# A Reduction from Smart Contract Verification to Model Checking

by

Alireza Lotfi Takami

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Alireza Lotfi Takami 2021

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

We present a reduction from verification of smart contracts to model checking. A smart contract is a computer program written in a language with constructs that correspond to real-world contracts, such as verified sending and accepting digital cash. Model checking is an approach to verification of state-transition systems in which a state is the valuation of a set of variables. A reduction, in our context, is a polynomial-time computable function which guarantees that an input smart contract possesses a property if and only if the output instance of model checking possesses a property to which the former property is mapped. Our focus is smart contracts written to run on the Ethereum blockchain in a language compiled to Ethereum Virtual Machine (EVM) code. Our work is motivated by the importance of checking smart contracts for properties of interest and also by the observation in recent empirical work that establishes that existing verification tools are deficient. Our approach has some distinguishing characteristics from prior approaches, which we discuss in this thesis. We have implemented and carried out a limited empirical assessment of our reduction. We used a dataset of 69 curated smart contracts that contains 115 instances of security vulnerabilities from 10 different classes of such vulnerabilities. Our empirical work suggests that our approach can scale to real-world smart contracts.

**Acknowledgements**

I would like to thank my supervisor, Prof. Mahesh Tripunitara, for his supervision and support that made this thesis possible. I want to thank Prof. Arie Gurfinkel and Prof. Derek Rayside for their precious comments as the readers of my thesis.

## Dedication

To my parents Farangis and Abdolreze, and my dear love Sahar.

# Table of Contents


**List of Figures** ix

**List of Tables** x

**1 Introduction** 1

**2 Background and Prior Work** 4

2.1 Ethereum Blockchain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1.1 Currencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.2 Ethereum Applications . . . . . . . . . . . . . . . . . . . . . . 5

2.1.3 Development of Smart Contracts . . . . . . . . . . . . . . . . . 6

2.1.4 Ethereum Virtual Machine (EVM) . . . . . . . . . . . . . . . . 7

2.1.5 Ethereum Tools and Technologies . . . . . . . . . . . . . . . . 12

2.2 Model Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.2.1 Kripke Structure . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.2 Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.2.3 SAT-Based Model Checking . . . . . . . . . . . . . . . . . . . . 16

2.2.4 NuSMV . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.2.5 nuXmv . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.3 Related Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**3 Smart Contracts Classification** **23**

3.1 Some Classification Methods . . . . . . . . . . . . . . . . . . . . . . . . . . 24

3.2 Implemented Classification Methods . . . . . . . . . . . . . . . . . . . . . . . 27

3.2.1 Bytecode Size classification . . . . . . . . . . . . . . . . . . . . . . . 28

3.2.2 Function Signature based Method . . . . . . . . . . . . . . . . . . . . 32

3.2.3 Combined bytecode size and signature method . . . . . . . . . . . . . 33

**4 Smart Contracts to SMV** **35**

4.1 Halting Problem for Smart Contracts . . . . . . . . . . . . . . . . . . . . . . 35

4.2 EVM Architecture in SMV . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

4.2.1 Main module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

4.2.2 Processor module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

4.2.3 Gas module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

4.2.4 Memory module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.2.5 Storage module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.3 Model Checker Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.3.1 Python Code Architecture . . . . . . . . . . . . . . . . . . . . . . . . 39

4.4 EVM Operations in SMV . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.4.1 Supported Operations by SMV . . . . . . . . . . . . . . . . . . . . . 42

4.4.2 Not Supported Operations by SMV . . . . . . . . . . . . . . . . . . . 44

**5 Validation** **58**

**6 Conclusion and Future Work** **66**

**References** **67**

**APPENDICES** **71**

**A Ethereum Blockchain Small Size Contracts Evaluation** **72**

# List of Figures


2.1 Structure of a sample unreal smart contract developed by Solidity . . . . . . 7

2.2 Two types of account in the Ethereum blockchain . . . . . . . . . . . . . . . . 8

2.3 Two types of transactions in the Ethereum blockchain [44] . . . . . . . . . . 9

2.4 Stack-based architecture and the execution model of EVM [44] . . . . . . . . 12

2.5 The structure of a basic model checker . . . . . . . . . . . . . . . . . . . . . 14

2.6 The Kripke structure of an Australian traffic light . . . . . . . . . . . . . . . 15

2.7 A sample code of NuSMV input language . . . . . . . . . . . . . . . . . . . . 18

3.1 Classification of the verified smart contracts using the signature-based method 32

3.2 ERC20 standard tokens in each specified range according to smart contracts bytecode length . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.1 The structure of the designed smart contracts model checker . . . . . . . . . 38

4.2 The structure of the designed Python Code . . . . . . . . . . . . . . . . . . . 39

5.1 The counter example generated by nuXmv as result of testing MUL operation of EVM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

# List of Tables


2.1 Ether's Denominations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 The cost of the execution of some of operations in the Ethereum blockchain . 11

3.1 Number of distinct contracts with a bytecode length between 0 and 2000 . . 29

3.2 Number of distinct contracts with bytecode length between 0 and 50000 . . . 30

3.3 Number of verified contracts in Etherscan for 6 different ranges. . . . . . . . 31

4.1 A reference table to details of the implementation of each EVM operation or EVM operation group . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

5.1 Validation of the reduction of some of the EVM operations. . . . . . . . . . . 60

5.2 Results of the validation of the model checker using three LTL specifications 64

# Chapter 1

# Introduction

Since 2008 that Bitcoin introduced, cryptocurrencies have attracted the attention of many researchers and investors. The core technology of most cryptocurrencies is Blockchain. The technology made it possible for financial transactions to occur without the need for a trusted third party. In other words, a blockchain is an append-only list of blocks that are created and maintained in a network by all nodes of the network. Each Blockchain has a consensus method that a new block could be added to it based on the method. In the beginning years of this technology's emergence, it was only exciting for the technology geeks. However, by the day of writing this thesis, the market capitalization of Bitcoin is 777.91B that indicates the importance of cryptocurrency and the technology [8]. Although Bitcoin is still the most prevalent cryptocurrency, Ethereum, with a total market capitalization of 316.5B, is the second most popular cryptocurrency globally. What distinguishes Ethereum from Bitcoin is that although Ethereum is a currency like Bitcoin, it has made it possible for its users to create and execute Smart Contracts.

As in the Ethereum blockchain, both users and contracts can create transactions. Ethereum assigns a distinct 20-byte address to each of the users and accounts. By the date of writing this thesis, there are more than 99M distinct addresses on the Ethereum blockchain [10]. As mentioned, the address is assigned to users and contracts; therefore, there are two types of accounts in the Ethereum blockchain. Contract accounts and externally own accounts. Both types of accounts have a balance and can generate transactions, but the contract accounts are managed by a code that exists in them, but users control the externally own accounts. Although the externally owned accounts do not have any code, they can trigger the execution of the code of a contract account by creating and sending them a message or transaction.

