Overview of Oyente Architecture .....	19
4.2	Rot13Encryption.sol .....	23
4.3	Contract.sol .....	25
4.4	Point of Attack .....	26
4.5	Tx_origin.sol .....	28
5.1	Complete Flow of the Execution .....	31
5.2	Sample Smart Contract Vulnerable to Attacks .....	31
5.3	Byte Code for the Sample Smart Contract After Compilation .....	32
5.4	Assembly Level Opcode File generated by Dis-assembler .....	33
5.5	Results of the Execution of the Original Tool .....	36
5.6	Results of the Execution of the Enhanced Tool .....	37

# 1. INTRODUCTION

Smart Contracts are one of the most promising features of blockchain technology and have created a lot of buzz over the years. They are essentially small computer programs that exist on the blockchain. This makes them immutable as long as the blockchain's integrity is not compromised. Therefore, the end-users can be sure that the program has not been changed or manipulated. This provides the people with trust in a distributed environment where they do not have to trust the other nodes or a centralized third party.

Ethereum is the second most valuable crypto-currency, just after Bitcoin. However, the most distinguishing feature of Ethereum was the introduction of smart contracts. This has fostered a new area of software development called decentralized applications (or dApps). These are applications that utilize the features of blockchains (immutability and lack of central authority) in the backend and combine them with user-friendly front-ends for non-technical users. The development of dApps for different application areas is an active area of research with the most popular ones being decentralized identity management, land record management, e-governance systems, etc. Essentially, blockchains help in maintaining a tamper-proof log that is used to establish ownership of actions. Therefore, it has wide applications and can help establish the trust of the users in the system.

Unfortunately, the most promising feature of smart contracts - immutability poses unique challenges from a security and a software engineering standpoint. The traditional software development life cycle (SDLC) is an iterative process where new features are added over time. Also, security issues and bugs found are fixed by releasing patches. However, as smart contracts are immutable they cannot be changed or fixed once they are deployed on the blockchain. Since Ethereum is a public blockchain, all the data is available in the public domain for hackers and other malicious actors to exploit.

The issue of smart contract security is not an academic one. Since Ethereum is a crypto-currency backed blockchain, hackers have a monetary incentive to find buggy contracts and exploit them. Over time we have seen many attacks happen on Ethereum smart contracts. The most

popular attack was the Re-entrancy attack on the Decentralised Autonomous Organisation (DAO). The DAO was a high-value contract with its token sale setting the record for the largest crowdfunding campaign. Unfortunately, because of the Re-entrancy bug, attackers were able to siphon Ether (the native cryptocurrency of Ethereum) worth US\$50M at the time. The attack was so massive that it caused the Ethereum community and the blockchain itself to split into two - Ethereum (which reverted the effects of the attack) and Ethereum Classic (which carried on after the attack). In 2017, hackers stole US\$30M from the Parity multisig wallet because of function default visibility and unchecked external call bugs. They would have stolen more, however, white hat hackers hacked the remaining wallets and saved around US\$150M.

Also, it has been observed that many smart contract variants of Ponzi schemes and pyramid schemes have come upon the Ethereum blockchain. Even though the community is actively spreading awareness, an unsuspecting user might get trapped in such schemes and lose their hard-earned money.

The main goal of this thesis is to serve as a reference for smart contract developers and smart contract end-users on the various security vulnerabilities, static time analysis for detecting the potential vulnerabilities. Also, we introduced new vulnerabilities, devised algorithms to detect these vulnerabilities, and coded the implementation to the existing tool. We have added 3 new vulnerabilities to the Oyente Tool. These vulnerabilities are as follows:

1. External Contract Referencing: When a contract calls another smart contract externally. It can lead to an attack on the user of the smart contract.
2. Efficient Transaction Ordering Dependence(TOD): We have made some improvements in the existing TOD, which increases the efficiency of the Oyente tool.
3. Tx\_origin : This is an environment variable in solidity which can be maliciously used by an attacker. So whenever a user uses tx.origin in his smart contract, we flag it as a potential vulnerability.

The thesis is divided as follows – Chapter 2 introduces the concepts of blockchains using Bitcoin. Then we look at the Ethereum blockchain and Smart Contracts. Then we introduce the operational semantics of the Ethereum. Finally, we look at the related work that has been done in

the past. Chapter 3 looks at the various requirements that are needed for this project ranging from tools to packages. Chapter 4 introduces the original Oyente tool and demonstrates its capabilities and Design. We also introduce new vulnerabilities, their description along with examples. Chapter 5 talks about the implementation methodology, our on-chain smart contract data collection, its analysis across different parameters, and performance analysis of the enhanced Oyente tool. We conclude with Chapter 6 with directions for future research work.

## **2. LITERATURE REVIEW**

### **2.1 BLOCKCHAIN INTRODUCTION**

A blockchain, originally block chain, is a growing list of records, called blocks, that are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle tree).

By design, a blockchain is resistant to modification of the data. It is "an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way". For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for inter-node communication and validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without the alteration of all subsequent blocks, which requires consensus of the network majority. Although blockchain records are not unalterable, blockchains may be considered secure by design and exemplify a distributed computing system with high Byzantine fault tolerance. Decentralized consensus has therefore been claimed with a blockchain.

#### **2.1.1 Bitcoin Introduction:**

The first (and arguably the most popular) blockchain is Bitcoin - the brainchild of the pseudonym Satoshi Nakamoto. The main motive behind Bitcoin was to serve as a decentralized and distributed global currency - without any centralized banks.

The ledger of transactions is stored as a chain (or a linked list) of blocks but the pointers connecting these blocks are hash pointers that are dependent on the block information. The block information does not store the transaction directly. Each transaction becomes the leaf node of a Merkle tree, and the root of the Merkle tree is included in the block information. Therefore, any attempt to manipulate the data would not be possible without breaking (or forking) the chain.

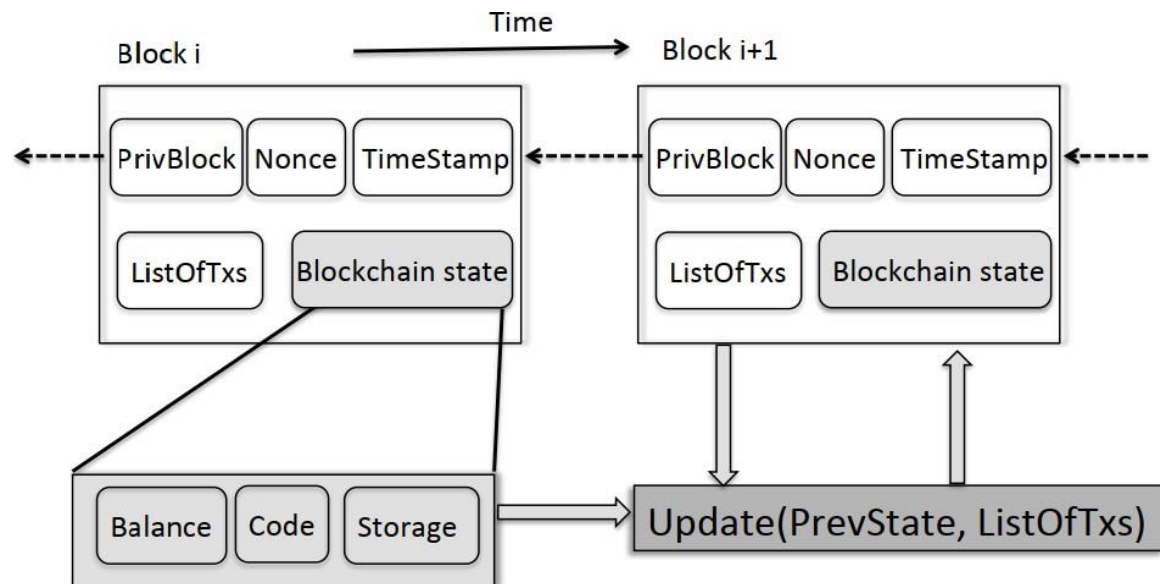
Also, as we are in distributed systems territory, we encounter one of the most famous problems of consensus, represented by the Byzantine General's Problem. Essentially in a distributed system that is prone to Byzantine faults, how do we ensure that every node maintains the same ledger of transactions i.e. all the nodes are in consensus about the contents of the ledger? Bitcoin introduced an approach called proof of work to deal with it. However, other consensus

algorithms like Proof of Stake and Proof of Burn, etc. are also used in other blockchains. Finding the answer to the cryptographic hash puzzle (proof of work for Bitcoin) is called mining. This is because whoever finds the answer gets a block reward in bitcoin (essentially creating or mining new bitcoin).

**2.1.2 Bitcoin Features:** The fundamental features on which Bitcoin (and other early blockchain platforms) was built upon are –

1. **Distributed and Public** – It is a distributed peer-to-peer (P2P) network hosted over the internet. Anyone with an internet connection can join the network, view the blockchain, or become a miner.
2. **Decentralized** – There is no central authority (like banks) in the network. This means that transactions cannot be reversed and there is little chance of grievance redressal.
3. **The consensus among nodes** – All the nodes agree to the state of the blockchain. There might be temporary soft forks along the way but eventually, only the longest chain survives.
4. **Cryptographically Secure and Immutable** – The blocks are linked by hash pointers which are computationally very expensive to find. Therefore blockchain contents are more or less immutable. The deeper the block is in the chain, the more difficult and resource-intensive it becomes to manipulate transactions in that block.
5. **Pseudo-anonymous** – There are no unique identifiers on the blockchain. Everyone is known by their address. Also, there is no restriction on the number of accounts a person can have. This had earlier led to Bitcoin becoming a hub for illegal activities as it provided people with ‘digital cash’.
6. **Easily Verifiable** – Since the blockchain data is publicly visible and known to all, anyone can view and verify transactions on the chain.
7. **Limited Supply** – The supply of bitcoins is limited to 21M. After this, no new bitcoins can be mined and miners will stop getting the block rewards and only receive the transaction fees.

