Symbolic Execution Tool to find DoS Vulnerabilities in Ethereum Smart Contract

A thesis submitted in fulfillment of the requirements

for the degree of Master of Technology

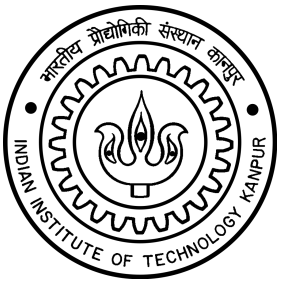
by

Deepak Yadav

18111014

under the guidance of

Prof. Sandeep Sukla



to the

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

JUN 2020

Page intentionally left blank

DECLARATION

This is to certify that the thesis titled **"Symbolic Execution Tool to find DoS Vulnerabilities in Ethereum Smart Contract"** has been authored by me. It presents the research conducted by me under the supervision of **Prof. Sandeep Sukla**.

To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

Name: Deepak Yadav

Roll No.: 18111014

Programme: MTech

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

Kanpur, 208016.

Page intentionally left blank

Certificate

It is certified that the work contained in the thesis titled **"Symbolic Execution Tool to find DoS Vulnerabilities in Ethereum Smart Contract"** has been carried out under my supervision by **Deepak Yadav** and that this work has not been submitted elsewhere for a degree.

Sandeep K. Shukla

Digitally signed by Sandeep K. Shukla
DN: cn=Sandeep K. Shukla, o=Indian Institute of Technology Kanpur, ou=Department of CSE, email=sandeeps@cse.iitk.ac.in, c=IN
Date: 2020.06.24 11:28:31 +05'30'

Prof. Sandeep Sukla

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

Kanpur, 208016.

JUN 2020

Page intentionally left blank

ABSTRACT

Name of student: **Deepak Yadav** Roll no: **18111014**

Degree for which submitted: **Master of Technology**

Department: **Department of Computer Science and Engineering**

Thesis title: **Symbolic Execution Tool to find DoS Vulnerabilities in Ethereum Smart Contract**

Name of Thesis Supervisor: **Prof. Sandeep Sukla**

Month and year of thesis submission: **JUN 2020**

Ethereum is the second most valuable cryptocurrency after bitcoin. The applications that can be built on Ethereum are countless. This was possible because of the introduction of the concept of smart contracts on ethereum. These smart contracts are immutable because they are stored on blockchain. The immutable property is of great use for eradicating the trust issue when two parties agree to communicate using a smart contract. Smart contracts can work along with other smart contracts for example finishing one smart contract can trigger the execution of another smart contract etc. Despite so many attractive promises made by smart contracts, they can also be prone to problems. The immutability property which is of great use also poses issues in case a smart contract contains some bugs. The code of smart contract logic must be accurate and bug free. Otherwise, it can be exploited for the wrongdoings by the scammers. For example, the DAO hack[3] is an example of such wrongdoings. The other major issue that arises because of such exploitation is that people will lose interest in the blockchain concept. Since this is a fairly new technology a good number of bugs are exposed by the community. Tools reporting various vulnerabilities are build in recent times to aid developers in developing bug-free and accurate functionality of smart contracts.

This thesis aims to build a tool that will flag DoS Vulnerable smart contracts. The tool uses symbolic execution to traverse all the possible paths in the smart contract. The tool test smart contracts for a different number of invocations and variable path length that is traversed by symbolic execution. The symbolic executions and the path length are flexible parameters provided as input along with the smart contract to the tool. To test, the tool is run on a collected sample of 35 smart contracts with invocations 1 and 2 and path length 10, 20, and 40 respectively for both the number of invocations.

Page intentionally left blank

Acknowldgements

I would like to express my sincere gratitude to Prof. Sandeep K. Shukla for his constant support and guidance. Blockchain Technology was a new topic for me and his critical and useful suggestions during the course of this thesis were extremely beneficial.

I am thankful to my lab-mates and class-mates for listening to my doubts and for providing me with constructive feedback.

I am thankful to my family for their constant encouragement and support for the past two years.

Deepak Yadav

Contents

List of Figures

Page intentionally left blank

Chapter 1

Introduction

The term smart contracts, in an informal definition, can be interpreted as a legal piece of agreement between two parties without the involvement of the third party inter mediators. With the arrival of Smart, we can say blockchain has been upgraded from Blockchain 1.0 to Blockchain 2.0. Blockchain 1.0 was only used for financial transactions. In Blockchain 2.0, with smart contracts, we can now mostly implement any idea, business model, games, multiple party agreements like buying a house without the burden of notary works, etc, and many of which we can think of. Ethereum smart contracts are executed on ethereum virtual machine(EVM) which is run by all the nodes in the distributed network blockchain. All nodes agree on how EVM should behave and hence when execute a smart contract with the same parameters all nodes get the same results. The EVM agreement is used as a part of a consensus mechanism so that all nodes eventually agree on the same output of a smart contract. Since smart contracts are stored on the blockchain, they cannot be manipulated and changed once deployed. This property though useful is also a cause of the problem if smart contract logic is not implemented correctly[44].

Smart Contracts were first introduced by an American Scientist and Cryptographer, Nick Szabo, in the early 1990s[22]. However, smart contracts gained popularity with the introduction of ethereum. Bitcoin also supports smart contracts, but programming in bitcoin is very constrained and limited. Bitcoin scripting language is not Turing complete, it only allows some specific kinds of the program to code and runs[43]. But unlike Bitcoin, ethereum can do much better. Developers can use ethereum to build exciting and a wide variety of applications which are called as distributed applications(dApps) because they are deployed on a distributed blockchain network. The ethereum community is the largest and most active blockchain community in the world. Thousands of developers are developing new and innovative ideas on ethereum, many of which we can use today. Hence smart contracts on ethereum are increasing at a faster pace. In fact, the number of smart contracts deployed on the Ethereum network reached near 2M in March 2020[37]. The total supply

of Ether has crossed 100M mark in April 2020[7].

Smart contracts are immutable as long as blockchain integrity is not compromised. Since we cannot change smart contracts once deployed, any security vulnerabilities due to coding logic in the smart contracts will be visible to the public and cannot be patched. Hackers/attackers have exploited many such vulnerabilities in the past. Due to these exploitations, damage to some million dollars worth ether happened already. The bigger issue is the trust of the public. Suppose a company/developer deployed a faulty contract on ethereum and it is attacked, then the people no longer trust that company/developer. Hence during coding, the developers have to be very careful and have to save his/her smart contract from exploitation at-least because of the known vulnerabilities. To facilitate the developers, a good number of tools using various approaches, eg-static analysis, symbolic analysis, etc are already built to check for various vulnerabilities. Every tool looks for a specific set of vulnerabilities in the smart contracts. This thesis focused on the DoS vulnerabilities of the smart contract in our tool. The main goal of this thesis is to build a symbolic analysis tool that will check for the Denial of Service vulnerabilities in smart contracts.

The thesis is divided as follows: Chapter 2 provides background for various concepts necessary to understand the thesis work. Chapter 3 introduces various DoS vulnerabilities found in the past and existing related work related to denial of service. Chapter 4 introduces the symbolic execution approach for the Dos vulnerabilities and various DoS vulnerabilities patterns which are summarised from the DoS vulnerabilities explained in Chapter 3. Chapter 5 explains the working of the tool and explains all components of the tool in detail. Chapter 6 shows the result of the smart contract and analysis of false positives of the tool. Chapter 7 summarises the thesis and the future scope of the thesis.

Chapter 2

Background

2.1 Ethereum

The ethereum system went live on 30 July 2015[8]. Like bitcoin, ethereum also has its own cryptocurrency, ether. Ethereum is an open source, blockchain based distributed computing system. Ethereum is the second largest cryptocurrency in terms of market capitalization after the bitcoin. While all other blockchains also can process code, most of them have a hard constraint on what can be developed what cannot. Ethereum on the other hand allows developers to create whatever operations/logic they want. This opens a wide area to explore and we can see more exciting and innovative Apps and concepts on ethereum. For example, we can develop games, manage financial transactions, building a multi-party system, and many more.

Smart contract on ethereum can let developers and organizations code their Apps(called as Distributed Apps because they are built on a distributed blockchain network, ethereum). These smart contracts are like computer programs and execute the exact functionality/logic using EVM. These apps once deployed is opened to the public and cannot be tampered with because of the blockchain property. These attractive promises of ethereum in terms of smart contracts are very appealing and attracting people to use ethereum. Due to the increasing popularity of ethereum many companies and developers have implemented many new attractive and innovative ideas on ethereum blockchain. The variety of projects that use Ethereum as their foundation is almost limitless. The State of the dApps website[15] shows how many Ethereum projects exist in different categories and how popular they are. Some sample examples of how ethereum can be used shown here[46] are listed below:

- Gnosis[16] and Augur[27] are many of the innovative companies using ethereum in interesting ways. Both provide a platform for the prediction markets. Users use company-specific tokens

to predict and wins token for successful predictions. These predictions are useful for the others who are looking for investment in stocks etc.