The code of the contract accounts that is developed and directly deployed into the blockchain is called a smart contract. The smart contracts developer defines some functions in it that

based on the input message to the contract, this function could be triggered and change the state of the contract or transfer fund by creating a transaction. Solidity [16] is the famous high-level programming language that is used for the development of smart contracts. However, the smart contract could be developed with any other programming language as long as it could be compiled into a correct EVM code. The developed contract with any programming languages must be compiled into the Ethereum Virtual Machine(EVM) code and then uploaded by users into the blockchain.

As only the EVM code of a contract exists on the Ethereum blockchain, there must be a way for the contract users to make sure that the deployed contract is exactly the contract that the contract owner claims. One way is that the users request the owner or developer of the contract to provide the contract's flattened solidity code. By flattened, we mean the owner must include the libraries' code used in the contract. Then, the users can compile the solidity code and obtain the EVM code. As they have the smart contract address on the Ethereum blockchain, they can access the deployed EVM code on the blockchain and compare it with the EVM code that have been created by compiling the provided solidity code. Etherscan [20] is doing this verification for contract users using the same approach. The contract's owner uploads the flattened solidity code with some extra information on Etherscan, and Etherscan confirms it by comparing the EVM code with the uploaded EVM code on the Ethereum blockchain.

Although trusting the contract owner is essential, the security and correctness of the contract itself is a critical issue. Because like the other programs, smart contracts may also suffer from security vulnerabilities. Like the safety-critical embedded systems whose inaccuracy and security vulnerabilities would have catastrophic consequences, smart contracts' security, inaccuracy, and vulnerabilities could lead to catastrophic financial consequences. For example, about 150M USD has been stolen from the DAO contract using the reentrancy vulnerability [19]. Moreover, 30M USD was stolen using a vulnerability in the Parity multi-signature wallet [9]. Furthermore, attackers could also use the smart contract's vulnerabilities to make some adverse effects on the blockchain without stealing the money. For instance, a bug in the Parity multi-signature wallet contract made a considerable amount of Ether out of access of the contract users. Hence it is necessary to have precise security analysis on the smart contracts before deploying them on the Ethereum blockchain.

In this work, we provide a reduction from smart contracts verification to model checking. We implemented the proposed design using Python programming language and tested our implementation of each of the EVM operations using some hypothetical smart contracts EVM code. We used a curated dataset of vulnerable smart contracts to validate our implementation.

The rest of this thesis is organized as follows. In Chapter 2 we provide the necessary background on the Ethereum blockchain and Model Checking. Moreover, we review some of the prior works in the safety and security of smart contracts. Chapter 3 is an analysis of available smart contracts on the Ethereum blockchain. In this chapter, we review some of the smart contracts classification techniques and apply these methods to the available smart contracts on the blockchain. Chapter 4 provides the overall design of our reduction and details of the implementation of each of the EVM operations in SMV language. In Chapter 5 we describe our validation method and the dataset we used to evaluate our tool. Chapter 6 presents the potential future works and concludes the thesis.

# Chapter 2

# Background and Prior Work

This chapter provides a quick review of the Ethereum blockchain and the Ethereum Virtual Machine and its architecture. Moreover, we will present some of the standard tools for interacting with the blockchain. Then we will provide the basic knowledge on model checking and introduce the model checker that we will use in this work.

## 2.1 Ethereum Blockchain

The first and most popular cryptocurrency was introduced by an anonymous person in 2008. Since then, many people have been attracted to cryptocurrencies and specifically Blockchain, which is its core technology. A Blockchain is an immutable append-only list of data blocks that are connected using hash functions [2]. Bitcoin and most of the other cryptocurrencies have been designed and implemented based on the technology. In these cryptocurrencies, each block includes a few transactions and is added to the end of the current chain of blocks. Therefore, the Blockchain has an append-only structure that only new transactions could be added into it by appending a new block. Whenever a block is appended to the chain, it cannot be updated or deleted from it unless someone owns more than 50% of the computing power of the peer-to-peer network of the Blockchain. The distributed nature of the Blockchain comes from a consensus mechanism that is feasible by using cryptographic hash function and asymmetric digital signature. Using this feature of Blockchain, transactions could occur in a quick and cost-efficient manner.

Although Bitcoin is the most popular currency using blockchain technology, blockchain is the core technology of many other currencies, and it could have many other applications. Ethereum is the second most popular cryptocurrency that is based on blockchain. As the base technology of Ethereum is the same as bitcoin, it also has a distributed nature in which

| Unit | Wei Value | Description |
|---|---|---|
| wei | 1 wei | Base denomination |
| Kwei | $1 \times 10^3$ wei | |
| Mwei | $1 \times 10^6$ wei | |
| Gwei | $1 \times 10^9$ wei | Most common unit |
| Twei | $1 \times 10^{12}$ wei | |
| Pwei | $1 \times 10^{15}$ wei | |
| Ether | $1 \times 10^{18}$ wei | Standard denomination |

Table 2.1: Ether's Denominations.

there is a distributed network of miners that tries to append a new block of transactions into the blockchain. Whenever a miner mines a new block of transactions, it sends it to all other miners in the Ethereum network and based on a consensus method, they decide if to add the mined block to the network or not. If the newly mined block is verified by the consensus method, then it could be added to the blockchain, and the miner of the block is awarded a specific amount of money.

### 2.1.1 Currencies

Just like all other cryptocurrencies, Ethereum also has its standard currency. It is called Ether. Ether could be sent or received using the Ethereum blockchain. It can be sent/received directly between different accounts by the transactions or by executing the smart contracts in the blockchain. Ether also has smaller denominations that can be seen in Table 2.1. While the base unit of Ether is wei, Gwei or nanoether is the most commen unit that is used in the Ethereum network.

### 2.1.2 Ethereum Applications

One of the significant differences between Bitcoin and Ethereum is that Ethereum provides the ability to create smart contracts on top of it. This feature of Ethereum makes it possible to generate new currencies on top of it without creating everything needed to introduce a new coin from scratch. As functions of Smart contracts could be called externally by other contracts or by externally owned Ethereum accounts, by developing a set of smart contracts that can communicate with each other, Ethereum users can create Distributed Applications (DApps). In General, Ethereum could be used for pure financial applications, semi-financial applications or non-financial applications. Some of the applications that are provided using the Ethereum blockchain are as follows:

- **Token Systems:** Tokens are digital assets that have been created using the Ethereum

blockchain. As implementing token systems in the Ethereum blockchain is effortless many token systems such as ERC20 and ERC223 have been deployed on the blockchain.

- **Games:** Ethereum blockchain and its currency could be used in games by providing them with the ability of trading for assets in the game.
- **Auctions:** Ethereum blockchain could be used in Auctions by sending and receiving Ether.