**2.1.3 Consensus Protocol:** Decentralized cryptocurrencies secure and maintain a shared ledger of facts between a set of peer-to-peer network operators (or miners). Miners run a peer-to-peer consensus protocol called the Nakamoto consensus protocol. The shared ledger is called a blockchain and is replicated by all miners. The ledger is organized as a hash-chain of blocks ordered by time, wherein each block has a set of facts, as shown in fig 2.1.



**Fig. 2.1 Blockchain's Design in Cryptocurrencies like Ethereum and Bitcoin.**

**Each block consists of several transactions.**

In every epoch, each miner proposes their own block to update the blockchain. Miners can select a sequence of new transactions to be included in the proposed block. At a high level, Nakamoto consensus works by probabilistically electing a leader among all the miners via a proof-of-work puzzle. The leader then broadcasts its proposed block to all miners. If the proposed block obeys certain predefined validity constraints, such as those ensuring mitigation of “double-spending” attacks, then all miners update their ledger to include the new block. Here excluded are certain details about the consensus protocol, such as the use of the longest-chain rule for resolving probabilistic discrepancies in leader election. Instead, we refer readers to the original Bitcoin or Ethereum paper for details [2, 3].

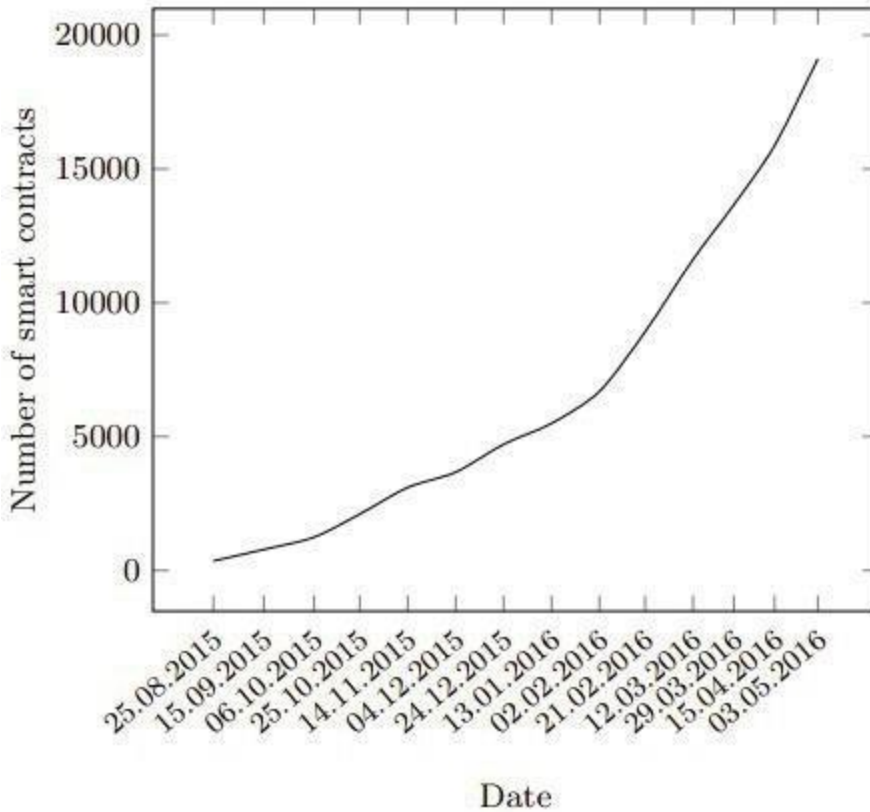
A blockchain state  $\sigma$  is a mapping from addresses to accounts; the state of an account at address  $\gamma$  is  $\sigma[\gamma]$ . While Bitcoin only has normal accounts that hold some coins, Ethereum

additionally supports smart contract accounts that have coins, executable code, and persistent (private) storage. Ethereum supports its own currency called Ether; users can transfer coins to each other using normal transactions as in Bitcoin, and additionally can invoke contracts using contract-invoking transactions. Conceptually, Ethereum can be viewed as a transaction-based state machine, where its state is updated after every transaction. A valid transition from  $\sigma$  to  $\sigma'$ , via transaction  $T$  is denoted as  $\sigma \xrightarrow{T} \sigma'$ .

## 2.2 SMART CONTRACTS IN BLOCKCHAIN

A smart contract (or contract for short) is an “autonomous agent” stored in the blockchain, encoded as part of a “creation” transaction that introduces a contract to the blockchain. Once successfully created, a smart contract is identified by a contract address; each contract holds some amount of virtual coins (Ether), has its own private storage, and is associated with its predefined executable code. A contract state consists of two main parts: private storage and the number of virtual coins (Ether) it holds (called balance). Contract code can manipulate variables like in traditional imperative programs. The code of an Ethereum contract is in a low-level, stack-based bytecode language referred to as Ethereum virtual machine (EVM) code. Users define contracts using high-level programming languages, e.g., Solidity (a JavaScript-like language), which are then compiled into EVM code. To invoke a contract at address  $\gamma$ , users send a transaction to the contract address. A transaction typically includes: payment (to the contract) for the execution (in Ether) and/ or input data for the invocation.





**Fig. 2.2 Graph of Number of Smart contracts in Ethereum with Time**

**Gas system:** By design, the smart contract is a mechanism to execute code distributively. To ensure fair compensation for expended computation effort, Ethereum pays miners some fees proportional to the required computation. Specifically, each instruction in the Ethereum bytecode has a pre-specified amount of gas. When a user sends a transaction to invoke a contract, she has to specify how much gas she is willing to provide for the execution (called `gasLimit`) as well as the price for each gas unit (called `gasPrice`). A miner who includes the transaction in his proposed block subsequently receives the transaction fee corresponding to the amount of gas the execution actually burns multiplied by `gasPrice`. If some execution requires more gas than `gasLimit`, the execution is terminated with an exception, the state  $\sigma$  is reverted to the initial state as if the execution did not happen. In the case of such aborts, the sender still has to pay all the `gasLimit` to the miner though, as a countermeasure against resource-exhaustion attacks.

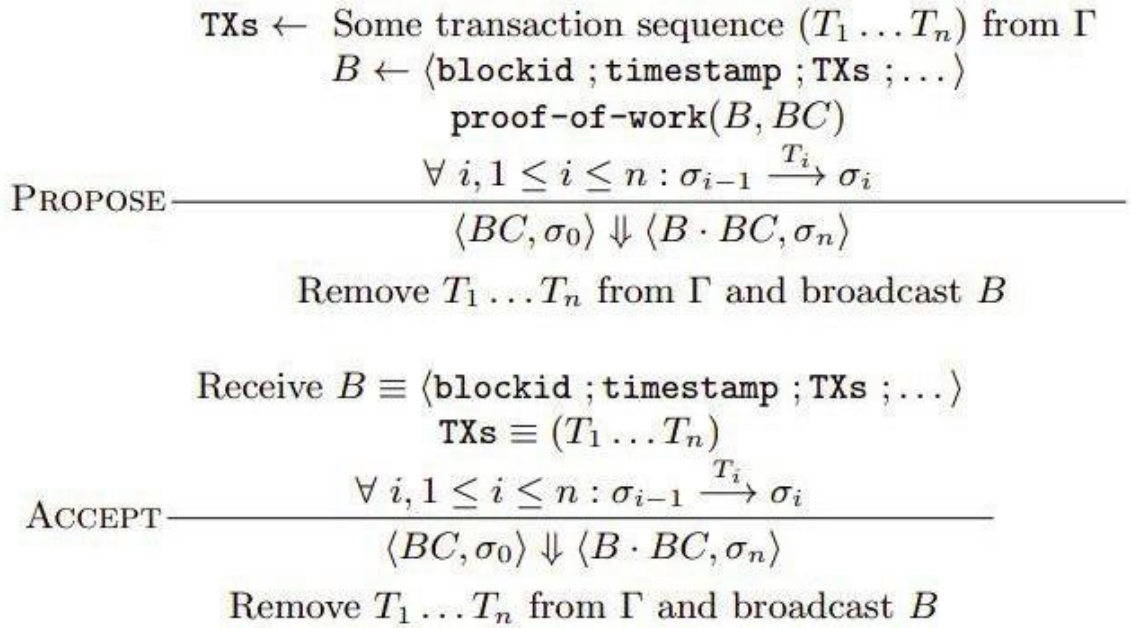
## 2.3 OPERATIONAL SEMANTICS OF ETHEREUM

The canonical state of Ethereum, denoted by  $\sigma$ , is a mapping between addresses and account states.

A valid transition from  $\sigma$  to  $\sigma'$  via transaction  $T$  is written as  $\sigma \xrightarrow{T} \sigma'$ .

**Formation and Validation of a Block:** To model the formation of the blockchain and the execution of blocks, a global Ethereum state is defined as a pair  $(BC, \sigma)$ , where  $BC$  is the current blockchain and  $\sigma$  is as before.  $\Gamma$  denotes the stream of incoming new transactions. For simplicity, we do not model miner rewards.

The actions of the miners to form and validate blocks are given in fig2.3. Only one “elected leader” executes successfully the Proposed rule at a time. Other miners use Accept rule to “repeat” the transitions  $\sigma_{i-1} \xrightarrow{T_i} \sigma_i$  after the leader broadcasts block  $B$ .



**Fig. 2.3 Semantics for Proposing and Accepting a Block in Ethereum [7]**

**Security Issues:** The issue of timestamp dependence arises because the elected leader has some slack in setting the timestamp, yet other miners still accept the block. On the other hand, the issue of transaction-ordering dependence exists because of some inevitable order among  $T_i$ ; It is shown that when dealing with Ether (or money), this might lead to undesirable outcomes.

**Transaction Execution:** A transaction can activate the code execution of a contract. In Ethereum, the execution can access three types of space in which to store data:

1. An operand LIFO stack  $s$ ;
2. An auxiliary memory  $l$ , an infinitely expandable array; and
3. The contract's long-term storage  $str$ , which is part of  $\sigma[id]$  for a given contract address  $id$ .

Unlike stack and auxiliary memory, which reset after computation ends, storage persists as part of  $\sigma$ .