- We can also find some games as well in the ethereum space. Cryptokitties[2] is one of the ethereum first games which is still popular. The game introduced blockchain-based crypto-collectibles. Cryptokitties are breed from the cattributes of their parents. The popularity of games can be seen from the fact that some of the cryptokitties are sold for over $10000 USD in the past.

- Ethereum platform is also used to make a social platform to share thoughts. EtherTweet[13] is the blockchain based alternative to twitter.

- Ethereum can be used to search for jobs using EthLance[14]. EthLance is a distributed platform for freelancers and employers to find each other, engage in jobs, and transfer payment in ether, and so on.

and many more...

The transaction rate on ethereum is around 7-8 lakhs per day[9]. The total supply of Ether was around 100 million as of 16 April 2020[7] and around 2M smart contracts already deployed on ethereum[37]. With so many smart contracts deployed and money involved, it attracts the attackers to find and exploit vulnerabilities in the smart contracts. We have seen many vulnerabilities in the past because of which there was a huge loss of money/ether[4]. Few famous vulnerabilities are listed below.

- The reentrancy bug was a shock to the community and it leads to the first hard fork on the chain. Reentrancy as the name is, the attacker reenters the smart contract from the previous execution. The attacker calls the smart contract once and during the execution the called contract sends ether to the caller. Here is the main twist, now the attacker can write a faulty fallback by calling the smart contract through its fallback function. Hence before completion of previous the execution, the attacker keeps on entering the smart contract. The famous DAO attack[3] is an example of the reentrancy attack which costs around 50 USD at the time.

- Access control issues which allowed any external user to perform privilege operations. Suppose a caller to the smart contract gets access to sensitive data which was leaked to the caller by some faulty logic implementation of the smart contract. This vulnerability cost around 30M USD at the time. The real-world impacts due to access control are Parity Bug 1[47], Rubixi[29], and Multi-sig Parity Bug 2[48], etc.

- DoS can take a smart contract offline forever because a smart contract once deployed cannot be changed. Hence this vulnerability is more dangerous than others. There are many ways that are found in the past that are responsible for the denial of service in the ethereum smart contracts. For example, a smart contract can behave maliciously it receives an invocation/call and therefore denying the functionality for which the smart contract was intended, increasing block gas limit by some loops manipulation, etc. The total estimated loss because of Denial of Service is around 300M USD at the time. The real-world impact due to DoS vulnerabilities are GovernMental[51], King of Ether contract[21], and Parity Multi-sig Wallet[48], etc.

2.2 Smart Contracts

Smart contracts are terms and conditions which the two parties agreed upon before doing business with each other[26]. Smart contracts enforce the terms and conditions programmatically and thus removes middle man who in the absence of a smart contract is the main enforcer of the terms and conditions. The middle man sometimes can favor one person over the other and can change the terms and conditions to get some money or cheat the other person. Smart contract reduces this fraud and removes the issue of trust because these are computer programs which runs exactly programmed and cannot be changed if they are implemented on blockchain. A smart contract can work on its own and it can also be implemented along with any number of smart contracts. For example, a smart contract can be called from another smart contract, successful completion of one contract can trigger the second and so. We can make the complete organization run on a smart contract.

Smart contracts give a number of benefits. Smart contract eradicate the need for the third party involved in the transaction. They provide trust as no one can steal them and are stored on a public ledger which can be seen by everyone and is verified by 100's or 1000's of nodes. Smart contracts save money by eliminating the notary, estate agents, assistance, etc charges. Smart contracts, if implemented correctly, provides safety as these are very hard to hack. Smart contracts also save a lot of time which will otherwise be wasted on manually processing lots of paperwork, sending and receiving transportation time, etc.

Smart contracts were first proposed in the early 1990s by a computer scientist, lawyer, and cryptographer, Nick Szabo. However, the implementation of smart contracts did not happen until 2009 when the first cryptocurrency blockchain bitcoin came and provides a suitable environment for the implementation of smart contracts. But their implementation still is very much regulated and controlled by the bitcoin. But with the arrival of ethereum, the smart contract gets more freedom because of the Turing completeness of EVM and become the most hyped feature of ethereum.

Smart contracts can now implement almost all practical life examples.
Smart contracts are an extremely new technology. Despite so many attractive promises, they can still be prone to problems. The code of the smart contract logic must be accurate and contains no bugs. This can lead to mistakes and most of the time are exploited by attackers for the wrongdoing to the system/ smart contracts. For example, the DAO hack, King of Ether vulnerability are examples of ethereum smart contracts where scammers found vulnerabilities in the smart contract logic and exploited them.

2.3 Ethereum Smart Contract Programming

Smart contracts after compilation are converted into bytecode and these bytecodes are executed on EVM. The popular programming languages that are used for writing smart contracts on Ethereum are Solidity and Vyper. Solidity is a more popular language among the two and is inspired by C++, python, and javascript. Vyper is based on python. The IDE's that are listed on official ethereum website[6] for the ethereum developments are Visual Studio Code with official ethereum support, Remix is a web-based IDE with built-in static analysis, and a test blockchain virtual machine and EthFiddle is also a web-based IDE to write, compile and debug smart contracts. We have used Remix to generate bytcodes for our sample test smart contracts.

2.4 Bytecode

When we compile the high-level smart contract code, say for example a solidity file on Remix, it will translate the high-level code into the bytecode. The bytecode is a hexadecimal representation of the smart contract. Since this bytecode will run on the EVM, the EVM have predefined opcodes. The bytecode contains these opcodes into their hexadecimal values. Every opcode has a corresponding hexadecimal value. The mapping of hexadecimal values to the opcodes can be seen in the ethereum yellow paper and here. EVM is a stack-based machine and every opcode takes arguments from the stack and processes on those arguments and puts back the result on the stack. There are some opcodes that put data on the stack without processing. Let's look at these opcodes below.

- **Opcodes of execution environment**: For example PC for program counter and MSIZE for current memory size.

- **Opcodes for transactions**: These are the opcodes that contain details about the transaction payload used to call smart contract, for example, CALLVALUE stores the ether value sent along with the transaction, CALLDATASIZE returns the size of CALLDATA(transaction payload), etc.

- **Opcodes to push constant values on the stack** : These opcodes push constant values on the stack to process further. For example, PUSH opcode that pushes value on the stack. The PUSH opcode can push different bytes of data on the stack base on the prefix number attached to the PUSH. For example, PUSH1 pushes 1 byte of data onto the stack, PUSH8 pushes 8 bytes of data onto the stack and so on. The highest value that can be pushed on the stack is 32 bytes by PUSH32. There is no PUSH33 and above.

The recent 16 values on the top of the stack are accessible because of which we have opcodes for duplicate and swap stack values with prefix DUP1...DUP16 and SWAP1...SWAP16.
There are certain opcodes used for identifying symbolic variables in the bytecode. These opcodes are those which are to process data about the CALLDATA. CALLDATA is the transaction payload string that is sent as a transaction to call a smart contract. External users interact with the smart contracts on ethereum blockchain this way. A list of opcodes for processing the CALLDATA is listed below.

- **CALLVALUE** : CALLVALUE stores the ether value send by the calling contract. This is decided by the external user who is calling the contract and hence we treated this value as symbolic.

- **Calling External Contract from the Contract** : The return value of calls made to external contract from the contract are purely symbolic as we don't know the external smart contract. It is generally a boolean value for successful completion of the execution of external smart contract or the failure of execution of the external smart contract. The opcodes for calling external smart contracts are CALL, CALLCODE, DELEGATECALL, and STATICCALL.

- **CALLDATALOAD(x)** : The opcode CALLDATALOAD(x) is used for loading a specific part of the CALLDATA string starting from xth byte up to 32 bytes either to load external function signature in the payload string which the user wants to call in the smart contract or for the function arguments provides in the payload string to call the particular functions. As these are purely set by the caller the smart contract doesn't know what value is coming in the payload and hence we set all these as symbolic.

- **CALLDATACOPY(t, f, s)** : CALLDATACOPY(t, f, s) copies s bytes from calldata at position f to memory at position t. These bytes stored in memory are all stored as symbolic variables.

- **CALLDATASIZE** : CALLDATASIZE returns the size of the calldata in bytes is also treated as symbolic.

There are mainly two types of bytecode[34]:

- **Creation Bytecode** : When a smart contract is compiled creation bytecode is generated. It contains the bytecode for the complete code which was written in a high-level language, say solidity. The creation code is written such that on the successful execution of the code, all one-time setup, i.e, the constructor logic is executed and initializes the smart contract state accordingly and returns the bytecode of the contract without constructor because constructor logic was one time execution. The payload of the transaction that is used to create a smart contract contains the same bytecode.