Moreover, the Ethereum blockchain could also be used for Savings wallets, Cloud computing, Access control, etc.

### 2.1.3 Development of Smart Contracts

We are aware of two high-level programming languages for development of smart contracts that both of them target EVM code: Vyper[48] and Solidty[16].

#### Solidity

Solidity is a high-level contract-oriented programming language whose syntax is like the Java programming language. Solidity is the most common programming language for smart contract development. Every solidity file starts with a pragma version that indicates the correct version of the solidity compiler that needs to be used for compiling the smart contract. Figure 2.1 indicated an unreal simple smart contract written in solidity. A developed smart contract in solidity could have State variables, Functions, Function Modifiers, Events, etc.

#### Vyper

Vyper is a contract-oriented high-level programming language that is used to develop smart contracts. The syntax of Vyper is similar to the Python programming language. The developed contract by this programming language is compiled to EVM code and then is deployed to the Ethereum blockchain. In Vyper, each file could only contain one contract. Like Solidity, each developed contact by Vyper has a pragma version line that specified the correct version of the Vyper compiler that needs to be used to compile this contract. It also includes State variables, Functions and Events.

```
pragma solidity >=0.4.22 <0.6.0;

contract ModifierFunction1 {
    uint a = 0;
    uint b = 0;

    constructor() public {
    }

    modifier modifierFunction{
        require(a < 7, "simple require statement");
        _;
    }

    function test() public modifierFunction{
        if(a > 5){
            a=a+1;
        }else{
            a=a+1;
        }
    }
}
```

Figure 2.1: Structure of a sample unreal smart contract developed by Solidity

### 2.1.4 Ethereum Virtual Machine (EVM)

Like the Java programming language that uses Java Virtual Machine (JVM) as its runtime environment, Ethereum also has its runtime environment called Ethereum Virtual Machine (EVM). EVM is activated whenever a Smart contract on the Ethereum blockchain needs to be executed by receiving a message or transaction. Execution of a contract in EVM does not need any external resources from the network or file system, but there may exist some external calls to the functions of the other contracts in the blockchain.

#### Accounts

The state of the Ethereum blockchain is specified by its accounts. Each account is an object in the Ethereum network that has its own unique 20-bytes address. As it can be seen in Figure 2.2 there are two types of accounts on the blockchain: Contract accounts and Externally Owned Accounts (EOA). Externally owned accounts are the accounts that are controlled by a human using a pair of Public-Private keys. Hence, the address of the externally owned accounts indicate the user's public key in the network. Contract accounts are accounts managed by the EVM code which is a smart contract. Based on the input message to the contract account, the EVM code can change the Ethereum state or generate a transaction. Each account has a nonce to make sure each transaction is processed exactly once. There is also a storage field in contract accounts that is empty by default. Both types of accounts have a field to keep their balance in Ether which is a denomination of the Ethereum currency.

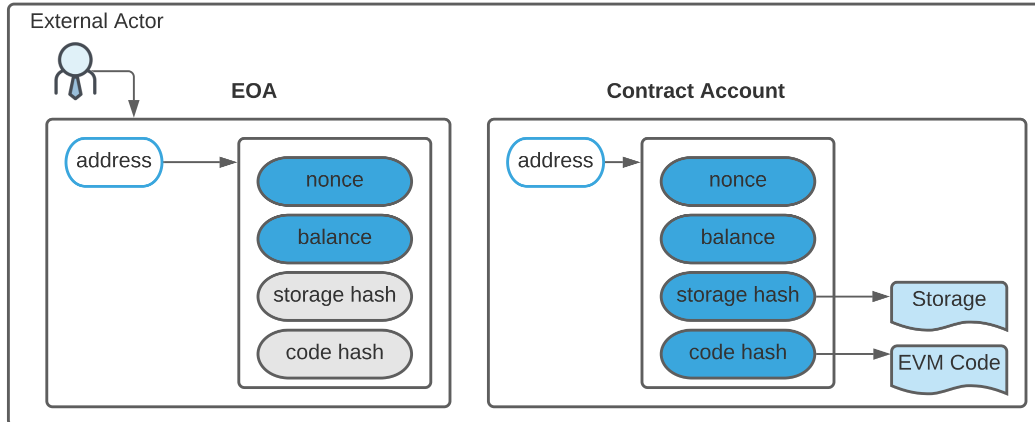


Figure 2.2: Two types of account in the Ethereum blockchain

### Transactions

Transactions are cryptographically-signed instruction that are created by an external actor, that could be human or a software, and are sent from one Ethereum account to another one. In other words, as can be seen in Figure 2.3 a transaction is a message associated with Ether and payload. If the recipient of the transaction is a contract account then the transaction triggers the execution of the EVM code of the target account and the payload will be used as input data of the contract. If the recipient of the transaction is null then the transaction leads to creation of a new contract account in the Ethereum blockchain. In this scenario the payload of the transaction is considered as an EVM code and is executed by Ethereum virtual machine. The output of this execution will be a new EVM code that is deployed permanently into the blockchain as a new contract account. The common fields of transactions based on the yellow paper are as follows [49]:

- **nonce:** A scalar value equal to the number of transactions sent by the sender.
- **gasPrice:** A scalar value equal to the number of Wei to be paid per unit of gas for all computation costs incurred as a result of the execution of this transaction.
- **gasLimit:** A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later.
- **to:** The 160-bit address of the message call's recipient.
- **value:** A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account.

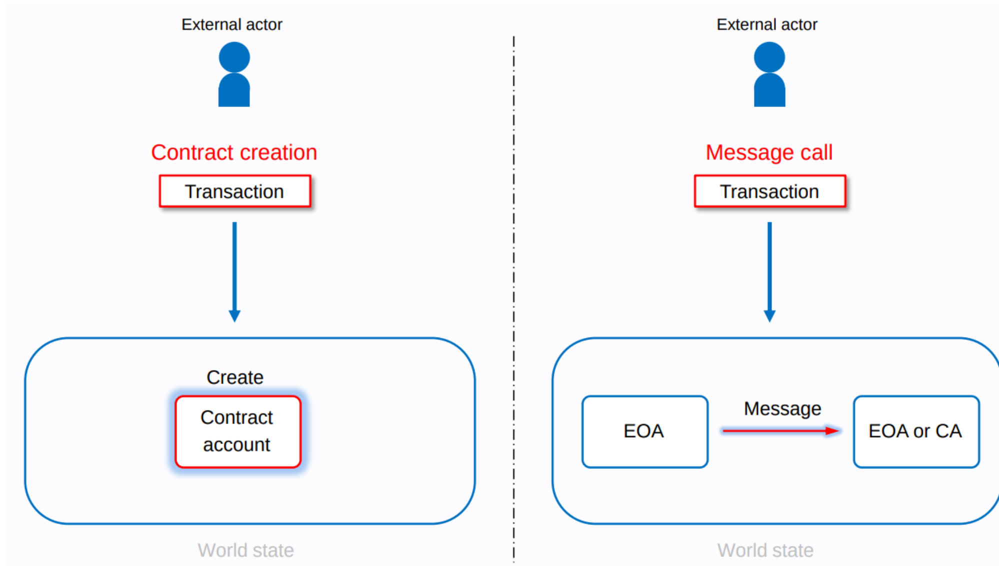


Figure 2.3: Two types of transactions in the Ethereum blockchain [44]

- **v, r, s:** Values corresponding to the signature of the transaction and used to determine the sender of the transaction.