Virtual machine's execution state  $\mu$  is defined as a configuration  $\langle A, \sigma \rangle$ , where  $A$  is a call stack (of activation records) and  $\sigma$  is as before. The activation record stack is defined as shown in fig 2.4.

$$\begin{aligned} A &\triangleq A_{normal} \mid \langle e \rangle_{exc} \cdot A_{normal} \\ A_{normal} &\triangleq \langle M, pc, l, s \rangle \cdot A_{normal} \mid \epsilon \end{aligned}$$

**Fig. 2.4 The Activation Record Stack Definition [7]**

where  $\epsilon$  denotes an empty call stack;  $\langle e \rangle_{exc}$  denotes that an exception has been thrown; and each part of an activation record  $\langle M, pc, l, s \rangle$  has the following meaning:

1.  $M$  : the contract code array
2.  $pc$  : the address of the next instruction to be executed
3.  $l$  : an auxiliary memory (e.g. for inputs, outputs)
4.  $s$  : an operand stack.

Though a transaction in Ethereum is a complex structure and specifies a number of fields, It can be abstracted to a triple  $(id, v, l)$  where

1.  $id$  is the identifier of the to-be-invoked contract,
2.  $v$  is the value to be deposited to the contract, and
3.  $l$  is a data array capturing the values of input parameters.

Thus a transaction execution can be modeled with the rules in fig 2.5: the first rule describes an execution that terminates successfully (or “normal halting”) while the second rule describes one that terminates with an exception.

$$\begin{array}{c}
 \text{TX-SUCCESS} \frac{
 \begin{array}{l}
 T \equiv \langle id, v, l \rangle \quad M \leftarrow \text{Lookup}(\sigma, id) \\
 \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\
 \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \epsilon, \sigma'' \rangle
 \end{array}
 }{
 \sigma \xrightarrow{T} \sigma''
 }
 \\
 \\
 \text{TX-EXCEPTION} \frac{
 \begin{array}{l}
 T \equiv \langle id, v, l \rangle \quad M \leftarrow \text{Lookup}(\sigma, id) \\
 \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\
 \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \langle e \rangle_{exc} \cdot \epsilon, \bullet \rangle
 \end{array}
 }{
 \sigma \xrightarrow{T} \sigma
 }
 \end{array}$$

**Fig. 2.5 Rules for Transaction Execution [7].**  $\text{Lookup}(\sigma, id)$  finds the associated code of contract address  $id$  in state  $\sigma$ ;  $\sigma[id][bal]$  refers to the balance of the contract at address  $id$  in state  $\sigma$ .

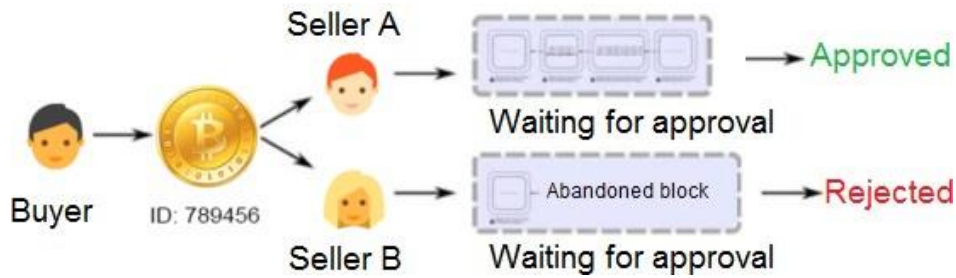
The execution of a transaction is intended to follow the “transactional semantics” of which two important properties are:

1. Atomicity, requiring that each transaction be “all or nothing”. In other words, if one part of the transaction fails, then the entire transaction fails and the state is left unchanged; and
2. Consistency, ensuring that any transaction will bring the system from one valid state to another.

## 2.4 SECURITY VULNERABILITIES IN ETHEREUM SMART CONTRACTS

**51% Attack:** In proof of work, the miners try finding the nonce value to solve the given cryptographic puzzle. However, if miner(s) get control of more than 51% of the computing power in the network then they essentially control what goes into the blockchain - compromising its integrity.

**Double Spending:** Double spending occurs when the attacker uses the same



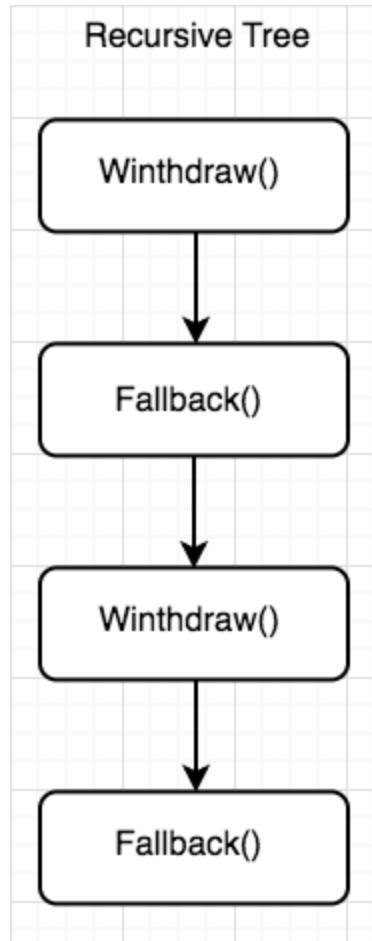
**Fig. 2.6 Double spending attack**

cryptocurrency more than once. This is done by leveraging race conditions, forks in the chain, or 51% attacks as shown in fig 2.6. A variant of this attack is the Finney attack. Such attacks have been shown on Bitcoin.

**Selfish Mining:** In selfish mining, a malicious miner does not publish the block immediately after solving the proof of work puzzle. Instead, reveals it only to its pool members which then work on the next block, while the other network continues working for essentially nothing. This was first demonstrated on Bitcoin [4], and recently on Ethereum as well .

**Re-entrancy:** A Re-entrancy condition is when a malicious party can call a vulnerable function of the contract again before the previous call is completed: once or multiple times. This type of function is especially problematic in the case of payable functions, as a vulnerable contract might be emptied by calling the payable function repeatedly as shown in fig 2.7. The call() function is especially vulnerable as it triggers code execution without setting a gas limit.

To avoid Re-entrancy bugs, it is recommended to use transfer() and send() as they limit the code execution to 2300 gas. Also, it is advised to always do the required work (i.e. change the balances, etc.) before the external call.



**Fig. 2.7 Mechanism of Re-entrancy attack.**

The DAO Attack is the most famous Re-entrancy attack which led to a loss of US\$50 Million and resulted in the chain being forked into two - Ethereum and Ethereum Classic.

**Transaction Order Dependence (Front Running):** The order in which the transactions are picked up by miners might not be the same as the order in which they arrive. This creates a problem for contracts that rely on the state of the storage variables. Gas sent is usually important as it plays an important role in determining which transactions are picked first. A malicious transaction might be picked first, causing the original transaction to fail. This kind of race-condition vulnerability is referred to as transaction order dependence.

**Overflows and Underflows:** Solidity can handle up to 256-bit numbers, and therefore increasing (or decreasing) a number over (or below) the maximum (or minimum) value can result in

overflows (or underflows). It is recommended to use OpenZeppelin's SafeMath library to mitigate such attacks.

**Timestamp Dependence:** A lot of applications have a requirement to implement a notion of time in their applications. The most common method of implementing this is using the `block.timestamp` either directly or indirectly. However, a malicious miner with a significant computational power can manipulate the timestamp to get an output in his/her favor.

### **3. REQUIREMENT ANALYSIS**

#### **3.1 FUNCTIONAL REQUIREMENTS**

The system outputs a set of results showing metrics like code coverage, a boolean value (Yes/No) for various vulnerabilities, and the corresponding instances where such a vulnerability exists if detected.

The input smart contract should adhere to the following requirements:

The smart contract should be syntactically correct.

The smart contract's version should be compatible with the corresponding version of the Oyente tool setup.

The smart contract should not have any syntax errors.

The system must be optimized for time and space complexities.

The system must be able to detect any of the mentioned vulnerabilities if present in the smart contract.

#### **3.2 NON FUNCTIONAL REQUIREMENTS**

##### **3.2.1 User Interface:**

There must be a simple and easy to use user interface where the user should be able to enter his/her smart contract for vulnerability detection. The interface also should be easy to view the results.

##### **3.2.2 Hardware:**

No special hardware interface is required for the successful implementation of the system. For files with multiple smart contracts, heavy memory is needed.

##### **3.2.3 Software:**

Operating system: Linux

Programming language: Python, Solidity, Assembly level language.

Tools: Oyente, Pycharm, Docker, Remix IDE.

Packages needed: Solc, evm.

Concepts used: Ethereum blockchain, Compiler design.



Important libraries used: Z3, Pickle, Math

### **3.2.3.1 Tools:**

**Oyente:** Oyente is one of the earliest security tools for Solidity smart contracts. It was developed by security researchers at the National University of Singapore and is now being maintained by Melonport. Oyente leverages symbolic execution to find potential vulnerabilities in the smart contracts. It works with both byte-codes and solidity files.

Being one of the first tools in this area, Oyente has been extended by many researchers over the years. For example, the control flow graphs generated by Oyente are also used by EthIR, which is a high-level analysis tool for Solidity.

**Pycharm:** PyCharm is an integrated development environment (IDE) used in computer programming, specifically for the Python language. It is developed by JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (VCSes). We used it primarily for debugging and finding the patterns in the assembly level code.

**Remix IDE:** Remix IDE is primarily an integrated development environment (IDE) for developing Solidity smart contracts. It can connect to the Ethereum network using Metamask and developers can directly deploy smart contracts from Remix. It is developed and maintained by the Ethereum Foundation.

### **3.2.3.2 Packages Needed:**

#### **Solc:**

Solc is the actual Solidity compiler. Solc is written in C++. The C++ code is compiled to JavaScript using emscripten. Every version of solc is compiled to JavaScript.

#### **Evm:**

EVM is an acronym for Ethereum Virtual Machine. The Ethereum blockchain is widely regarded as being the first blockchain system that has a team dedicated to its development and maintaining its well-being. Simply, Ethereum is a decentralized platform that primarily deals with running smart contracts.