  Creation bytecode retrieval from ethereum for a contract can be done by using

  type(ContractName).creationCode

- **Runtime Bytecode** : Runtime Bytecode is the code that will be presented to the user after the creation of the smart contract and initialization of the parameters. On the invocation of a smart contract, a part of the runtime bytecode is executed bases on the function which is called. This code does not contain constructor code because the constructor code is not relevant to the smart contract once it is deployed on the blockchain.
  Creation bytecode retrieval from ethereum for a contract can be done by using

  extcodecopy(a)

  And retrieval of the hash of the runtime bytecode can be done by using

  extcodehash(a)

2.5 Ethereum Virtual Machine(EVM)

Virtual machines make code run on different types of machines. The input to virtual machines is some bytecode which not a completely high-level code nor a machine-level code. Virtual machine understood this bytecode and based on the machine on which the bytecode is to be run, converts it to the low machine-level code. This helps in achieving portability. All nodes in the ethereum network agree on how should EVM behave. This helps in achieving consensus for the data and the code on the code as everyone will get the same data due to their agreement on EVM[31].
EVM is a stack-based machine with the maximum limit of the stack is 1024. EVM also keeps track of the gas usage to terminate the program when out of gas. In this thesis, we have implemented a Custom EVM which only executes bytecode instead of implementing fully functional EVM. Hence

knowledge of all storage components is sufficient. The Ethereum virtual machine specification lists three separate storage[28] as listed below:

- **Stack** : EVM is a stack-based machine. The stack is the area where all general-purpose computation happens. The stack size can go up to a maximum of 1024. If a contract reaches this stack size the EVM stops execution of the contract. The stack can hold up to a maximum of 32 bytes of the data item. The stack is the cheapest memory in terms of gas consumption.
- **Memory** : Memory is linear and holds a temporary variable because memory is temporary and erased between function calls. Memory can be seen as an additional assist to the stack computation for the execution of the current call. The maximum bytes of data that can be read at one time from a memory is 32 bytes. We can write from a range of 8bit to 256bits of data on the memory. To expand memory from a default size one needs to pay extra gas. Memory scales quadratically and the more it grows, the more it costs.
- **Storage** : Storage is used to store the smart contract's state. This is permanent between function calls. The gas consumption is the highest of all three memory in case of storage.

2.6 Control Flow Graph

Control flow graph is a graph representation of all the paths that a program might traverse during the execution of a program[1]. Each node in the control flow graph is called a basic block. The control flowing from one basic block to another basic block is shown using directed edges. The control flow graphs can be used to detect loops, detect various control structures, for example, if-then-else, etc in the program.

2.7 Denial of Service Attack

Denial of service attack makes resources unavailable. These attacks can be performed by increasing traffic on a host and because of which the waiting queue of the users requesting resources from the site/host increases and hence the response time of the host also increases. In the case of ethereum smart contract, dos can be performed without increasing the traffic although this is also an option to perform a denial of service attack. These could be falsely written smart contract to fail in case of invocation, block gas limit exceeds, etc.[5].

Chapter 3

DoS Vulnerabilities in Ethereum Smart Contracts

There are few DoS attacks/vulnerabilities found in the past which exploit smart contract vulnerabilities[19][20].

3.1 King of Ether Smart Contract[18]

In 2016 this contract was deployed. It was a game where players send ether to the contract to take the throne. The new player claims the throne whenever he/she send more ether. If 14 days passed and there is no new successor, the throne was reset and the game started all over again. The idea is every successor has to pay more ether than the previous successor to take the throne. Hence the value of the throne increases with more and more successors[35]. The contract's function to update the successor looks similar to the code below.

```
contract setThrone {
    address currentSuccessor;
    uint highestValue;

    function claimThrone() payable {
        require(msg.value > highestValue);
        require(currentSuccessor.send(highestValue)); // Refund the old Successor else revert
        currentSuccessor = msg.sender;
        highestValue = msg.value;
    }
}
```

The problem in the contract is with the statement

require(currentSuccessor.send(highestValue));

The *address.send()* has a gas limit of 2300 gas. While this is good to prevent reentrancy bug but this limit of gas will fail to send the ether to the previous successor's contract if the player's(previous successor's) fallback function consumes more than 2300 gas. Hence the player can claim the throne and then fails the send command purposely so that no new player can claim the throne even if the new player is eligible to claim the throne. This result is a denial of service of the contract by rejecting new thrones even if eligible.

3.2 Block Gas Limit

Each block has a limit on the amount of gas spent. If the amount of gas spend exceeds the limit the transaction will fail. Hence this transaction can never be added to the blockchain. This will lead to the denial of service since that contract will no longer work as this transaction will not be added into the blockchain. Unbounded loops are the cause of this vulnerability. If the attacker can manipulate the loop limit, he/she can manipulate the amount of gas spend within the block. There are two primary ways to make loops unbounded.

1. Let's consider the below smart contract

```
1       pragma solidity ^0.4.24;
2       contract BasicToken {
3           uint256[] balances;
4           function add(uint256 a) private  {
5               balances.push(a);
6           }
7           function balanceOf() public  view returns (uint256) {
8               uint c = 1;
9               add(i);
10              return c;
11          }
12          function print() public view returns (uint256){
13              uint256 temp = 0;
14              for(uint256 i = 0; i < balances.length; i++){
15                  temp += balances[i];
16              }
```

```
17                return temp;
18            }
19        }
```

The *balanceOf()* function in the above contract lets anyone add items to the *balances* array. Then in the *print()* function, the loop condition is on the length of the *balances*. An attacker can intentionally add garbage values to the *balances* array and then the *print()* function will exceed the block gas limit. Hence the transaction containing *print()* function call will always fail when the *balances* array length reaches a certain limit. Hence *print()* function will become unavailable.

2. Changing loop length directly

   The iteration of loops can be on a constant value, or a variable, or dynamic array, or mapping. All of these other than the constant value can be changed directly by assigning value to it by assigning a direct value. If such an assignment statement is controlled by an external user then it could lead to a vulnerable loop.

   For example the following statement in the above smart contract
   balances.length = ***arbitrary value***
   arbitrary value is given as argument to *balanceOf()* function.

It is to be noted that these conditions for making loops unbounded could be made accessible only to some allowed addresses. These addresses are decided by the owner of the smart contract for example if the above smart contract changes as below. Line 6-13 are added in the above smart contract. A variable *owner* of the type address is added. The value of the *owner* is set in the constructor. *modifier onlyowner()* checks whether the caller address is the same as the owner. If both the addresses match then the program proceeds further else revert. Now even if the *balanceOf()* function is called by anyone, the call from the owner address will proceed further. All other addresses(other than the owner) which try to take benefit of *balanceOf()* function will fail. Even though this will also result in denial of service for the *print()* function, this case should not be considered as a vulnerability because no outsider can exploit the denial of service vulnerability. Maybe the coder is aware of such behavior and if still, he wants to code this contract. Hence we cannot be sure of the intent of the coder and therefore this thesis will focus only on the cases which can be exploited by the attackers.

```
pragma solidity ^0.4.24;

contract BasicToken {
   uint256[] balances;

   address owner; // owner address
   constructor(address o) public { // constructor to set owner
       owner = o;
   }
   modifier onlyowner(){ // modifier to check if caller is owner else revert
       if(msg.sender != owner) throw;
       _;
   }

   function add(uint256 a) private  {
       balances.push(a);
   }
   function balanceOf() public  onlyowner view returns (uint256) {
       uint c = 1;
       add(i);
       return c;
   }
   function print() public view returns (uint256){
       uint256 temp = 0;
       for(uint256 i = 0; i < balances.length; i++){
           temp += balances[i];
       }
       return temp;
   }
}
```

3.3 CREATE2 Information Leak Vulnerability

An exploitable information leak/denial of service vulnerability exists in the libevm (Ethereum Virtual Machine) *CREATE2* opcode handler of CPP-Ethereum. The attacker can create smart

contract to trigger this vulnerability. Hence use of *CREATE2* vulnerability should be considered as a denial of service vulnerabilities.[41]

3.4 Existing Related Work on DoS

- **An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks[30]** proposes a novel gas cost mechanism, which dynamically adjusts the costs of EVM operations according to the number of executions, to thwart DoS attacks.
- **SmartCheck: Static Analysis of Ethereum Smart Contracts[50]** is a static analysis tool[23][49] which looks for conditional statements (if, for, while) containing condition as an external call. This tool also looks for an unbounded loop.
- **Slither: A Static Analysis Framework for Smart Contracts[33]** is another static analyzer tool[32] where external calls in loops are considered as one of the vulnerability.