Moreover, if the transaction is a contract creation transaction, there exists another field in it that is **init**. it is a byte array with unlimited size that include an EVM code which is responsible for the account initialization process. If the contract type is a message call, it will have a **data** field that is an unlimited byte array. **data** is the input data of the message call.

### Blocks

Each block in the Ethereum blockchain consists of three main parts: Header, a set of Transactions, and some information from the header of other blocks related to this block by their parent. Some data fields that exist in the block header are as follows [49].

- **beneficiary:** This is the address of the account that the collected fees of mining of this block should be transferred.
- **difficulty:** This is the difficulty level of this block that could be calculated using the difficulty level and timestamp of the previous block.
- **number:** This field indicates the number of ancestors of this block. This number for the genesis block is zero.

- **gasLimit:** This field indicates the current gas limit on this block. This value specifies a limit for the amount of gas that could be used in the current block. The miners in the blockchain determine this value.
- **gasUsed:** This is the amount of gas used by block's transactions.

Moreover, there are some other pieces of information in the blockchain. The detail of these fields is found in the Ethereum yellow paper [49].

#### Gas

In the Ethereum blockchain, whenever an account generates a transaction, it must specify the amount of gas that it is willing to pay to execute the generated transaction. Execution of each EVM operation costs a specific amount of fee specified in a gas unit. Table 2.2 indicates the amount of fee for the execution of some of the EVM operations in the Ethereum blockchain. More information on the EVM operation costs could be found in the Ethereum yellow paper [49].

The amount of gas that is associated with each transaction is called **gasLimit**. The other parameter that is specified with the transaction is **gasPrice**. The generator of the transaction has to pay $gasLimit \times gasPrice$ upfront, and if the processing of the transaction costs less than this value, the unused gas will be refunded to the creator of the transaction. The amount of gas that the execution of the transaction has used is sent to the account of the miner of the transaction's block.

There is no constraint on gasPrice, and the generator of a transaction could set it to an arbitrary value. However, miners could decide to put a higher priority on the transactions that have a higher gasPrice. Therefore, Transactor could guarantee the fast execution of its transaction by setting a high gasPrice with it. If the Transactor does not provide enough gas for the execution of the transaction, it may run out of gas in the middle of the execution. In that case, the execution of the transaction will be stopped, and the transaction will not have any effect on the Ethereum blockchain state. However, the miner will receive the spent gas on the partial execution of the transaction.

#### EVM Architecture

Ethereum Virtual Machine, a Turing-complete virtual machine, is responsible for executing the EVM code. In this section, we will describe EVM architecture and its execution model.

| Name | Value | Description |
|---|---|---|
| STOP | 0 | Nothing needs to be paid for this operation |
| RETURN | 0 | Nothing needs to be paid for this operation |
| ADDRESS | 2 | two units of gas need to be paid for this operation |
| CALLVALUE | 2 | two units of gas need to be paid for this operation |
| ADD | 3 | three units of gas need to be paid for this operation |
| GT | 3 | three units of gas need to be paid for this operation |
| MUL | 5 | five units of gas need to be paid for this operation |

Table 2.2: The cost of the execution of some of operations in the Ethereum blockchain

As can be seen in Figure 2.4, EVM has a stack-based architecture. Unlike most other architectures, it does not use any registers in executing the code; instead, it uses three kinds of memories: Stack memory, Volatile memory, and a Persistent memory.

**Stack memory:** As we mentioned, EVM is a stack-based virtual machine; therefore, it uses a stack to execute each operation. The maximum size of the stack is 1024, and the size of each item in the stack, which also is the machine's word size, is 256 bits. The reason for choosing this size for the stack items and machine word is that EVM uses Keccak-256 hash and elliptic-curve computations during the execution of the code. Hence, using a word size of 256-bit could facilitate these computations. Although all EVM operations could access the stack, there are some operations like *PUSH*, *POP*, *DUP*, and *SWAP* that access it most frequently.

**Volatile memory:** Alongside the stack, there is a byte addressing volatile memory in the EVM architecture. It is like the memories of the other architectures and is used during the execution of the code to temporarily hold some values. All rows of the memory are initially set to zero. Some of the EVM operations that can access the memory are *MSTORE*, *MSTORE8*, and *MLOAD*. While *MSTORE8* stores 8 bits of data into the memory, *MLOAD* and *MSTORE* can load and store 256 bits of data.

**Persistent memory:** To keep the system's state, EVM uses a non-volatile memory that is called EVM storage. This memory is used to store contracts' state variables. While the Volatile memory was a byte array, EVM storage is a key-value memory that both size of the key and the size of the value is 256 bits. All locations of this memory are initially set to zero. There are two operations that EVM uses to access the data on the storage: *SSTORE* and *SLOAD*. These two operations are used to store and retrieve 256-bit data in/from the storage.

Moreover, EVM uses a virtual ROM to keep the contract's code. Based on the Program Counter (PC) value and the available gas, the next operation is fetched from the virtual ROM and is processed using the stack, memory, and storage. There are some situations like a stack underflow exception, out of gas exception, or facing an invalid operation that in

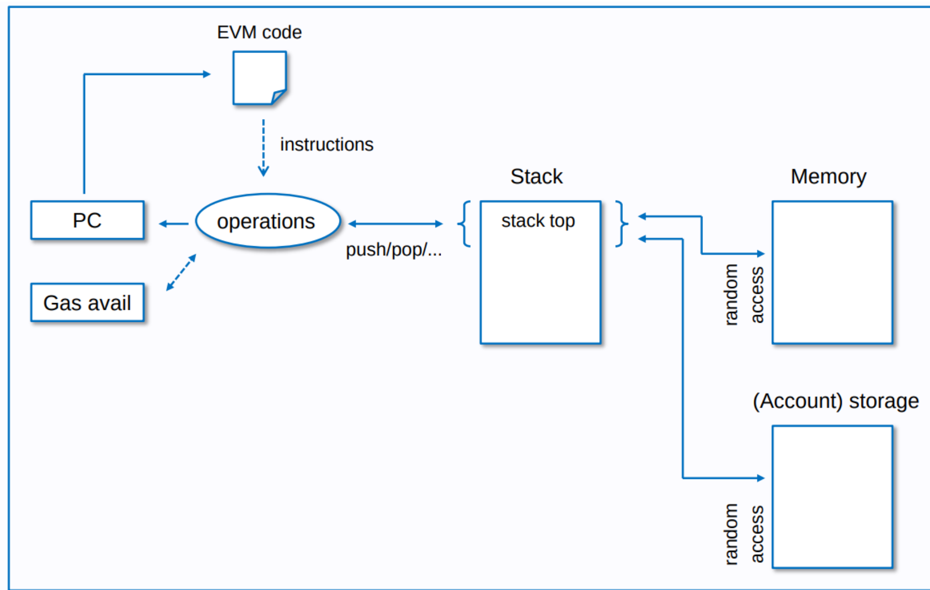


Figure 2.4: Stack-based architecture and the execution model of EVM [44]

which the execution of an EVM code may stop. In these scenarios, EVM does not change the Ethereum blockchain state.