### **3.2.3.3 Important Libraries used:**

**Z3:** Z3 is a state-of-the-art theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories. Z3 offers a compelling match for software analysis and verification tools since several common software constructs map directly into supported theories. Z3 is a low-level tool. It is best used as a component in the context of other tools that require solving logical formulas.

**3.2.4 Performance:** The system must be optimized, reliable, consistent, and available all the time. The system must be able to detect the vulnerabilities with higher accuracy and less false positivity rate.

## **3.3 CONSTRAINTS AND ASSUMPTIONS**

### **3.3.1 Constraints:**

- The smart contracts should be syntactically correct.
- The smart contracts should be compatible with the Oyente tool version.
- The results for vulnerability detection are expected to be present after the whole process on the terminal.

### **3.3.2 Assumptions:**

The given smart contract is assumed to be syntactically correct.

The smart contract generated is assumed to be semantically correct.

The input smart contract is version compatible with the Oyente tool.

## **3.4 TOOLS AVAILABLE FOR ETHEREUM SMART CONTRACTS**

There are many different tools available for Ethereum Smart Contracts. These tools have been gathered from research publications and through Internet searches. In this section, we have classified the various tools available into different categories, so that the end-users can easily find which tool to use for their particular application.

### **3.4.1 Security Tools:**

These are tools that take as input either the source code or the bytecode of a contract and give outputs on the security issues present. These are the tools that we are primarily concerned with.

### **3.4.2 Visualization Tools:**

Visualization tools help give graphical outputs like control flow graphs, dependency graphs, etc. of the given contract to help in analysis. Tools like solgraph and rattle fall under this category.

### **3.4.3 Disassemblers and Decompilers:**

A dis-assembler converts the binary code back into the high-level language code while a decompiler converts the binary code to a low-level language for better understanding. evm-dis is a popular dis-assembler for smart contracts.

### **3.4.4 Linters:**

Linters are static analysis tools primarily focused on detecting poor coding practices, programming errors, etc. Ethlint is a common linting tool of Ethereum smart contracts.

## **3.5 METHODS EMPLOYED BY THE SECURITY TOOLS**

### **3.5.1 Static Analysis:**

Static Analysis essentially means evaluating the program code without actually running it. It looks at the code structure, the decompiled outputs, and control flow graphs to identify common security issues. RemixIDE is a static analysis security tool for Ethereum smart contracts.

### **3.5.2 Symbolic Execution:**

Symbolic execution is considered to be in the middle of static and dynamic analysis. It explores possible execution paths for a program without any concrete input values. Instead of values, it uses symbols and keeps track of the symbolic state. It leverages constraint solvers to make sure that all the properties are satisfied. Mythril and Oyente are the popular Symbolic Execution tools for smart contract security.

## 4. SYSTEM DESIGN

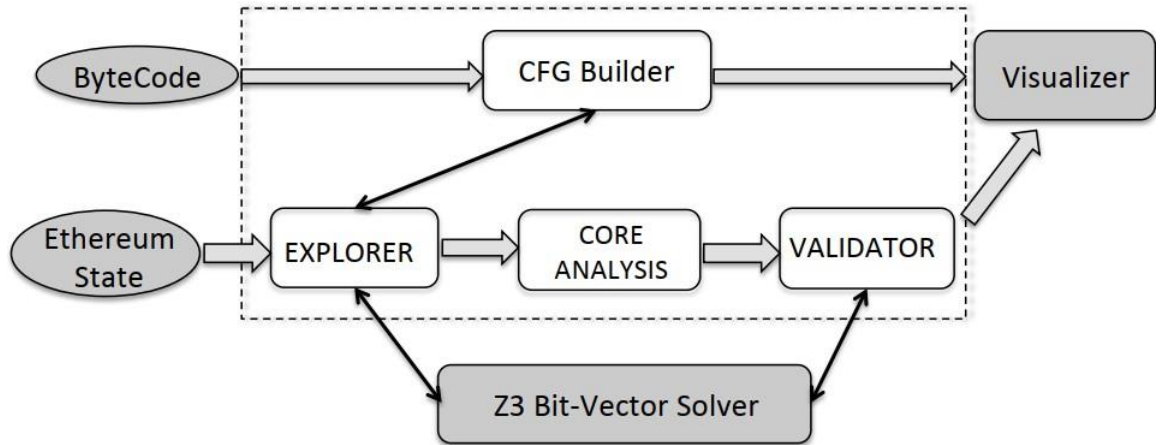
### 4.1 OYENTE TOOL AND ARCHITECTURE:

Oyente helps:

1. Developers to write better contracts; and
2. Users to avoid invoking problematic contracts. Importantly, other analyses can also be implemented as independent plugins, without interfering with our existing features. E.g., a straightforward extension of Oyente is to compute a more precise estimation of worst-case gas consumption for contracts.

The Oyente tool is based upon symbolic execution. Symbolic execution represents the values of program variables as symbolic expressions of the input symbolic values. Each symbolic path has a path condition which is a formula over the symbolic inputs built by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is infeasible if its path condition is unsatisfiable. Otherwise, the path is feasible.

#### 4.1.1 Design Overview



**Fig. 4.1 Overview of Oyente architecture [7]**

Fig 4.1 depicts the architecture overview of Oyente. It takes two inputs including the bytecode of a contract to be analyzed and the current Ethereum global state. It answers whether the contract has any security problems (e.g., TOD, timestamp-dependence, mishandled exceptions), outputting

“problematic” symbolic paths to the users. One by-product of the tool is the Control Flow Graph (CFG) of the contract bytecode.

The bytecode is publicly available on the blockchain and Oyente interprets EVM instruction set to faithfully map instructions to constraints, i.e., bit-level accuracy. The Ethereum global state provides the initialized (or current) values of contract variables, thus enabling more precise analysis. All other variables including value, data of message calls are treated as input symbolic values.

Oyente follows a modular design. It consists of four main components, namely CFGBuilder, Explorer, CoreAnalysis, and Validator. CFGBuilder constructs a Control Flow Graph of the contract, where nodes are basic execution blocks, and edges represent execution jumps between the blocks. Explorer is the main module that symbolically executes the contract. The output of Explorer is then fed to the CoreAnalysis. Finally, Validator filters out some false positives before reporting to the users.

#### **4.1.2 Implementation:**

The components of Oyente are as follows:

**CFG Builder:** CFGBuilder builds a skeletal control flow graph which contains all the basic blocks as nodes, and some edges representing jumps of which the targets can be determined by locally investigating the corresponding source nodes. However, some edges cannot be determined statically at this phase, thus they are constructed on the fly during symbolic execution in the later phase.

**Explorer:** The Explorer starts with the entry node of the skeletal CFG. At any one time, Explorer may be executing a number of symbolic states. The core of Explorer is an interpreter loop that gets a state to run and then symbolically executes a single instruction in the context of that state. This loop continues until there are no states remaining, or a user-defined timeout is reached.

At the end of the exploration phase, a set of symbolic traces are produced. Each trace is associated with a path constraint and auxiliary data that the analyses in the later phase require. The employment of a constraint solver, Z3 in particular, helps in eliminating provably infeasible traces from consideration.

**Core Analysis:** CoreAnalysis contains sub-components to detect contracts which are TOD, timestamp-dependent, or mishandled exceptions. Currently, the Explorer collects only paths which exhibit distinct flows of Ether. Thus, if a contract is detected as TOD vulnerable if it sends out Ether differently when the order of transactions changes. Similarly, checking is done if a contract is timestamp-dependent if the condition to send includes the block timestamp.

**Validation:** The last component is Validator which attempts to remove false positives. For instance, given a contract flagged as TOD (Transaction ordering dependency) by CoreAnalysis and its two traces t1 and t2 exhibiting different Ether flows, Validator queries Z3 to check if both ordering (t1, t2) and (t2, t1) is feasible. If no such t1 and t2 exist, the case is considered as a false positive.

## 4.2 OUR WORK

The Oyente tool was able to detect seven different vulnerabilities. There are many other vulnerabilities which are either detected but their solution have not been implemented yet or remain undetected. In our thesis we found three new attacks and implemented their solution for increasing efficiency of Oyente tool. We studied the vulnerabilities, the attacks that are possible exploiting these vulnerabilities in given contracts. We designed algorithms to detect these vulnerabilities by finding patterns from the assembly level opcode file of given smart contracts. Then we implemented the algorithms and added them to the existing tool to increase overall performance.

The three vulnerabilities are:

- Contract Referencing
- Efficient Transaction ordering Dependence
- Tx\_origin

Also, we were able to improve existing vulnerabilities detection capability for Integer Overflow and Integer Underflow. Table 4.1 shows the vulnerabilities detected by both the old and enhanced tools.

Old Tool	New Tool
<ul style="list-style-type: none"> <li>• Integer Overflow</li> <li>• Integer Underflow</li> <li>• Call Stack depth attack</li> <li>• Transaction ordering Dependence</li> <li>• Timestamp Dependence</li> <li>• Re-entrancy vulnerability</li> <li>• Parity Multisig Bug</li> </ul>	<ul style="list-style-type: none"> <li>• Integer Overflow</li> <li>• Integer Underflow</li> <li>• Call Stack depth attack</li> <li>• Transaction ordering Dependence</li> <li>• Timestamp Dependence</li> <li>• Re-entrancy vulnerability</li> <li>• Parity Multisig Bug</li> <li>• <b>Contract Referencing</b></li> <li>• <b>Efficient Transaction ordering Dependence</b></li> <li>• <b>Tx_origin</b></li> </ul>

**Table 4.1 Vulnerabilities detection in old and enhanced tools.**

#### **4.2.1 External Contract Referencing**

One of the benefits of Ethereum global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.

##### **The Vulnerability**

In Solidity, any address can be cast as a contract regardless of whether the code at the address represents the contract type being cast. This can be deceiving, especially when the author of the contract is trying to hide malicious code. Let us illustrate this with an example:

Consider a piece of code which rudimentarily implements the Rot13 cipher.