Chapter 4

Symbolic Approach and DoS Patterns

4.1 A Symbolic Approach to find DoS Vulnerabilities in Ethereum Smart Contracts

Symbolic execution[24] is a means of analyzing a program based on the inputs that the program can take and paths that can be traversed because of different inputs. Suppose we have a function with arguments that can be called by an external user. These arguments will be of some types for example integer, string, boolean, etc. Every type has some range of sets of values that can be assigned to a variable. Hence the external user has a set of choices which he can use while calling the function. Hence instead of testing the program on some random fixed inputs, every argument is considered as a symbolic variable. The program execution will go on as normal and when the argument is used in some computation during the execution of the program the symbolic variable will be used. This will generate symbolic expression as the program execution proceeds. This symbolic expression/variable can be used to resolve the condition branch to decide logically whether a certain path from the current execution point is reachable or not. An example showing symbolic execution is shown in the example below.

```
1      pragma solidity ^0.4.24;
2      contract BasicToken {
3      function fun(uint256 s) public  view returns (uint256) {
4          uint c = s; /*/ c = sym
5          c = s + 1; /*/ c = sym + 1
```

```
6           for(uint i = 0; i < c; i++){ /*/ for(uint i = 0; i < sym + 1; i++){
7               // some code
8           }
9           if(c > 10){ /*/ if(sym + 1 > 10){
10              //some code
11          }
12          else{
13              //some code
14          }
15          return c;
16      }
17  }
```

In the above example, the variable s is symbolic because *fun is an external function in solidity and can be called by whoever wants to interact with the smart contract using the* fun *function. The function also has an argument that is to be supplied by the caller through the transaction payload that is used to call the function. The symbolic execution of the function will treat* a as a symbolic argument. On assignment of the symbolic variable on line 4, the local variable will be assigned a symbolic value, *sym. Line 5 of the codes do some processing with the symbolic variable and hence a symbolic expression is the result of the processing,* sym + 1. Now this symbolic expression if used for branch decision, then it will be feed to the SMT solver to symbolically decide which branch which path to traverse. There may be a possibility to traverse all the paths or one path according to the satisfiability of the symbolic expression. For example, the loop condition on line 6 is satisfiable for both the paths, loops can be traversed for the condition $i >= sym$ and loop can break for the condition $i < sym$. Similarly on Line 9 of the code if the condition is satisfied for $sym >= 10$ and not satisfied for $sym < 10$.

Hence to find DoS vulnerabilities, we will execute all paths of a smart contract which can be traversed symbolically and check for vulnerabilities patterns as discussed below during the execution of the smart contract.

4.2 Vulnerability Patterns

- **Call to address changed by an attacker**

  An attacker can change the address of the external call made from the victim contract. This ability may be the functionality of the contract as we have seen in chapter 3. But this can be used against the victim contract. The attacker can purposely and deliberately code his

fallback function in such a way that the call will never be executed successfully. Now there will be multiple scenarios

- **Exception check on the call in victim contract**

  The failed call will revert the execution of the victim contract and hence the particular function will throw an exception and the execution of the victim contract will stop every time the particular function containing such call is invoked. This will make every execution trace containing such calls revert and become unavailable for use.

- **No exception check on the call in the victim contract**

  The inbuilt functions *send() and call()* do not throw exceptions by themselves, instead, they must be handled manually. In case they are not handled then the **King of Ether** contract will execute normally and only the attacker will be in loss because he is removed from the throne and the ether he invested also lost due to the failed call. But these functions return values for example call return 1 on success and 0 on failure. There could be some computations based on this return value. Now even though there is no exception and nobody knows if there is anything wrong, the path of execution that is supposed to be executed on the success of the call will never be reached. Hence this can also lead to denial of service if the logic of the smart contract permits.

- **variable gas limit in *call()* function**

  The *send() and transfer()* function have gas limits and are easy to exploit for the DoS attack in the above scenario. But the *call()* function has the option to set the limit of gas to be sent to the called contract. But this will not pose any serious threat to the vulnerability as the attacker can try different gas limit fallback function to consume all the gas.

Hence if on traversing any execution trace we came across a *call()* function whose *address* argument is symbolic variable or symbolic expression, the tool flag the contract as vulnerable.

- **Loop with Symbolic Variable Condition**

  Ethereum has a block gas limit. If a block consumes more gas than the limit then the execution stops and the transaction reverts. Hence this path of execution will never be completed and hence the denial of service. If the conditional expression is a symbolic variable/expression it can take different values and can lead to block gas limit. Please note that loops with conditional expressions as fixed can also reach the block gas limit if the fixed value is large enough. But this block gas limit will be from the start of the deployment of code on ethereum and no malicious person is needed to invoke that vulnerability. Hence the loops with conditional expressions as symbolic variables/expressions are considered as vulnerable.

- **Call to address changed by an attacker in a loop**

  This is no different vulnerability. It is the same as *Call to Address Changed by Attacker* but inside the loop. There is no condition on the loop. It can be with the conditional expression as a fixed value or it can be with the conditional expression as a symbolic variable/expression.

- **CREATE2 Opcode**

  An exploitable information leak/denial of service vulnerability exists in the libevm (Ethereum Virtual Machine) *CREATE2* opcode handler of CPP-Ethereum. Hence if in the bytecode *CREATE2* opcode id found we considered it as vulnerable.

Chapter 5

Working of Tool

An ethereum smart contract after compilation generates bytecode. This bytecode is deployed on the ethereum blockchain. The source code is not stored on the blockchain. Hence there is no way to know the source code of a deployed smart contract. Etherscan[12] is a Block Explorer, Search, API, and Analytics Platform for Ethereum. Among other things they also provide the source code verification as a service. Anyone can use this service to attach a source code with any deployed smart contract. The tool builds the deployment payload and checks if the same payload is on the blockchain or not. If both matches then the source code is linked to the payload on etherscan[38]. Hence we can find the source code of many smart contracts on etherscan. Since there are millions of smart contracts deployed on blockchain not all smart contract's source code can be found on etherscan. But for the working of our tool, we only need bytecode of smart contract instead of source code. The bytecode of a smart contract is given as input to the Tool, it processes the bytecode and generates the output as shown in Figure 5.1.

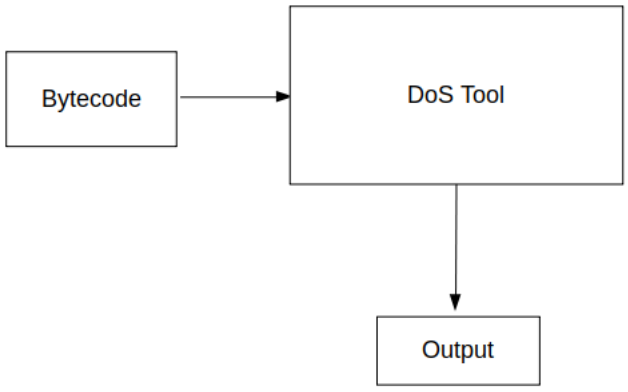


Figure 5.1: Workflow of the system

5.1 Insight of Components of Tool

We will now dive deep into DoS Tool working. The bytecode is feed to the ***parser*** which prepossess the bytecode. The ***parser*** converts bytecode into an opcode view named as parsed code in the tool. These opcodes and the control flow graph are feed to the ***Symbolic Execution Phase*** which is the Custom EVM executed symbolically. The Symbolic Execution Phase along with the output also updates a list of visited nodes. These visited nodes are the nodes that can be visited during the execution of Symbolic Execution. Hence these nodes are a subset of the nodes present in the Static CFG.