### 2.1.5 Ethereum Tools and Technologies

There are many tools and technologies for interacting with the Ethereum blockchain. In this section, we will introduce some of them.

Ethereum network consist of many computers that are called "node". Each node has an implementation of Ethereum running on it that is called "client". There are different Ethereum clients that have been implemented in various programming languages. The following list indicates some of them:

- **Geth (`https://geth.ethereum.org/`):** Is an original implementation of Ethereum protocol that has been developed in Go programming language and is currently the most popular one.
- **OpenEhereum (`https://github.com/openethereum/openethereum`):** A fast CLI-based Ethereum client that has been developed in Rust programming language.
- **Nethermind (`https://nethermind.io/`):** Developed in C# programming language.

- **Besu** (https://consensys.net/quorum/developers/)**:** Developed in Java programming language.
- **Trinity** (https://trinity.ethereum.org/)**:** Developed in Python programming language.
- **Erigon** (https://github.com/ledgerwatch/erigon)**:** An implementation of Ethereum protocol that has been completely re-architected and developed in Go programming language.

**Web3JS:** This is a very popular set of Javascript libraries that are used for interaction with an Ethereum node [17].

**Truffle:** All programming languages have some some frameworks. For example, Java has Spring web development framework or Python has Django framework. Truffle is also a Contract development framework that is used for creation of DApps [46].

**Ganache:** This is an in-memory blockchain that DApp developers usually used for development purposes. The reason for using this personalized blockchain is that it accelerates the development, deployment, and test of distributed applications. It can be installed as CLI or GUI interfaces [45].

**Metamask:** It is an Ethereum wallet that could be used as a plugin in the web browser [31].

**Etherscan:** It is used to explore transactions, blocks, contracts, etc. on the Ethereum blockchain [20].

**Remix:** It is an online platform that can be used for the development of smart contracts. It also provides the ability to compile the smart contract with different versions of compilers. Moreover, it is possible to deploy, test, and debug the created contracts on an Ethereum blockchain [41].

**EthMiner:** It is a GPU-based Ethereum miner that uses CUDA and OPENCL. It has a command line user interface and could be easily lunched on Linux console or Window command prompt [18].

## 2.2 Model Checking

It is critical to make sure about the correctness of some types of programs like Smart contracts or safety-critical embedded systems. While some works were done to provide a method of proving the correctness of programs, they are not practical; the major weakness of these methods and techniques is that they are not scalable. It means they could not be used for large and complex programs that are currently being used. Moreover, using these methods

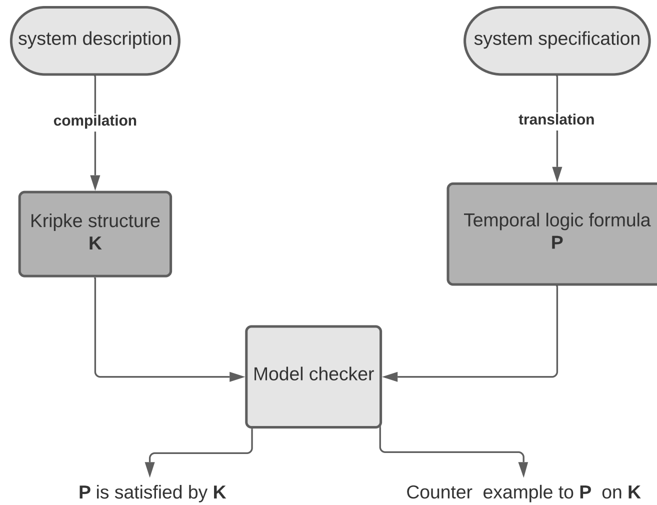


Figure 2.5: The structure of a basic model checker

requires a high level of knowledge and expertise that makes usage of these methods too complicated. These drawbacks of the suggested methods led to computer-aided verification in which the developed hardware or software is verified using another computer program. Model checking is a method for analysis of systems that could be modelled as state-transition systems. This approach is broadly using for verification purpose in computer hardware, and software [6].

Model checking requires creating a model of the system under verification. The model checking modelling technique uses a directed state-transition graph. The state-transition graph is also known as Kripke structure [26]. It could be used for the modelling of hardware and software systems. There is a property in the model checking method that its correctness needs to be evaluated by the model checker. In model checking, this property is called *specification*. There is also an algorithm in model checking that uses the finite state graph and the specification to evaluate if the property holds in the system or not. If the property does not hold in the system, then the algorithm provides a counterexample. Figure 2.5 indicates the structure of a basic model checker [6].

Although model checkers and the model checking approach have experienced a considerable progress in recent years, there are still some challenges in applying this method to real-life systems. One of the significant problems is the state explosion issue in the Kripke structure. This problem makes it difficult to use this technique for large and complex real-life systems. The other challenge is that for many systems, we are not able to use the Kripke structure and the temporal logic specification. Hence in those systems, the basic model checking is not

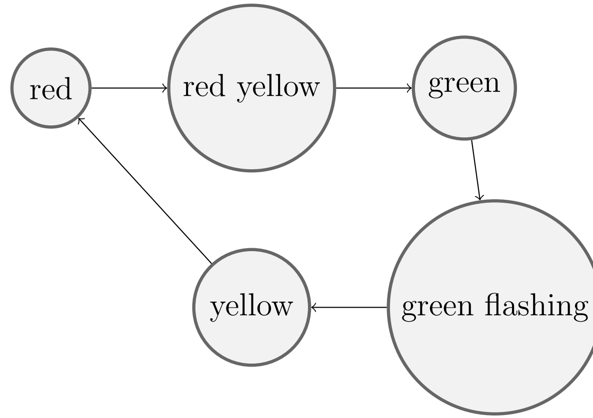


Figure 2.6: The Kripke structure of an Australian traffic light

thoroughly feasible. However, model checking has advantages that make it a good choice for systems verification. Firstly, the model checker could be systematically created and then used by the system developers without deep knowledge in verification. Secondly, model checking could be easily used in different steps of the development of the system, from the design process to programming. Moreover, model checking can manage concurrency that could be the origin of many bugs and errors in the system under verification [6].