This code simply takes a string (letters a-z, without validation) and encrypts it by shifting each character 13 places to the right (wrapping around 'z'); i.e. 'a' shifts to 'n' and 'x' shifts to 'k'.

Consider the following contract which uses this code for its encryption

The issue with this contract is that the encryptionLibrary address is not public or constant. Thus the deployer of the contract could have given an address in the constructor which points to this contract:

which implements the rot26 cipher (shifts each character by 26 places, get it? :p). Again, there is no need to understand the assembly in this contract. The deployer could have also linked the following contract:

**Fig 4.2 Rot13Encryption.sol**

```
contract Private_Bank{
    mapping (address => uint) public balances; uint public MinDeposit = 1 ether; Log TransferLog;
    function Private_Bank(address _log)
    {
        TransferLog = Log(_log);
        function Deposit() public payable{
            if(msg.value >= MinDeposit){
                balances[msg.sender]+=msg.value;
                TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
            }
        }
        function CashOut(uint _am){
            if(_am<=balances[msg.sender]){
                if(msg.sender.call.value(_am)()){
                    balances[msg.sender]-=_am;
                    TransferLog.AddMessage(msg.sender,_am,"CashOut");
                }
            }
        }
        function() public payable{}
    }
}

contract Log {
    struct Message{
        address Sender; string Data; uint Val; uint Time;
    }
    Message[] public History; Message LastMsg;
    function AddMessage(address _adr,uint _val,string _data)
    public{
        LastMsg.Sender = _adr; LastMsg.Time = now; LastMsg.Val = _val; LastMsg.Data = _data; History.push(LastMsg);
    }
}
```

If the address of either of these contracts were given in the constructor, the `encryptPrivateData()` function would simply produce an event which prints the unencrypted private data. Although in this example a library-like contract was set in the constructor, it is often the case that a privileged user (such as an owner) can change library contract addresses. If a linked contract doesn't contain the function being called, the fallback function will execute. For example, with the line `encryptionLibrary.rot13Encrypt()`, if the contract specified by `encryptionLibrary` was: then an event with the text “Here” would be emitted. Thus if users can alter contract libraries, they can in principle get users to unknowingly run arbitrary code.

Note: Don't use encryption contracts such as these, as the input parameters to smart contracts are visible on the blockchain. Also the Rot cipher is not a recommended encryption technique :p



## **Preventative Techniques**

As demonstrated above, vulnerability free contracts can (in some cases) be deployed in such a way that they behave maliciously. An auditor could publicly verify a contract and have its owner deploy it in a malicious way, resulting in a publicly audited contract which has vulnerabilities or malicious intent.

There are a number of techniques which prevent these scenarios.

One technique, is to use the new keyword to create contracts. In the example above, the constructor could be written like:

```
constructor() {  
    encryptionLibrary = new Rot13Encryption();  
}
```

This way an instance of the referenced contract is created at deployment time and the deployer cannot replace the Rot13Encryption contract with anything else without modifying the smart contract.

Another solution is to hard code any external contract addresses if they are known.

In general, code that calls external contracts should always be looked at carefully. As a developer, when defining external contracts, it can be a good idea to make the contract addresses public (which is not the case in the honey-pot example) to allow users to easily examine which code is being referenced by the contract. Conversely, if a contract has a private variable contract address it can be a sign of someone behaving maliciously (as shown in the real-world example). If a privileged (or any) user is capable of changing a contract address which is used to call external functions, it can be important (in a decentralised system context) to implement a time-lock or voting mechanism to allow users to see which code is being changed or to give participants a chance to opt in/out with the new contract address.

## **Real-World Example: Re-Entrancy Honey Pot**

A number of recent honey pots have been released on the main net. These contracts try to outsmart Ethereum hackers who try to exploit the contracts, but who in turn end up getting ether lost to the contract they expect to exploit. One example employs the above attack by replacing an expected contract with a malicious one in the constructor.

### 4.2.2 Efficient Transaction ordering Dependence:

A Transaction-Ordering Attack is a race condition attack. To put it simply if you purchase an item at a price advertised, you expect to pay that price. A transaction-ordering attack will change the price during the processing of your transaction because some one else (the contract owner, miner or another user) has sent a transaction modifying the price before your transaction is complete. Two transactions can be sent to the mempool/tx-pool, the order in which they arrive is irrelevant. The gas sent with the transaction is vital in this scenario as it determines which transaction is mined first. An attacker could also be a miner, as the miner can choose the order in which the transactions are mined. This creates a problem in Smart Contracts that rely on the state of storage variables to remain at certain value according to the order of transactions.

**Fig 4.3 Contract.sol**

```
contract TransactionOrdering {
    uint256 price;
    address owner;
    event Purchase(address _buyer, uint256 _price);
    event PriceChange(address _owner, uint256 _price);
    modifier ownerOnly() {
        require(msg.sender == owner);
    }
    function TransactionOrdering() {
        owner = msg.sender;
        price = 100;
    }
    function buy() returns (uint256) {
        Purchase(msg.sender, price);
        return price;
    }
    function setPrice(uint256 _price) ownerOnly() {
        price = _price;
        PriceChange(owner, price);
    }
}
```

#### Attack Scenario:

1. The buyer of the digital asset will call the buy() function, to set a purchase at the price specified in the storage variable, with a starting price=100.
2. The contract owner will call setPrice() and update the price storage variable to price=150.
3. The contract owner will send the transaction with a higher gas fee.

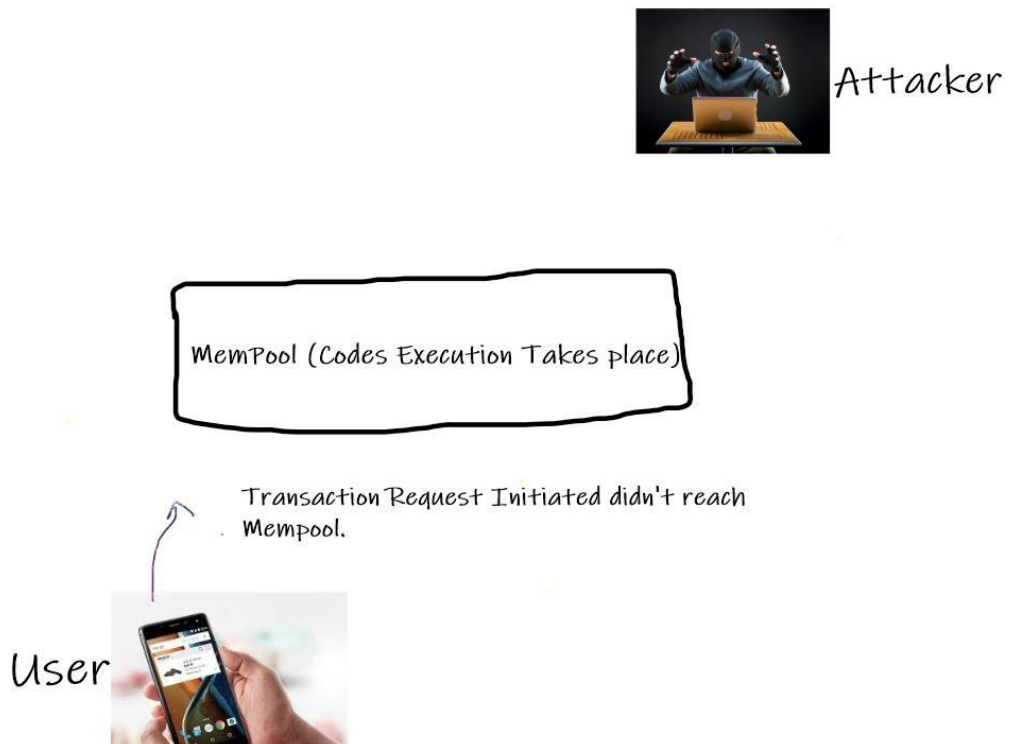
4. The contract owner's transaction will be mined first, updating the state of the contract due to the higher gas fee.
5. The buyers transaction gets mined soon after, but now the buy() function will be using the new updated price=150.

### Existing Solution

A solution is to use a transaction counter to “lock” in the agreed price

But solution of locking is not enough, if the hacker changes the products price at “Point Of Attack” then the user will be cheated.

**Fig 4.4 Point of Attack:**



**Efficient Solution:**

So we need to have a check function in the contract. Once the execution starts in mempool the check function verifies if the product price is changed, if the product price is changed then the current transaction will be aborted and if the price is same as shown to user then the transaction executes successfully.

**4.2.3 Tx\_origin:**

tx.origin is a security vulnerability, breaks compatibility with other contracts including security contracts, and is almost never useful. Removing it would make Solidity more user-friendly. If there are exceptional cases where access to the transaction origin is needed, a library using in-line assembly can provide it.

**The Problem**

1. tx.origin is a security vulnerability. As we recently saw with the Mist wallet, using tx.origin makes you vulnerable to attacks comparable to phishing or cross-site scripting. Once a user has interacted with a malicious contract, that contract can then impersonate the user to any contract relying on tx.origin.
2. tx.origin breaks compatibility. Using tx.origin means that your contract cannot be used by another contract, because a contract can never be the tx.origin. This breaks the general composability of Ethereum contracts, and makes them less useful. In addition, this is another security vulnerability, because it makes security-based contracts like multisig wallets incompatible with your contract.
3. tx.origin is almost never useful. This is the most subjective point, but I have yet to come across a use of tx.origin that seemed legitimate to me. I welcome counter-examples, but I've written dozens or hundreds of smart contracts without needing it, and I have never heard of anyone else needing it either.

**Fig 4.5 Tx\_origin.sol**

```
contract TxUserWallet {  
    address owner;  
  
    constructor() {  
        owner = msg.sender;  
    }  
  
    function transferTo(address dest, uint amount) public {  
        require(tx.origin == owner);  
        dest.transfer(amount);  
    }  
}
```

### **Solutions**

1. Never use tx.origin to check for authorisation of ownership, instead use msg.sender.
2. Don't use address.call.value(amount>(); instead use address.transfer().
3. address.transfer() will have a gas stipend of 2300 — meaning possible attacking contracts would not have enough gas for further computation other than emitting Events.
4. address.transfer() also throws on failure.