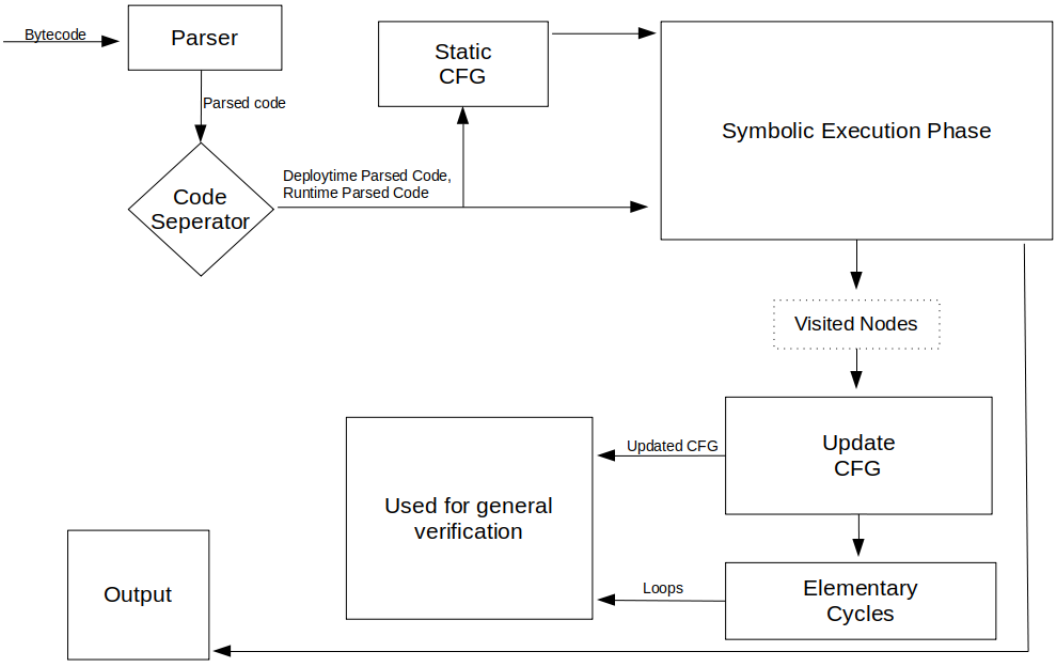


Figure 5.2: Components of Tool

The ***Symbolic Execution Phase*** also resolves new paths that were not able to be resolved statically. This path resolution will be discussed in detail. Using the visited node list we update the Static CFG by deleting the nodes from the CFG which are not present in the visited node list. The graph algorithms to find elementary cycles are used to find loops in the updated CFG.The *Updated CFG* is used to find the number of paths that can be traversed. The numbers in most cases should not match with the number of paths found in Static CFG because of the overestimation of paths. This will be discussed in the subsequent topics. A flow diagram of the Tool is shown in the Figure 5.2.

Let's discuss the Components of the Tool in detail.

5.1.1 Parser

Bytecode is a hexadecimal representation of the smart contract[42]. EVM(Ethereum Virtual Machine) has predefined opcodes. EVM understands these opcodes. The hexademicals in the bytecode corresponds to these opcodes. For example, **PUSH1** is an opcode that pushes 1 byte of the data on the stack. The corresponding hexadecimal value is **'0x60'**. Similarly, **MSTORE** is an opcode that stores data in memory. The corresponding opcode is **'ox52'** and many more. The size of every opcode is 1 byte. Data in the bytecode can have a maximum of 32 bytes size. PUSH is an opcode that pushes constant data onto the stack without any computation. Hence such data is given directly in the bytecode in the hexadecimal format right next to the PUSH opcode. For example, the string ***'6040'*** means ***60*** corresponds to the PUSH1 opcode where 1 means 1 byte, hence PUSH1 pushes 1 byte of data on the stack. The next one byte, i.e., ***40*** is the data that PUSH1 will push to the stack. We can find a list of EVM opcodes and their corresponding hexadecimal value along with some extra information, such as gas consumption of the opcode in the Ethereum Yellow Paper[11] and also here[10].
The parser takes as input the hexadecimal string(bytecode) and returns a list of these opcodes. The list that parser returns have 4 fields.

- ***step***: step corresponds to the position in the bytecode string. step is calculated in bytes. for example, let's consider the string '604052'. Here step = 0 means at 0th position in the string. step 1 corresponds to 2nd position in the string and so on. step can be used in the **JUMP/JUMPI** as an argument of position to jump to.

- ***operand***: operand is nothing but the hexadecimal value in the bytecode corresponding to the opcode. For example, in the string '604052', 60 is the operand for the opcode PUSH1.

- ***input***: input is the integer value of the corresponding hexadecimal data in the bytecode. For example, in the string '604052', the PUSH1(60) needs an input of 1 byte which will be right next to it, i.e., 40 in the string, and its integer value is 64.

- ***o***: o is the opcode corresponding to the hexadecimal value of the opcode in the bytecode. For example, in the string '604052', opcode corresponding to '60' is PUSH1, and opcode corresponding to '52' is MSTORE.

A sample parsed code of an opcode looks like below:

{'step' : 2, 'operand' : 60, 'input' : 64, 'o' : PUSH1}

step equals to 0 means the positions of this opcode is the 2nd byte in the bytecode, operand value at 2nd byte is 60, the input for the operand 60 is 64 and the opcode corresponding to the operand

60 is PUSH1. Hence parser converts bytecode into the custom opcode/assembly code list which we called a parsed code.

5.1.2 Code Separator

The bytecode is a combination of creation bytecode, we named it as deploytime bytecode, and runtime bytecode, we named it as runtime bytecode. The runtime bytecode is the code that is exposed to the external user. The deploytime bytecode generates the runtime bytecode. It is equivalent to the input data of the transaction which creates a smart contract[45]. We have separated deploytime parsed code and runtime parsed code from the parsed code beforehand to feed to the other components of the DoS Tool. A static pattern search is performed to separate these two codes from the bytecode. **CODECOPY** is an opcode which is used to copy constructor parameters and to copy the runtime bytecode to the memory of the EVM. Also, every bytecode, be it deploytime or runtime, starts with fixed instructions. These fixed instructions set a pointer to the free memory pointer from where the program starts storing data in memory. These instructions are as follows

PUSH1 mm

PUSH1 dd

MSTORE

Here mm and dd are the hexadecimal 1 byte data for the PUSH1 opcodes. MSTORE needs two inputs the memory location where we want to store, i.e., mm and the data to store at the memory location, i.e., dd. Hence every deploytime bytecode or runtime bytecode starts with a string '60mm60dd52'. The algorithm to separate bytecode is as follows.

```
1       Search for CODECOPY in the parsed code
2       if CODECOPY found
3
4           search for RETURN ahead
5           if RETURN found
6
7               search for the string ahead
8               if CODECOPY found
9                   goto line 2
10              if string found
11                  the parsed code above this position is deploytime bytecode
12                  the parsed code below from this position is runtime bytecode
```

```
13                  return
14              if string NOT found
15                  return can't separate
16
17          if CODECOPY found
18              goto line 2
19          if RETURN not found
20              return can't separate
21
22      if CODECOPY not found
23          return can't separate
```

5.1.3 Static Control Flow Graph

There is an opcode **JUMPDEST** in the EVM opcode list. The sole purpose of the opcode is to mark the beginning of a block. Every block starts with **JUMPDEST** except the very first block which starts with the magic string '60mm60dd52' as discussed under code separator component. Hence we can obtain a list of basic blocks in the parsed code. Now, these basic blocks need to be connected to determine the control flow of the program. There are four ways to determine the control flow of the program.

- **JUMP** opcode: This is an unconditional jump. JUMP only takes one argument which is the destination of the jump. If the destination opcode is JUMPDEST then it is a valid jump else EVM stop traversing this path. When EVM encounters JUMP instruction it takes the current top value of the stack as argument and jumps to this location in the bytecode. This location argument is either directly pushed before JUMP instruction by using PUSH opcode or is a result of some computation, for example - add, mul, etc. Since we are not executing Custom EVM, we don't have stack trace and we can only resolve the JUMP location only when its argument is directly pushed by PUSH instruction in the bytecode just before the JUMP instruction.

- **JUMPI** opcode: JUMPI is a conditional jump. JUMPI takes two arguments, the top of the stack is the jump location, and the next value on the stack is a boolean condition value. If condition value is 1 the code jumps to the location provided by jump location argument and if the condition value is 0 the code continues executing without the jump. Also if condition value is 1, the jump location in the bytecode is only valid if it is JUMPDEST opcode. JUMPI location can also remain unresolved like the JUMP location.

- **JUMPDEST** opcode: If on executing the current block, JUMPDEST instruction is encountered then this is the starting of the next block, and control is flowing directly from the current block to the next block. Therefore we simply connect the block with the next adjacent block.

- **INVALID**: If we encounter an INVALID opcode on traversing a block then it is the end of the block and this block is the end block in the path and not connected to any other block in the control flow of the program.

The output of this component is a control flow graph we named as STATIC CFG. The name static is added just to clarify that the control flow of the program is not obtained by executing the program and keeping track of the state trace of the program. There are two shortcomings to the CFG as discussed below.

- **Overestimation of Paths**

  To determine static CFG no dynamic or symbolic approach is used. Hence the control flow paths are overestimated because we have considered all paths. There may be a possibility that some paths will never be traversed for any inputs during the execution life of the program.

- **Unresolved JUMP/JUMPI**

  Even though the Static CFG is overestimated, the CFG may have missed the control flow that is possible to reach during the execution life of the program. This is mainly due to the reason for unresolved JUMP/JUMPI instructions as discussed above.

5.1.4 Symbolic Execution Phase

The Symbolic Execution Phase will run the bytecode in a Custom EVM symbolically. A diagram of the Symbolic Execution Phase is shown in Figure 5.3.