### 2.2.1 Kripke Structure

Kripke structure that is used in model checking to model the system under verification has been proposed by Saul Kripke [26]. Kripke structure is a finite directed graph. The graph's vertices indicate the states of the system under verification, and the edges of the graph represent the state transitions of the system. A labeling function maps a set of atomic propositions to the graph's vertices that indicate a state in the system.

**Formal Definition:** A Kripke structure over a set $A$ of atomic propositions is a 4-tuple $K = \langle S, I, R, L \rangle$ where [6]:

- $S$ is a finite set of states. A state is a valuation of a set of variables, where each variable takes a value from a finite set which corresponds to program variables in software)
- $I \subseteq S$ is a set of initial states
- $R \subseteq S \times S$ is a set of transitions
- Labeling function $L : S \to 2^A$

The existing atomic propositions in each vertex of the graph indicate the atomic proposition is True in the corresponding state, and the other atomic propositions in this state are False.

The system's behavior is represented using the paths of the graph in the Kripke structure. A path in the Kripke structure is a sequence of states in the graph $P = S_0, S_1, S_2, ...$ where for $i > 0$, $(S_i, S_{i+1}) \in R$. Figure 2.6 indicates the Kripke structure of an Australian traffic light. In this example, a state is associated with the value of traffic light variables where each variable could have the value of on, off, or flashing. Here, there are five states and four state transition where each state could be a subset of red, yellow, and green.

### 2.2.2 Specification

The specification that its correctness needs to be evaluated by the model checker is a temporal logic formula. The temporal logic formula could be in the form of Computation Temporal Logic (CTL) or Linear Temporal Logic (LTL) [6]. In the subsequent sections, we will introduce nuXmv, which is a model checker based on NuSMV. We will discuss its approach for modeling of the system under verification and also its specification.

### 2.2.3 SAT-Based Model Checking

SAT solvers have had considerable progress in recent years; hence this progress made it possible to use them in model checkers as their core technology and enhance their quality. It is specially used in Bounded Model Checking (**BMC**). The main idea behind BMC is that representing the system model and the specification in propositional logic. In other words, the model checking problem is reduced to a satisfiability problem; then, it will be passed to a modern sat solver. Moreover, a bound k will be used as the maximum number of steps to evaluate the formula's satisfiability. In this research, we use BMC for model checking of the system under verification.

### 2.2.4 NuSMV

In this research we needed to use a model checker. The model checker that we use is nuXmv which is based on NuSMV model checker. NuSMV is a state of the art symbolic model checker that supports both SAT-based and BDD-based model checking techniques. Moreover it has a thorough documentation that could be helpful during modeling of the system under model checking. In order to use NuSMV, the system needs to be modeled as a finite state machine and the correctness of the specification which could be in form of CTL or LTL, is evaluated using a BDD-based or SAT-based methods. Furthermore there are heuristic techniques to avoid the state explosion issue as much as possible. This model checker could be used through a command line user interface.

**Input Language**

In this section we will briefly describe the syntax of the input language of NuSMV. As its name indicates NuSMV uses SMV language as its input language. the following data types are supported by the NuSMV input language [5].

- **Boolean:** This data type can have the symbolic value of TRUE or FALSE
- **Integer:** In NuSMV the integer value could be in range of $-2^{31}+1$ to $2^{31}-1$.
- **Enumeration Types:** This data type could include a enumeration of integer numbers and symbolic constants. For instance $\{1, TRUE, -2, OK\}$ could be its value.
- **Word:** Using this data type we can model a vector of bits with different sizes. The bit vector could be signed or unsigned. For example "unsigned word[8]" is an unsigned vector of bits with a length of 8.
- **Array:** In NuSMV we can define an array of a data type by specifying the upper and lower bound for its index. For example, "array 0..3 of boolean", is an array of boolean values and its indices could be from 0 to 3. Despite the other languages in NuSMV the index of an array must be a constant integer.

Moreover, there is a "Set Type" in NuSMV. It could be boolean set, integer set, symbolic set, and integer-and-symbolic set.

The following operations is possible in NuSMV.

- **Logical and Bitwise**
- **Comparison**
- **Relational Operators**
- **Arithmetic Operators**
- **Remainder Operator "mod"**
- **Shift Operators**
- **Index Subscript Operator**
- **Bit Selection Operator**
- **Word Concatenation Operator**

```
MODULE main
VAR
start : boolean;
done : boolean;
state : {idle,cyc1,cyc2};
test : test_module(state);

CTLSPEC AG done = FALSE;

ASSIGN
done := (state = cyc3);
init(start) := TRUE;
init(state) := idle;
next(state) :=  case
    state = idle & start : cyc1;
    state = idle : idle;
    state = cyc1: cyc2;
    state = cyc2: cyc3;
    state = cyc3: idle;
esac;

MODULE test_module(state)
VAR
value : 0 .. 100;
ASSIGN
init(value) := 0;
next(value) := case
    value >= 94 : 0;
    state != idle : value + 7;
    TRUE : value;
esac;
```

Figure 2.7: A sample code of NuSMV input language

- **Word sizeof Operator**
- **Resize Word Conversions**
- **Set Expressions**
- **Inclusion Operator in**

The input language of NuSMV also supports a version of switch case and if-else statement. More details about the input language of the model checker could be found in its user manual [5]. Figure 2.7 indicates a sample code of NuSMV input language. As can be seen, there are two modules in the sample code. One of them is "main" and the other one is "test_module". In the main module there are two boolean variables: "start" and "done". There is also an enumeration variable that could have the value "idle", "cyc1", or "cyc2". Moreover there is an instance of the test_module variable in VAR section of the main module (line 2 to 6). In ASSIGN section of each module of NuSMV input language, the initial and also the next

value of a variable could be specified. The value of "done" variable is set at line 11. It will be TRUE if $state = cyc3$ and otherwise it will be FALSE. The initial value of "start" and "state" variables are set at line 12 and 13 accordingly. Also the next value of "state" is determined in a "case" statement in lines 14 to 20.

#### NuSMV Specifications

In previous sections, we introduced the NuSMV input language that is used to model the system under model checking. This section will indicate how the specification that its correctness needs to be checked is expressed in NuSMV. In NuSMV, the specification could be in the form of temporal logic like LTL and CTL. For example, a specification in the form of the CTL formula can be seen in line 8 of Figure 2.7. NuSMV will evaluate the correctness of the specification, and if it is detected to be false, then NuSMV generates a counter-example of a path in the state machine that leads to the falsity of the specification.

**CTL:** In NuSMV, a CTL specification could have one of the following forms:

```
1 ctl_specification :: CTLSPEC ctl_expr [;]
2                     | SPEC ctl_expr [;]
3                     | CTLSPEC NAME name := ctl_expr [;]
4                     | SPEC NAME name := ctl_expr [;]
```

More information about the syntax of a CTL specification in NuSMV could be found in the user manual [5].