### 4.3 ALGORITHMS DESIGNED TO DETECT VULNERABILITIES

The control flow graph will be generated by the tool for a smart contract with basic blocks. A basic block is a straight-line code sequence with no branches into it except for the entry and no branches out of it except at the end. Bytecode and CFG are given as inputs to the algorithms. It returns a boolean value indicating, presence of vulnerability, and program counters responsible for the vulnerabilities.

For detecting Contract Referencing vulnerability, Efficient Transaction Ordering Dependence vulnerability and Tx\_origin vulnerability, the following algorithm has been designed. Time complexity of the algorithm is  $O(n)$ ,  $n$  denotes length of the bytecode file.

**INPUT:** Bytecode of the compiled code

**OUTPUT:** possibility of the vulnerability and line responsible for it.

1. **procedure** checkNewListedVulnerability(bytecode):
2.     **initialize** pcs **as** NULL
3.     **for each** opcode **in** bytecode **do**:
4.         instruction\_line=opcode
5.         instruction\_check=opcode->next->next->next->next
6.         **if** instruction\_check **is** PUSH&&instruction\_check.offset **is** '4' &&  
           instruction\_check.value **is** 'ffffff' **do**
7.             **if** instruction\_check **is** PUSH&&instruction\_check.offset **is** '4' **do**
8.                 pcs.append(opcode.pc)
9.     **for each** opcode **in** bytecode **do**:
10.         instruction\_line=opcode
11.         **if** instruction\_check **is** PUSH&&instruction\_check.offset **is** '4' &&  
           instruction\_check.value **is** '919840ad' **do**
12.             pcs.append(opcode.pc)
13.     **for each** opcode **in** bytecode **do**:
14.         instruction\_line=opcode
15.         **if** instruction\_check **is** PUSH **do**
16.             pcs.append(opcode.pc)
17. **return** pcs

## 5. IMPLEMENTATION AND RESULTS

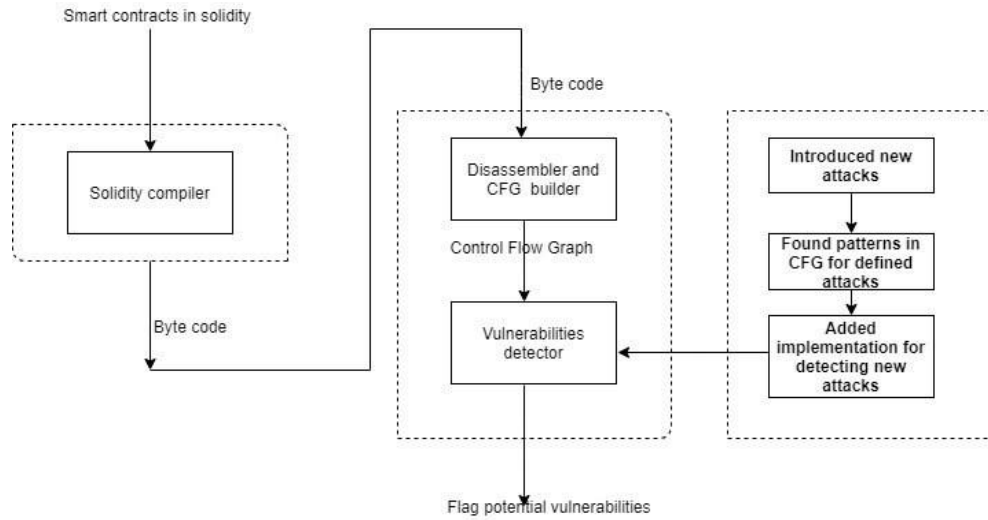
The criteria chosen to test the efficiency and performance of the enhanced tool is by feeding a set of smart contracts to both the existing tool and the enhanced tool. The vulnerabilities detected by both the tools are tabulated and compared to find the improved efficiency and performance of the tool.

For testing purposes, we collected a sample of five hundred smart contracts, which are taken from the existing blockchain. (The smart contracts in the blockchain can be accessed through etherscan.io). The limitation this testing poses is, the inability of the home computer to be able to run the smart contracts consecutively for a continuous period of time, pertaining to the fact that, running smart contracts for its vulnerabilities in OYENTE is a heavy memory and time-consuming process, making the user wait for a brief period of time. This is one of the main reasons for not automating the task of testing, as the system is getting crashed when numerous smart contracts are fed into the tool one after other using python script.

### 5.1 IMPLEMENTATION:

#### The execution flow of Oyente tool:

- Fig. 5.1 shows the complete flow of the execution. The two blocks from the left are common to both the old and enhanced tools. The third block on the right side shows the enhancements made by us. We will discuss the entire flow of execution now.
- The smart contract in fig 5.2 is a sample code having three potential vulnerabilities i.e., DOS with unexpected revert in function defined from line 6, unchecked division in function defined from line 13, unexpected ether in function defined from line 18. The original old tool is not able to flag these potential vulnerabilities.



**Fig 5.1 Complete Flow of the Execution.**

```

1 import "referenced_one.sol";
2 import "referenced_two.sol";
3
4 contract 3AttacksCombine {
5     //Tx_origin
6     address owner;
7     constructor() {
8         owner = msg.sender;
9     }
10    function transferTo(address dest, uint amount) public {
11        require(tx.origin == owner);
12        dest.transfer(amount);
13    }
14    //Efficient Transaction Ordering Dependence
15    uint public price;
16    uint public stock;
17    uint public amount;
18    address public owner;
19    int transaction_successful = 0;
20    function check() {
21        if (msg.value == price)
22            transaction_successful = 1;
23    }
24    function updatePrice ( uint _price ) {
25        if (msg.sender == owner) {
26            price = _price;
27        }
28    }
29
30    function buy ( uint quant ) returns ( uint ) {
31        if ( msg.value < quant*price || quant > stock ) {
32            revert();
33        }
34        check();
35        stock -= quant ;
36    }
37    // External Contract Referencing
38    referenced_one x;
39    function calculate(){
40        x.sum();
41    }
42    referenced_two y;
43    function operate(){
44        y.difference();
45    }
46 }
  
```

**Fig. 5.2 Sample Smart Contract Vulnerable to Attacks.**





1	.code			46	RETURN	contra	91	JUMPI	contra
2	PUSH 80	contract		47	.data		92	PUSH [tag] 1	contract
3	PUSH 40	contract		48	0:		93	JUMP	contract
4	MSTORE	contract		49	.code		94	tag 12	contract
5	PUSH 0	0		50	PUSH 80	co	95	JUMPDEST	contract
6	PUSH 4	int tra		51	PUSH 40	co	96	DUP1	contract
7	SSTORE	int tra		52	MSTORE	co	97	PUSH 2CCB1B30	contract
8	CALLVALUE	con		53	PUSH 4	co	98	EQ	contract
9	DUP1	constructor		54	CALLDATASIZE		99	PUSH [tag] 2	contract
10	ISZERO	constru		55	LT	contra	100	JUMPI	contract
11	PUSH [tag] 1	con		56	PUSH [tag] 1		101	DUP1	contract
12	JUMPI	constru		57	JUMPI	co	102	PUSH 7159A618	contract
13	PUSH 0	constru		58	PUSH 0	co	103	EQ	contract
14	DUP1	constructor		59	CALLDATALOAD		104	PUSH [tag] 3	contract
15	REVERT	constru		60	PUSH E0	co	105	JUMPI	contract
16	tag 1	constructor		61	SHR	contra	106	DUP1	contract
17	JUMPDEST	constru		62	DUP1	contra	107	PUSH 8D6CC56D	contract
18	POP	constructor		63	PUSH A035B1FE		108	EQ	contract
19	CALLER	msg.sen		64	GT	contra	109	PUSH [tag] 4	contract
20	PUSH 0	owner		65	PUSH [tag] 12		110	JUMPI	contract
21	DUP1	owner		66	JUMPI	co	111	DUP1	contract
22	PUSH 100	owner =		67	DUP1	contra	112	PUSH 8DA5CB5B	contract
23	EXP	owner = msg		68	PUSH A035B1FE		113	EQ	contract
24	DUP2	owner = msg		69	EQ	contra	114	PUSH [tag] 5	contract
25	SLOAD	owner =		70	PUSH [tag] 7		115	JUMPI	contract
26	DUP2	owner = msg		71	JUMPI	co	116	DUP1	contract
27	PUSH FFFFFFFFFFFFFFFFFF			72	DUP1	contra	117	PUSH 919840AD	contract
28	MUL	owner = msg		73	PUSH AA8C217C		118	EQ	contract
29	NOT	owner = msg		74	EQ	contra	119	PUSH [tag] 6	contract
30	AND	owner = msg		75	PUSH [tag] 8		120	JUMPI	contract
31	SWAP1	owner =		76	JUMPI	co	121	tag 1	contract
32	DUP4	owner = msg		77	DUP1	contra	122	JUMPDEST	contract
33	PUSH FFFFFFFFFFFFFFFFFF			78	PUSH BDF3C4AE		123	PUSH 0	contract
34	AND	owner = msg		79	EQ	contra	124	DUP1	contract
35	MUL	owner = msg		80	PUSH [tag] 9		125	REVERT	contract
36	OR	owner = msg		81	JUMPI	co	126	tag 2	function
37	SWAP1	owner =		82	DUP1	contra	127	JUMPDEST	function
38	SSTORE	owner =		83	PUSH CA77AB65		128	PUSH [tag] 13	function
39	POP	owner = msg		84	EQ	contra	129	PUSH 4	function
40	PUSH #[\$] 0000000000000000			85	PUSH [tag] 10		130	DUP1	function
41	DUP1	contract ov		86	JUMPI	co	131	CALLDATASIZE	function
42	PUSH [\$] 0000000000000000			87	DUP1	contra	132	SUB	function
43	PUSH 0	contract		88	PUSH D96A094A		133	DUP2	function
44	CODECOPY	contract		89	EQ	contra	134	ADD	function
45	PUSH 0	contract		90	PUSH [tag] 11		135	SWAP1	function
46	RETURN	contract		91	JUMPI	co	136	PUSH [tag] 14	function
47	.data			92	PUSH [tag] 1		137	SWAP1	function