EVM works on the bytecode of the smart contract. EVM is a stack-based machine with the maximum limit of the stack is 1024. The EVM generally uses the stack for intermediate values in computations. EVM also uses memory and storage other than the stack. The memory is the real workhorse and is non-permanent. Storage is used for values that need to be persisted between different executions of the smart contract. Hence storage is permanent. The gas usage of storage is very high as compared to memory. Hence one should wisely decide between what to store permanently and what should be temporary during smart contract execution. EVM also keeps track of the gas usage to terminate the program when out of gas. The Custom EVM used in the Symbolic Analysis Phase implements the stack, memory, and storage, basically the state implementation. The Custom EVM doesn't implement other features of EVM. The idea for the implementation of custom EVM is taken from the tool Maian[40][39].

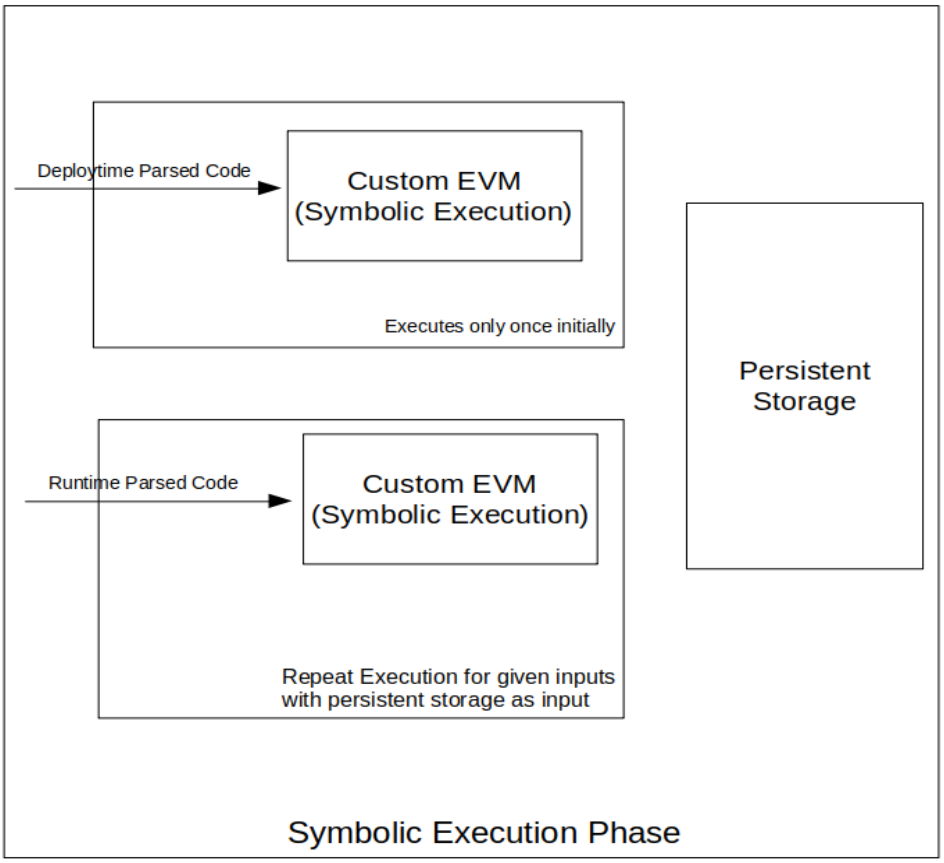


Figure 5.3: Symbolic Execution Phase

The Custom EVM starts executing the parsed code from the start. Any value which can be decided by an external user is considered as symbolic. For example, CALLDATA of functions which can be called to interact with the smart contracts, the return value from the external smart contracts which were called from the contract, CALLVALUE - the value of ether send to a payable function by the external user. Upon traversing the execution path, these symbolic variables build symbolic expressions. When the control flow branches we use SMT solver to decide whether the further path is feasible of not. If the SMT solver returns not satisfiable for a certain path condition then it will not be traversed. We have taken a few design choices to mitigate some of the problems discussed below.

- **Infinite Loops**: Infinite loop detection is undecidable. Therefore during the execution of the Custom EVM, there may be a possibility that the execution never stops. Hence we introduced maximum path length from the start of the execution. When the execution arrives at the new block the path length increases by one and if a certain path length reaches maximum path length limit the Custom EVM reverts and stops traversing the path. There is no way to decide what should be the appropriate value of maximum path length. Hence we tested the tool on three maximum path length, i.e, 10, 20, and 40. The maximum path length is

given as input to the tool by the user.

- **Multiple Invocation of smart contracts**: Let's understand this with an example.

```
pragma solidity ^0.4.24;
contract BasicToken {
    uint256[] balances;
    function balanceOf() public  view returns (uint256) {
        uint c = 1;
        balances.push(a);
        return c;
    }
    function print() public view returns (uint256){
        uint256 temp = 0;
        for(uint256 i = 0; i < balances.length; i++){
            temp += balances[i];
        }
    return temp;
    }
}
```

It is necessary to know how bytecode calls an external function. The first 4 bytes of the keccak of an external function known as function signature is the identifier of functions in the bytecode. For example, the function signature of *print()* function in the above code will be '13bdfacd' and function signature of *balanceOf()* function in the code will be '722713f7'. The assembly view of external functions in the smart contract above is

```
PUSH4 0x13BDFACD
EQ
PUSH2 0x51
JUMPI
DUP1
PUSH4 0x722713F7
EQ
PUSH2 0x7C
JUMPI
```

Hence the first four bytes of the CALLDATA will be compared with the function signature and jump to the corresponding function block. Since our Custom EVM starts executing

bytecode from the start, the *print()* function will be executed first and then the *balanceOf()* function. The loop condition in the *print()* function is symbolic because of the *balanceOf()* function. Hence balances.length will be set to symbolic when the Custom EVM executes *balanceOf()* function. Therefore in the first invocation of the smart contract, the vulnerability will not be captured. But since the balances.length is stored in the storage and we know storage is persistent, in the second invocation of the smart contract, even on executing the *print()* function first again, the vulnerability will be captured because the balances.length was set to symbolic in the previous invocation of the CUstom EVM. Hence the tool should test on multiple invocations of the smart contract to capture such vulnerabilities. We have tested on invocations 1 and 2 but this is also given as an argument to the Custom EVM.

- **Constant Loops with iterations more than MAXIMUM PATH LENGTH**

  In the constant loop, the loop condition is constant and hence SMT solver can do nothing much but will traverse the loop iteration until the loop completes. Only after completion of the loop, the path after the loop in the program will be traversed. Hence the branch of execution after a constant loop is only accessible after the completion of the loop. Hence if a constant loop is terminated in between the path after the loop will never be traversed. Although we have made this design choice by keeping in mind that all paths will not be traversed for the cost of not trapping into the infinite loop. But we can break this constant loop early and since we know that loop condition is constant we can traverse the path after the loop instead of looping the maximum path length into the constant loop. This should also be noticed that these constant loops can also exceed the block gas limit. So only constant loops for 10000 iterations are considered for our Custom EVM. EVM block gas limit in practice exceeds for loops around 2-4 lakhs of iterations. Again this can be set and not fixed in the Custom EVM.

The Static CFG is also an input to the Symbolic Execution Phase to resolve the unresolved control flow paths as discussed in the Static CFG. We have also maintained a list of visited nodes. Whenever a new block is reached at any point in the execution, on any invocation we add it to the visited list. Hence visited node is a list of blocks ever visited during the Symbolic Execution phase of the Tool.

5.1.5 Output

The symbolic Execution Phase outputs the vulnerability of the smart contract from the four vulnerabilities as discussed above and as safe if no vulnerability is found.

5.1.6 Update CFG and Elementary Loops

The updated CFG will be used for updating the visited nodes list by Symbolic Execution Phase as input. Update CFG will remove nodes and their paths from the Static CFG. Hence the number of paths in updated CFG are subsets of the path in the Static CFG. The number of paths corresponding to the Static CFG and Updated CFG is shown in the output of the Tool.
The elementary loops will find all the elementary loops in the updated CFG. It is to be noted that the loops found during the symbolic analysis phase are different from the loops obtained from the Updated CFG. After careful observation, we noticed that the loops obtained during Symbolic Execution should be a subset of the loops obtained from the Static CFG. There may be many reasons for this and we will see on reason with the help of the graph in Figure 5.4. Let the contract execution starts from node A and we will consider only 1 invocation of the smart contract. Suppose the control flow AC is not symbolically feasible during the execution of the smart contract. The execution can traverses path AB or ABC or ABCD or ABCDA. If the execution goes back to the A by traversing path ABCDA, then all the nodes will be in the visited nodes list. Hence the Static CFG and the Updated CFG will be the same because we considered all visited nodes without considering whether the path from one node to another is feasible or not.