**LTL:** In NuSMV an LTL specification could have one of the following forms:

```
1 ltl_specification :: LTLSPEC ltl_expr [;]
2                     | LTLSPEC NAME name := ltl_expr [;]
```

More information about the syntax of an LTL specification in NuSMV could be found in the user manual [5].

### 2.2.5 nuXmv

The model checker that is used in this research is nuXmv [4]. nuXmv is a model checker that has been developed based on NuSMV. Hence it covers all functionality and features of NuSMV and supports the model created for NuSMV. Moreover, It has added more features and improvements to NuSVM. Some of the noticeable differences are as follows:

- Adding "real" and "integer" as two new types.

- nuXmv does not support "process" keyword
- Adding a set of new commands to the CLI interface

## 2.3 Related Works

This section will briefly describe some of the prior works that focus on the Ethereum blockchain and the smart contracts' reliability and security.

Ethor [42] is a sound and automated static analyzer tool for Ethereum smart contracts. This static analyzer gets an abstraction of EVM code as input and applies the static analysis on the abstract. The abstract is created using Horn clauses. For the correctness evaluation, the Ethereum virtual machine tests [15] are used.

ContractLarva [1] is a run-time verification tool for EVM smart contracts but this tool have not been tested over specific test-cases. E-EVM [34] and Erays [50] are two smart contract analyzing tools. These tools help users to read and understand the EVM code easier. E-EVM simulates the Ethereum virtual machine. It disassembles the input EVM code and creates its CFG. It also indicates the EVM stack state after execution of every EVM operation. Erays also disassembles an input EVM code but it also turns the output stack-based assembly code into a registered base assembly code. This change could help the readability of the assembly code. They also provide a pseudo-code of the smart contract that indicates the smart contracts functions declaration.

Ethertrust [24] has been developed by Ilya Grishchenko et al. before their recent work Ethor [42]. Both of these tools are EVM static analyzers but the former has been tested on real word smart contracts from the Ethereum blockchain but the latter has been tested on semantic tests of the official EVM suite. FsolidM [29] provides a graphical user interface in which the user can design its smart contract in form of Finite State Machine (FSM). This state machine will be verified using nuXmv model checker. Then FsolidM generates the Solidity code of the smart contract using the input state machine. KEVM [25] is a tool that uses K framework to determine the semantics of EVM. By specifying the formal semantics of EVM, KEVM can be used for formal verification of the smart contract. To make sure about the correctness of their semantics they used the official Ethereum VMTests test suite. MAIAN [32] is another smart contract analyzing tool that uses symbolic execution of EVM code. MAIAN has been tested using 970,898 smart contracts currently running on the Ethereum blockchain (It has been mentioned in the paper that the distinct EVM codes are considerably less than 970,898). Manticore [28] is another smart contract analysis tool. This tool uses a

symbolic analysis technique to find vulnerabilities of the given bytecode of the smart contract. To test their tool they have created different smart contracts and included different types of vulnerabilities inside them and showed that their tool can detect the vulnerabilities.

There are many other EVM analysis tools. Most of the tools use a set of solidity contracts or bytecode of them to evaluate themselves. For example on the solidity level, they use a collection of available solidity contracts for example, the solidity codes on Etherscan. The main reason for using the collection is that it is available! because smart contract developers do not deploy their high-level solidity code on Ethereum blockchain and it is only available in form of EVM code. There are also some standard benchmarks like Ethernaut [37] or Not so their Smart Contracts [36] could be used.

As we mentioned before, Schneidewind et al. [42] have used the Ethereum official test suite to evaluate Ethor. The important point here is that the test suite is not the test that evaluates Ethor as a static analysis tool of Ethereum Smart contracts. Authors of Ethor paper in their previous work [23] provided a small-step semantics of EVM code. The test suite has been used to validate the semantics. The suggested semantics have been used in Ethor and the authors have mentioned again that they have used the test suite to validate Ethor. What they are actually doing is that they are indicating that the small-step semantics of EVM code that has been used in Ethor, has been evaluated beforehand. For evaluating Ethor itself they have used a limited number of contracts on the Ethereum blockchain to test specific vulnerabilities like Reentrancy.

Durieux et al. [14] worked on the evaluation and comparison of 9 state-of-the-art smart contracts analysis tools. They have listed many tools but they only selected 9 of them. To evaluate these tools they needed a proper dataset. They used two different datasets.

First, a collection of 47,587 smart contracts that are already available on the Ethereum blockchain and their solidity code is also available on Etherscan. They use this database to evaluate and compare the ability of these tools in detecting vulnerabilities of the existing smart contracts on the Ethereum blockchain. To collect these contracts, they have used Google BigQuary to gather addresses of all available contracts on the Ethereum blockchain. Then they used the API of Etherscan to collect the solidity code of the addresses. As for most of the contracts, their solidity code is not available, and most of the contracts are duplicates on the Ethereum blockchain; they finally obtained 47,587 solidity code of smart contracts. In other words, at that time, there were 2,263,096 contracts on the Ethereum blockchain that 972,975 of them had solidity files on Etherscan, and only 47,587 were unique.

The second dataset that has been used is a collection of 69 solidity code of smart contracts that have been classified into 10 different categories of known smart contract vulnerabilities. This classification leads to a better evaluation of these 9 tools. The smart contracts have been gathered mostly from different Github repositories. There are also some contracts in this dataset that have been gathered from the Ethereum blockchain (contracts that are known to have specific vulnerabilities) [21]. Using these two datasets, they have analyzed the mentioned tools using the execution framework that they have provided.

SmartACE [43] is a framework for the verification of the smart contracts written in Solidity programming language. Right now, it supports the Solidity version of 0.5.9. It could be built from the source code, or the docker container could be easily used.

# Chapter 3

# Smart Contracts Classification

Numerous contracts have been deployed on the Ethereum blockchain so far. To deploy a smart contract on the Ethereum blockchain, the developers need to compile their contracts, and they only need to deploy the bytecode of their contracts on the blockchain. Therefore, the solidity code of a significant number of the deployed contracts on the Ethereum blockchain is not publicly available. Thus it is not straightforward to examine the contracts' bytecode to determine what the contracts are doing. Although, some smart contract developers deploy the solidity code of their contract on Github and Etherscan[1] [20], there are thousands of contracts on the websites. Hence, it is almost impossible to go through all of them manually one by one to realize what is happening in these contracts. Classification of the smart contracts would be helpful to get an understanding of the content of the available contracts on the Ethereum blockchain and discover the common contracts.