140	JUMP [in]	func	184	SUB	func	229	DUP1	address
141	tag 14	function	185	DUP2	func	230	SWAP2	add
142	JUMPDEST	func	186	ADD	func	231	SUB	address
143	PUSH [tag] 16	func	187	SWAP1	fu	232	SWAP1	add
144	JUMP [in]	func	188	PUSH [tag] 22	fu	233	RETURN	add
145	tag 13	function	189	SWAP2	fu	234	tag 6	function
146	JUMPDEST	func	190	SWAP1	fu	235	JUMPDEST	fun
147	STOP	function	191	PUSH [tag] 23	fu	236	PUSH [tag] 30	fun
148	tag 3	function	192	JUMP [in]	fu	237	PUSH [tag] 31	fun
149	JUMPDEST	func	193	tag 22	function	238	JUMP [in]	fun
150	CALLVALUE	func	194	JUMPDEST	fu	239	tag 30	function
151	DUP1	function	195	PUSH [tag] 24	fu	240	JUMPDEST	fun
152	ISZERO	func	196	JUMP [in]	fu	241	STOP	function
153	PUSH [tag] 17	func	197	tag 21	function	242	tag 7	uint pu
154	JUMPI	func	198	JUMPDEST	fu	243	JUMPDEST	uin
155	PUSH 0	func	199	STOP	func	244	CALLVALUE	uin
156	DUP1	function	200	tag 5	address	245	DUP1	uint pu
157	REVERT	func	201	JUMPDEST	ac	246	ISZERO	uin
158	tag 17	function	202	CALLVALUE	ac	247	PUSH [tag] 32	uin
159	JUMPDEST	func	203	DUP1	address	248	JUMPI	uin
160	POP	function	204	ISZERO	ac	249	PUSH 0	uin
161	PUSH [tag] 18	func	205	PUSH [tag] 25	ac	250	DUP1	uint pu
162	PUSH [tag] 19	func	206	JUMPI	ac	251	REVERT	uin
163	JUMP [in]	func	207	PUSH 0	ac	252	tag 32	uint pu
164	tag 18	function	208	DUP1	address	253	JUMPDEST	uin
165	JUMPDEST	func	209	REVERT	ac	254	POP	uint pu
166	STOP	function	210	tag 25	address	255	PUSH [tag] 33	uin
167	tag 4	function	211	JUMPDEST	ac	256	PUSH [tag] 34	uin
168	JUMPDEST	func	212	POP	address	257	JUMP [in]	uin
169	CALLVALUE	func	213	PUSH [tag] 26	ac	258	tag 33	uint pu
170	DUP1	function	214	PUSH [tag] 27	ac	259	JUMPDEST	uin
171	ISZERO	func	215	JUMP [in]	ac	260	PUSH 40	uin
172	PUSH [tag] 20	func	216	tag 26	address	261	MLOAD	uin
173	JUMPI	func	217	JUMPDEST	ac	262	PUSH [tag] 35	uin
174	PUSH 0	func	218	PUSH 40	ac	263	SWAP2	uin
175	DUP1	function	219	MLOAD	ac	264	SWAP1	uin
176	REVERT	func	220	PUSH [tag] 28	ac	265	PUSH [tag] 36	uin
177	tag 20	function	221	SWAP2	ac	266	JUMP [in]	uin
178	JUMPDEST	func	222	SWAP1	ac	267	tag 35	uint pu
179	POP	function	223	PUSH [tag] 29	ac	268	JUMPDEST	uin
180	PUSH [tag] 21	func	224	JUMP [in]	ac	269	PUSH 40	uin
181	PUSH 4	func	225	tag 28	address	270	MLOAD	uin
182	DUP1	function	226	JUMPDEST	ac	271	DUP1	uint pu
183	CALLDATASIZE	func	227	PUSH 40	ac	272	SWAP2	uin
184	SUB	function	228	MLOAD	ac	273	SUB	uint pu
185	DUP2	function	229	DUP1	address	274	SWAP1	uin
186	ADD	function	230	SWAP2	ac	275	RETURN	uin

1000	SWAP2	1049	JUMPDEST	1089	MSTORE
1001	POP	1050	PUSH 0	1090	PUSH 11
1002	PUSH [tag] 112	1051	PUSH [tag] 120	1091	PUSH 4
1003	DUP4	1052	DUP3	1092	MSTORE
1004	PUSH [tag] 99	1053	PUSH [tag] 117	1093	PUSH 24
1005	JUMP [in]	1054	JUMP [in]	1094	PUSH 0
1006	tag 112	1055	tag 120	1095	REVERT
1007	JUMPDEST	1056	JUMPDEST	1096	tag 80
1008	SWAP3	1057	SWAP1	1097	JUMPDEST
1009	POP	1058	POP	1098	PUSH [tag] 125
1010	DUP3	1059	SWAP2	1099	DUP2
1011	DUP3	1060	SWAP1	1100	PUSH [tag] 118
1012	LT	1061	POP	1101	JUMP [in]
1013	ISZERO	1062	JUMP [out]	1102	tag 125
1014	PUSH [tag] 113	1063	tag 117	1103	JUMPDEST
1015	JUMPI	1064	JUMPDEST	1104	DUP2
1016	PUSH [tag] 114	1065	PUSH 0	1105	EQ
1017	PUSH [tag] 109	1066	PUSH FFFFFFFFFFFFFFFFFFFFFFFFFF	1106	PUSH [tag] 126
1018	JUMP [in]	1067	DUP3	1107	JUMPI
1019	tag 114	1068	AND	1108	PUSH 0
1020	JUMPDEST	1069	SWAP1	1109	DUP1
1021	tag 113	1070	POP	1110	REVERT
1022	JUMPDEST	1071	SWAP2	1111	tag 126
1023	DUP3	1072	SWAP1	1112	JUMPDEST
1024	DUP3	1073	POP	1113	POP
1025	SUB	1074	JUMP [out]	1114	JUMP [out]
1026	SWAP1	1075	tag 99	1115	tag 84
1027	POP	1076	JUMPDEST	1116	JUMPDEST
1028	SWAP3	1077	PUSH 0	1117	PUSH [tag] 128
1029	SWAP2	1078	DUP2	1118	DUP2
1030	POP	1079	SWAP1	1119	PUSH [tag] 99
1031	POP	1080	POP	1120	JUMP [in]
1032	JUMP [out]	1081	SWAP2	1121	tag 128
1033	tag 95	1082	SWAP1	1122	JUMPDEST
1034	JUMPDEST	1083	POP	1123	DUP2
1035	PUSH 0	1084	JUMP [out]	1124	EQ
1036	PUSH [tag] 116	1085	tag 109	1125	PUSH [tag] 129
1037	DUP3	1086	JUMPDEST	1126	JUMPI
1038	PUSH [tag] 117	1087	PUSH 4E487B710000000000000000	1127	PUSH 0
1039	JUMP [in]	1088	PUSH 0	1128	DUP1
1040	tag 116	1089	MSTORE	1129	REVERT
1041	JUMPDEST	1090	PUSH 11	1130	tag 129
1042	SWAP1	1091	PUSH 4	1131	JUMPDEST
1043	POP	1092	MSTORE	1132	POP
1044	SWAP2	1093	PUSH 24	1133	JUMP [out]
1045	SWAP1	1094	PUSH 0	1134	.data
1046	POP	1095	REVERT	1135	

**Fig 5.4 Assembly Level Opcode File generated by Dis-assembler.**

- The Control Flow Graph is then fed to the vulnerability detector. Based on the algorithm we have devised before for finding a vulnerability, the vulnerability detector runs through the Control Flow Graph for finding the potential cases.

- We have identified some vulnerabilities that the old version of the Oyente cannot flag. We studied these new vulnerabilities, devised algorithms, and implemented them.
- CFG of the input smart contracts are analyzed manually and if the pattern is found, it implies the existence of the vulnerability.
- After these vulnerabilities are flagged, we report them with their respective source code lines in the smart contract.
- The various blocks of code used for the detection of vulnerabilities are embedded in the source Oyente code.
- We can execute smart contracts on the Oyente tool in two ways. The following commands need to be executed in the terminal:
  1. If the input smart contract is a solidity file:  
**oyente.py -s contract.sol**
  2. If the input smart contract is a byte code:  
**oyente.py -s contract.evm -b**
- After the execution is done, the results for the execution are displayed. If the specific vulnerability exists, then the vulnerability is flagged as 'True'. if the vulnerability does not exist, it is flagged as 'False'.

```

/usr/bin/python2.7 /home/shaz/Downloads/oyente/oyente.py
WARNING:root:You are using solc version 0.4.24, The latest supported version is 0.4.19
INFO:root:contract tx_origin.sol:TxUserWallet:
INFO:symExec:  ===== Results =====
INFO:symExec:      EVM Code Coverage:          94.4%
INFO:symExec:      Integer Underflow:             False
INFO:symExec:      Integer Overflow:                False
INFO:symExec:      Parity Multisig Bug 2:          False
INFO:symExec:      Callstack Depth Attack Vulnerability: False
INFO:symExec:      Transaction-Ordering Dependence (TOD): False
INFO:symExec:      Timestamp Dependency:             False
INFO:symExec:      Re-Entrancy Vulnerability:           False
INFO:symExec:      ===== Analysis Completed =====
Process finished with exit code 0

```

**Fig. 5.5 The Result of the Execution of the Original Tool**

```

/usr/bin/python2.7 /home/shaz/Downloads/oyente/oyente.py
WARNING:root:You are using solc version 0.4.24, The latest supported version is 0.4.19
INFO:root:contract tx_origin.sol:TxUserWallet:
INFO:symExec: ===== Results =====
INFO:symExec:      EVM Code Coverage:          94.4%
INFO:symExec:      Integer Underflow:          False
INFO:symExec:      Integer Overflow:           False
INFO:symExec:      Parity Multisig Bug 2:       False
INFO:symExec:      Transaction Ordering Dependence Efficient:  True
INFO:symExec:      Tx_Origin:                  True
INFO:symExec:      External Contract Referencing:  True
INFO:symExec:      Callstack Depth Attack Vulnerability:  False
INFO:symExec:      Transaction-Ordering Dependence (TOD):  False
INFO:symExec:      Timestamp Dependency:           False
INFO:symExec:      Re-Entrancy Vulnerability:       False
INFO:symExec:      ===== Analysis Completed =====

Process finished with exit code 0

```

**Fig 5.6 The Result of the Execution of the Enhanced Tool.**

- The final results of both the old and new tools are shown in fig 5.5 and fig 5.6. Fig 5.5 shows results when the smart contract is run on the old tool. Similarly, fig 5.6 shows the results when the smart contract is run on the new tool.
- The old tool was not able to flag the three vulnerabilities whereas the enhanced tool is able to do.