Hence when we find loops using graph algorithm on the Updated CFG, we will get two loops

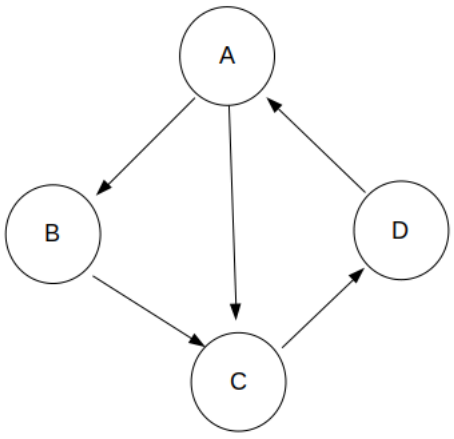


Figure 5.4: A Sample CFG

ABCDA and ACDA, while the Symbolic Analysis Phase returns only 1 loop, i.e., ABCDA. These

features are not necessary for the tools but we build these components to do some reasoning about the Tool.

Chapter 6

Results

The tool is set up on a 64-bit Ubuntu 20.04 LTS, 7.7 GB of RAM and 8 Core Intel(R) Core(TM) i5-8250U CPU @ 1.60GHZ. A total of 35 smart contracts are tested with 6 different sets of {Invocations, Path length}. We have tested the tool on 1 and 2 invocations with path length 10, 20, and 40 respectively for each of the mentioned invocations. The result of the tool for 1 invocation and sets of path length is shown in Figure 6.1.

A total of 6 smart contracts are flagged as vulnerable in the setup with 1 invocation and path

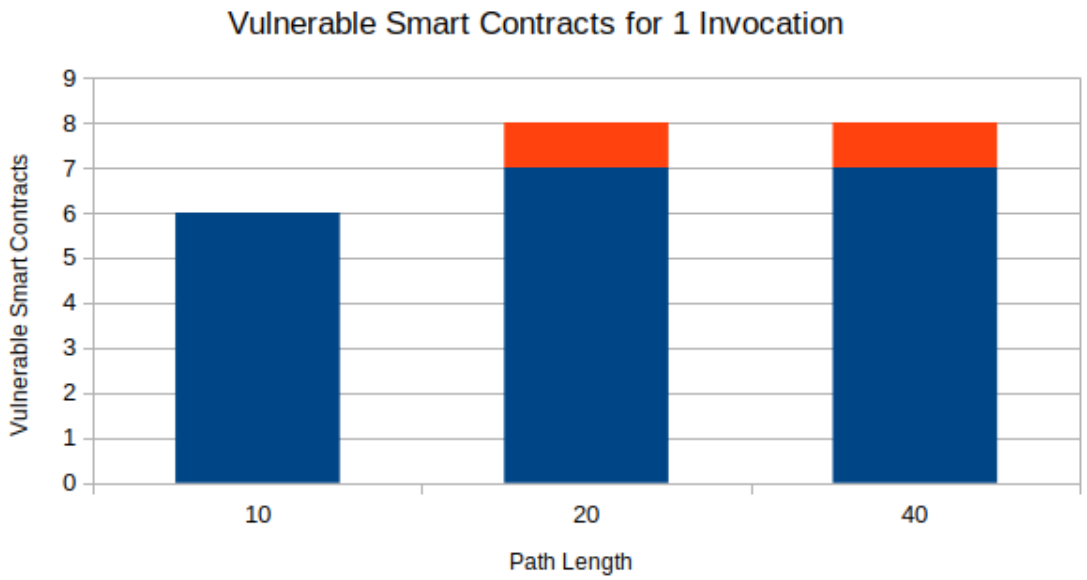


Figure 6.1: Vulnerable Smart Contracts for 1 Invocation

length equals to 10. A total of 7 smart contracts are flagged as vulnerable in the setup with 1 invocation and path length equals to 20. Out of those 7 smart contracts. 1 smart contract is a false positive. The result with 1 invocation and path length 40 path length is the same as the 1 invocation and 20 path length.

The result of the tool for 2 invocations is shown in Figure 6.2. A total of 9 smart contracts are

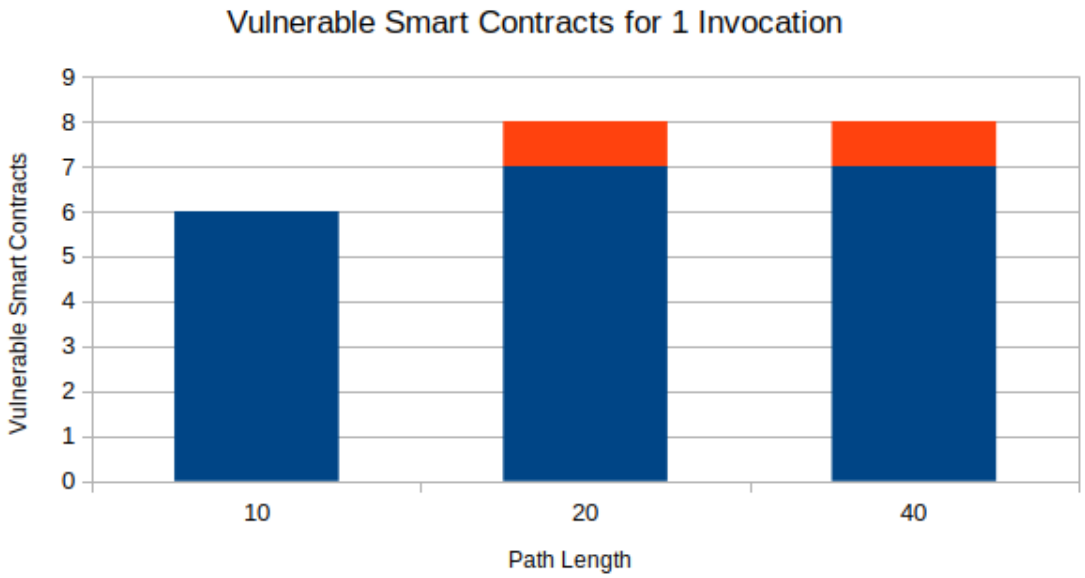


Figure 6.2: Vulnerable Smart Contracts for 2 Invocation

flagged as vulnerable in the setup with 2 invocations and path length equals to 10. A total of 11 smart contracts are flagged as vulnerable in the setup with 2 invocations and path length equals to 20. Out of those 11 smart contracts, 1 smart contract is a false positive. The result with 2 invocations and path length 40 path length is also the same as the 2 invocations and 20 path length.

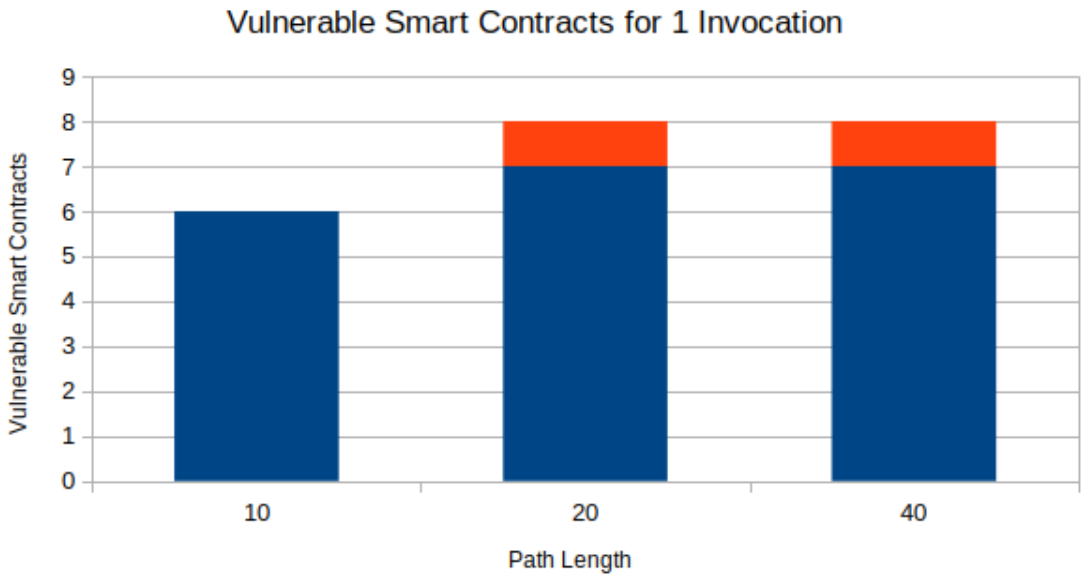


Figure 6.3: Comparison of Vulnerable Smart Contracts

The 2 invocations setup was able to find more vulnerable contract which otherwise was left by

the 1 invocation setup. This was expected as we have explained in Symbolic Execution Phase of the Dos Tool. Also with the increase in path length the tool was able to traverse path more deeply and explore more vulnerabilities if present. A comparison chart with different invocation setup and path length is shown in Figure 6.3.