The smart contracts are given as input to both the old and new tools as byte code of the respective smart contracts. For the commercially implemented smart contracts are available in public as solidity code along with byte code. The metrics taken into consideration for analyzing performance are :

### Metrics

- **Number of vulnerabilities detected:** Sum of all vulnerabilities in all the smart contracts, in the original tool it includes, the sum of all the existing vulnerabilities detected. In enhanced tool, it includes the sum of all existing vulnerabilities and newly added vulnerabilities detected.
- **Number of smart contracts detected as vulnerable :** If a single vulnerability exists in a smart contract, then the smart contract is considered vulnerable. In the original tool, a smart contract is vulnerable, if existing vulnerabilities are detected in the smart contract. In the



enhanced tool, a smart contract is vulnerable, if any of the existing vulnerabilities are detected as well as if newly added vulnerabilities are also detected.

- **External Contract Referencing vulnerability:** Count of External Contract Referencing vulnerabilities in all the smart contracts.
- **TOD Efficient vulnerability:** Count of TOD Efficient vulnerabilities in all the smart contracts.
- **Tx\_origin vulnerability:** Count of Tx\_origin vulnerability in all the smart contracts.

## 5.2 RESULTS:

Metrics	Old Tool	Enhanced Tool	Percentage Increase in performance
Number of vulnerabilities detected	385	424	10.13%
Number of smart contracts detected as vulnerable	192	207	7.81%
External Contract Referencing	0	12	NA (newly added)
TOD Efficient	0	25	NA (newly added)
Tx_origin	0	2	NA (newly added)

**Table 5.1 Comparison of performance between old and enhanced tools.**

As shown in the above table 1, there is a 10 percent increase in the total number of vulnerabilities detected by the enhanced Oyente tool for the pool of tested 500 smart contracts. The increase in the number of smart contracts detected as vulnerable is slightly less than 8 percent.

The newly detected vulnerabilities:

1. The External Contract Referencing is flagged in the enhanced tool in 12 instances of the total 500 smart contracts,
2. TOD Efficient vulnerability is flagged in the enhanced tool in 25 instances,
3. Tx\_origin is flagged in 2 instances.

The other metric taken into consideration to analyze the performance of the tool is the recall of the newly detected vulnerabilities.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True positive} + \text{False negative}}$$

For this, 64 commercially implemented contracts, whose solidity code is available to the public. The source of these smart contracts are well-maintained GitHub repositories and extensively researched blogs based on the security of smart contracts. The solidity code is required to verify if the tool is actually flagging all possible instances of the vulnerabilities.

<b>Vulnerability</b>	<b>Total detected</b>	<b>Undetected</b>	<b>Recall</b>
External Contract Referencing	8	3	0.72
TOD Efficient	5	1	0.83
Tx_origin	2	0	1

**Table 5.2 Recall of the newly detected vulnerabilities.**

As shown in Table 2, these are the recall values for vulnerabilities:

1. As the solidity code is available we are able to verify from the 64 smart contracts that recall of Tx\_origin vulnerability is 1, implying no vulnerability went undetected.
2. Recall for the External Contract Referencing was 0.72.
3. The recall of the TOD Efficient vulnerability is found to be 0.83.



## **6. SUMMARY AND CONCLUSION**

### **6.1 SUMMARY:**

In this work, we look at Ethereum smart contracts from a security viewpoint. The project focuses on increasing pre-emptive security of the smart contracts that will be deployed in the Ethereum blockchain. We started by studying the various security vulnerabilities and introduced new vulnerabilities, studied them, devised algorithms for their detection, and implemented them. The tool under consideration for providing security to the implemented smart contracts is Oyente, a security testing tool for smart contracts developed by professionals from the National University Of Singapore. The primary step we undertook going forward is to understand the context of the smart contracts, their implementation, and the scope of security in implementing the smart contracts.

We undertook the installation of the Oyente tool to the local system and we were able to make changes to the tool to adapt for it to accommodate the changes made into the source code. The tool is made to be editable, thereby giving us the scope to add the ability to detect new vulnerabilities on the top of the existing detecting capabilities. The working and flow of the execution in the Oyente tool, when a smart contract is given as input into the tool is analyzed.

We introduced three new vulnerabilities: External Contract Referencing, TOD Efficient, Tx\_origin, which cannot be detected by the original Oyente tool. Also, we made improvements to the implementation of existing vulnerabilities bearing false negatives. We chose a sample of smart contracts and gave them as input to the original Oyente tool and we analyzed the CFG's for devising algorithms. Based on the patterns observed in the CFG's of various smart contracts, we designed an algorithm for every new vulnerability that the enhanced Oyente detects.

The performance of the tool is analyzed by comparing the results we got by giving the same set of 500 smart contracts to both the original and enhanced tools. The results obtained from both the tools are compared to analyze the increase in performance; ability to detect vulnerabilities and ability to flag that smart contract is vulnerable or not.

### **6.2 CONCLUSION:**

Ethereum allows us to write smart contracts. As the contracts are on the blockchain, they become immutable making them attractive for various decentralized applications. However, the biggest

advantage of smart contracts - their immutability also poses the biggest threat from a security standpoint as any bug once found cannot be resolved and results in massive monetary losses. In such a scenario it becomes imperative to develop and interact with secure smart contracts.

We used a well-known static analysis tool called Oyente in detecting bugs. In this thesis, we analyzed the smart contracts from a security viewpoint. We have successfully implemented the ability to detect three new vulnerabilities: External Contract Referencing, TOD Efficient, Tx\_origin to the existing Oyente tool by understanding the nature of each vulnerability.

Though there are some downfalls to the modified detection algorithm, it sure surpassed the detection capabilities of the existing algorithm. The algorithm for it is deeply embedded in the code, so to rewrite the entire algorithm for it, is to write the entire tool from scratch, and that is beyond the scope of the project we undertook.

A minor improvement is also made in the installation procedure of Oyente. The majority of the users use Oyente using docker, which prevents the user from making changes to the Oyente source files. To make the tool editable, we collected information from various sources and formulated a concrete step by step procedure for installing the Oyente tool, that is native to the user's machine.

### **6.3 SCOPE FOR FUTUREWORK:**

As we succeeded in defining a procedure to make Oyente a native tool to the user, any interested user can pick a vulnerability from the ever-growing list of vulnerabilities and attacks. The user can devise algorithms for flagging potential vulnerabilities. Also, users can improve the performance by decreasing the false positivity rate.

## REFERENCES

- [1] **Leslie Lamport, Robert Shostak, and Marshall Pease** . The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):382– 401, 1982.
- [2] **Satoshi Nakamoto** et al. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [3] **Ethereum Foundation**. Ethereum’s white paper. <https://github.com/ethereum/wiki/wiki/White-Paper>, 2014.
- [4] **Ayelet Sapirshtein, Yonatan Sompolsky, and Aviv Zohar**. Optimal selfish mining strategies in bitcoin. In International Conference on Financial Cryptography and Data Security, pages 515–532. Springer, Christ Church, Barbados, 2016.
- [5] **Jianyu Niu and Chen Feng**. Selfish Mining in Ethereum. arXiv e-prints, Jan 2019.
- [6] **ConsenSys**. Ethereum Smart Contract Best Practices – **Known Attacks**. [https://consensys.github.io/smart-contract-best-practices/known\\_attacks/](https://consensys.github.io/smart-contract-best-practices/known_attacks/)
- [7] **Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor**. Making smart contracts smarter. In Proceedings of the ACM SIGSAC conference on computer and communications security, pages 254–269. ACM, Vienna Austria, 2016.
- [8] **Elvira Albert, Pablo Gordillo, Benjamin Livshits , Albert Rubio, and Ilya Sergey**. Ethir: A framework for high-level analysis of ethereum bytecode. In International Symposium on Automated Technology for Verification and Analysis, pages 513–520. Springer, Los Angeles, CA, USA, 2018.
- [9] **ConsenSys**. **Mythril Classic**. <https://github.com/ConsenSys/mythril-classic>.
- [10] **James C. King**. Symbolic execution and program testing. Commun. ACM, 19(7):385–394.
- [11] **ConsenSys Known Attacks**. [https://consensys.github.io/smart-contract-best-practices/known\\_attacks/](https://consensys.github.io/smart-contract-best-practices/known_attacks/)
- [12] **V Saini, HackPedia**: 16 solidity hacks/vulnerabilities, their fixes and real world examples, 2018.
- [13] **Solidity documentation**. <https://solidity.readthedocs.io/en/v0.6.9/types.html>.
- [14] **Github**. <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol>

- [15] **Github** <https://github.com/runtimeverification/verified-smart-contracts/wiki/List-of-Security-Vulnerabilities>
- [16] **Tx-origin Documentation** <https://docs.soliditylang.org/en/develop/security-considerations.html#tx-origin>
- [17] **Detection of Vulnerabilities in Smart Contracts Specifications in Ethereum Platforms**  
<https://drops.dagstuhl.de/opus/volltexte/2020/13015/pdf/OASICS-SLATE-2020-2.pdf>