About the False Positive Case

The false positive case is for the vulnerability of unbounded loop condition. We have assumed that the symbolic value can take any value. For example, consider this value as the integer value. Now the integer type variable has different ranges based on the type of integer in solidity. For example, solidity has signed and unsigned integer of different sizes. The range varies from 8 bit number to 256 bit number with keywords unint8 to unint256 and int8 to int256. An unsigned integer uint32 has a range from 0 to 2**32 - 1[17].

But SMT solver on symbolic expressions yield the same result of all different ranges of an integer value in the loop condition and we have given the range limit of 256-bit number for the SMT Solver. We need to know the different range of integers which with the current tool is not possible to determine and the SMT will overestimate paths based on our current setup. The smart contract which gives false-positive looks like below:

```
1       function nameFilter(string _input)
2           returns(bytes32)
3       {
4           uint256 _length = _temp.length;
5           require (_length <= 32 && _length > 0);
6           for (uint256 i = 0; i < _length; i++)
7           {
8               //some code
9           }
10      }
```

Although the *_input* length is symbolic since the string is symbolic. Its length is limited by the line number 5 of the code. But the SMT will return satisfiable for the Symbolic expression in the loop condition at line 6. These limitations based on integer size or checks as performed in the above code are falsely interpreted as vulnerable contracts by the tool.

Chapter 7

Summary and Future Work

Summary

In this thesis, we develop a symbolic analysis tool for checking whether a smart contract is vulnerable to denial of service vulnerabilities or not. We start by looking at the denial of service vulnerabilities in smart contracts discovered already in the past. We deduced some patterns because of which those vulnerabilities were possible. We then go on to explain the tool and various components in the tool. We explained some designed choices for the successful execution of the tool. We then discuss the results and some details about the shortcoming of the tool, where the tool will fail.

Future Work

- We can build a type checking[25] module that works alongside the code to help SMT solver to decide what range to check instead of checking fixed range values.
- One can think of extending the tool for more patterns of Dos attacks which can be found in the future.
- The tool is detecting whether a smart contract is vulnerable and that vulnerability arises from which external function using the function signature. But we have only used the function signature which is keccak256 of the functions in solidity. We can also join bytecode to source code reverse engineering[52] tool[36] to exactly determine the functionality of the function that is causing the vulnerability.
- This tool is specifically made for checking vulnerabilities in ethereum smart contracts. A similar approach can be followed for other smart contracts available.

Bibliography

[1] Control-flow graph. Available at https://en.wikipedia.org/wiki/Control-flow_graph.

[2] Cyrtokitties. Available at http://ethertweet.net/.

[3] The dao (organization). Available at https://en.wikipedia.org/wiki/The_DAO_(organization).

[4] Decentralized application security project (or dasp) top 10 of 2018. Available at https://dasp.co/.

[5] Denial-of-service attack. Available at https://en.wikipedia.org/wiki/Denial-of-service_attack.

[6] Developer resources. Available at https://ethereum.org/developers/.

[7] Ethereum. Available at https://en.wikipedia.org/wiki/Ethereum#Ether.

[8] Ethereum. Available at https://en.wikipedia.org/wiki/Ethereum.

[9] Ethereum daily transactions chart. Available at https://etherscan.io/chart/tx.

[10] Ethereum vm (evm) opcodes and instruction reference. Available at https://github.com/crytic/evm-opcodes#ethereum-vm-evm-opcodes-and-instruction-reference.

[11] Ethereum yellow paper. Available at https://github.com/ethereum/yellowpaper#ethereum-yellow-paper.

[12] Etherscan. Available at https://etherscan.io/.

[13] Ethertweet. Available at http://ethertweet.net/.

[14] Ethlance. Available at https://ethlance.com/.

[15] Explore decentralized applications. Available at https://www.stateofthedapps.com/.

[16] Gnosis. Available at https://blog.gnosis.pm/.

[17] Integer. Available at https://solidity.readthedocs.io/en/v0.5.3/types.html#integers.

[18] King of the ether. Available at https://www.kingoftheether.com/thrones/kingoftheether/index.html.

[19] Known attacks. Available at https://consensys.github.io/smart-contract-best-practices/known_attacks/.

[20] Known attacks. Available at https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/known_attacks/.

[21] Koet. Available at http://blockchain.unica.it/projects/ethereum-survey/attacks.html#kotet.

[22] Smart contract. Available at https://en.wikipedia.org/wiki/Smart_contract#History.

[23] Smartcheck. Available at https://tool.smartdec.net/.

[24] Symbolic execution. Available at https://en.wikipedia.org/wiki/Symbolic_execution.

[25] Type checking and type equality. Available at https://people.cs.umass.edu/~creichen/csci3155-s07/handout-B.pdf.

[26] Understanding bytecode on ethereum. Available at https://medium.com/authereum/bytecode-and-init-code-and-runtime-code-oh-my-7bcd89065904.

[27] The world's most accessible, no-limit betting exchange. Available at http://www.augur.net/.

[28] Ethereum virtual machine: Why and how you should use it, January 2016. Available at https://www.bitdegree.org/learn/ethereum-virtual-machine.

[29] Vitalik Buterin. Thinking about smart contract security, June 2016. Available at https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[30] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against underpriced dos attacks. In Joseph K. Liu and Pierangela Samarati, editors, *Information Security Practice and Experience*, pages 3–24, Cham, 2017. Springer International Publishing.

[31] eth. What is the ethereum virtual machine?, September 2019. Available at https://ethereum.stackexchange.com/questions/27367/what-is-the-ethereum-virtual-machine.

[32] J. Feist, G. Grieco, and A. Groce. Slither, the solidity source analyzer. Available at https://github.com/crytic/slither.

[33] 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.

[34] Shane Fontaine. What are smart contracts? guide for beginners, September 2019. Available at https://cointelegraph.com/ethereum-for-beginners/what-are-smart-contracts-guide-for-beginners.

[35] Pete Humiston. Smart contract attacks [part 2] - ponzi games gone wrong. Available at https://hackernoon.com/smart-contract-attacks-part-2-ponzi-games-gone-wrong-d5a8b1a98dd8.

[36] Deepak Kumar and James Levy. Erays. Available at https://github.com/teamnsrg/erays#erays.

[37] Joshua Mapperson. Ethereum smart contracts up 75% to almost 2m in march. Available at https://cointelegraph.com/news/ethereum-smart-contracts-up-75-to-almost-2m-in-march.

[38] Steve Marx. Verifying contract source code, January 2018. Available at https://programtheblockchain.com/posts/2018/01/16/verifying-contract-source-code/.

[39] Ivica Nikolic. Maian. Available at https://github.com/ivicanikolicsg/MAIAN#maian.

[40] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Marcin 'Icewall' Noga. Cpp-ethereum libevm create2 information leak vulnerability. Available at https://talosintelligence.com/vulnerability_reports/TALOS-2017-0503.

[42] Bernard Peh. Solidity bytecode and opcode basics, September 2017. Available at https://medium.com/@blockchain101/solidity-bytecode-and-opcode-basics-672e9b1a88c2.

[43] David Petersson. How smart contracts started and where they are heading. Available at https://www.forbes.com/sites/davidpetersson/2018/10/24/how-smart-contracts-started-and-where-they-are-heading/#2f37c82337b6.

[44] David Schatsky. Getting smart about smart contracts. Available at https://www2.deloitte.com/us/en/pages/finance/articles/cfo-insights-getting-smart-contracts.html.

[45] shane. What is the difference between bytecode, init code, deployed bytedcode, creation bytecode, and runtime bytecode?, September 2019. Available at https://ethereum.stackexchange.com/questions/76334/what-is-the-difference-between-bytecode-init-code-deployed-bytedcode-creation.

[46] Michael Solomon. Top 10 ethereum uses. Available at https://www.dummies.com/personal-finance/top-10-ethereum-uses/.

[47] Parity Technologies. The multi-sig hack: A postmortem, July 2017. Available at https://www.parity.io/the-multi-sig-hack-a-postmortem/.

[48] Parity Technologies. A postmortem on the parity multi-sig library self-destruct, November 2017. Available at http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[49] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck. Available at https://github.com/smartdec/smartcheck/.

[50] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, May 2018.

[51] u/ethererik. Governmental's 1100 eth jackpot payout is stuck because it uses too much gas, 2016. Available at https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.

[52] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering ethereum's opaque smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1371–1385, Baltimore, MD, August 2018. USENIX Association.