In section 3.1 we go through some Ethereum smart contracts classification methods that tried to detect various types of smart contracts and categorize them into several classes. These classifications are important because the classifications' result determines the most popular contracts, the most active contracts, garbage contracts, dormant contracts, etc., on the Ethereum blockchain. Researchers could use this information and help them only focus on specific types of contracts on the blockchain that are more relevant to their research topic. In section 3.2 we are going to exploit some of the mentioned approaches in section 3.1 of the

[1] **Etherscan Verification:** As only the EVM code of a contract exists on the Ethereum blockchain, how can users trust the contract owner to sign it? One way is that the users ask the contract owner or developer to provide the contract's flattened solidity code. By flattening, the owner must include the libraries' code employed in the contract, not just the libraries' name. Then the users can compile the solidity code and obtain the EVM code. Then, as they have the smart contract address on the Ethereum blockchain, they can access the deployed EVM code on the blockchain and compare it with the EVM code that they have created by compiling the provided solidity code. Etherscan is doing the same thing to verify the smart contracts. The contract's owner uploads the flattened solidity code with some extra information on Etherscan, and Etherscan verifies it by comparing the EVM code with the existing EVM code on the Ethereum blockchain.

chapter and provide some classifications on the Ethereum smart contracts.

## 3.1 Some Classification Methods

The first method that we will evaluate has been proposed by Fröwis *et al* [22]. In their work, they seek to detect Standard tokens on the Ethereum blockchain. They presented two different methods for identifying standard tokens on the blockchain. Both of these methods are applied to the bytecode of the smart contracts. These two approaches are behavioural-based and Signature-based. Using both methods, they showed that until 2018 about 33% of contracts on the Ethereum blockchain were token contracts.

**Signature-based Method:** As the interface of ERC20 standard token is available. The function signatures of all functions that have been used in the interface are also available. There are six mandatory functions on the ERC20 standard token interface, so in the Signature-based approach, they just checked the presence of the function signatures in the bytecode of the contract. A Function dispatcher exists at the beginning of the bytecode of a smart contract. The signature of all functions that have been used in the contract could be found in the function dispatcher part. So the only thing that needs to be done is to compare the function signatures with the function signatures that exist in the interface of the ERC20 standard token.

Although this method works appropriately in recognizing ERC20 tokens and the experimental result also confirms that claim, false positives are also possible in this approach. Because some developers may have these functions in their smart contract code, but they never use them. False-negative is also possible in this approach. Because some contract developers may rename the name of functions in the ERC20 standard token, but their implementation is ERC20 standard token.

**Behavioral-based Method:** The behavioural-based method is based on the symbolic analysis of programming languages. Using symbolic analysis of the smart contract and the ERC20 standard interface, they can determine how much the contract's behaviour is similar to the ERC20 standard token contract by analyzing the execution path's effect on the memory and storage. The disadvantage of this approach is that it overlooks communications between contracts. Besides, the tool-chain they are using is heavily under development, affecting the result.

M. di Angelo *et al.* [12] have done a thorough evaluation of the Ethereum blockchain. Their work concentrated on varieties of smart contracts on the Ethereum blockchain, The quantity of each type, and the contracts' activity on the Ethereum blockchain. In their work,

they evaluate two claims. First, Most of the contracts on the blockchain are unused. Second, the most popular contracts on the Ethereum blockchain are Token contracts.

The different types of contracts that they evaluated are as follows:

1. Dormant
2. Active
3. Self-destructed
4. Short-Lived
5. Prolific
6. Token
7. Wallet
8. GasToken
9. Attack
10. ENS deed

They employ different methods to detect each of the categories. The methods are as follows:

1. The Data (using messages and logs provided by the Ethereum client)
2. Static Analysis
   (a) Code skeletons
   (b) Function signatures
   (c) Event signature
   (d) Code patterns
   (e) Symbolic execution
3. Dynamic Analysis
   (a) Time stamps
   (b) Message statistics
   (c) Temporal patterns of messages
   (d) Log entry analysis

4. Combined Approaches
    (a) Interface method
    (b) Blueprint fuzzing
    (c) Ground truth

According to the results, both of the claims seem to be true. Most blockchain contracts are unused, and Token contracts are the most active contracts on the blockchain.

In the Ethereum blockchain, some different contracts control access to tokens and currencies. We identify this type of contract as a wallet. It seems that a considerable amount of contracts on the blockchain are wallet contracts. To identify these types of contracts, Di Angelo et al. [13] Provided a blueprint of various types of wallet contracts. These blueprints could be used in the signature-based smart contract detection method. Using the interfaces, the blueprints, and the signature-based method, we can classify the smart contracts of the Ethereum blockchain.

Di Angelo et al. analyzed some contracts' bytecode and interfaces to discover potential wallets in this work. Then, according to the potential wallets' characteristics, classified them into different categories and then provided a blueprint for each category. The blueprints specify some mandatory functions in each category that the required functions could identify that category's wallets.

Norvill et al. [33] proposed a framework for classifying smart contracts on the Ethereum blockchain. As only the bytecode of the smart contracts is publicly available on the Ethereum blockchain, this work's motivation is to develop a method to classify the contracts into different groups according to their similarity and stick a label to them. The work's most contribution is suggesting a method for automatically labelling the unknown contracts on the Ethereum blockchain, only using their bytecode. In this work, the source code of verified contracts has been used as the clustering method's input. According to the names of the verified contracts, they have provided the following labels.

1. token
2. dao
3. withdraw
4. dice

5. presale
6. factory
7. ether
8. ponzi
9. eth
10. asset

The input to this framework is an unknown bytecode, and it will stick one of the mentioned labels to it. Thus we can speculate what the purpose of the unknown contract on the Ethereum blockchain is.

As many empirical studies need a classified source code repository, Pierro et al. provided Smart Corpus [39]. Smart Corpus can provide a list of contracts to its users according to different parameters. These parameters are the Pragma version of the solidity code, Source lines of code, number of functions, number of modifiers, and number of payable. Users have three options for each of these parameters to filter the output contracts. For the Pragma version, they can select versions 4, 5, or 6. For the other parameters, they can choose greater than 1, greater than 10, or greater than 100. After specifying the filters, users can submit their requests to Smart Corpus. The result is a list of addresses of smart contracts that can be downloaded. The user can see the exact Pragma version for each of the contracts, number of lines, comments, mappings, functions, payable, events, and modifiers. There is also a field in the output named "Addresses" that shows how many variables of address type exist in the solidity code. This tool also considers the duplicate solidity code on the Ethereum blockchain.

Pierro et al. also provided PASO a Smart contract static analysis tool. The tool's input is the address of a smart contract on the Ethereum blockchain or a solidity code of a smart contract that the user has developed. The output of PASO is like the information provided for each contract in Smart Corpus, but it also shows libraries and interfaces that have been used in the input contract.

## 3.2 Implemented Classification Methods

Here we are going to apply some classification to the Ethereum blockchain. In section 3.2.1, we will classify the smart contracts on the blockchain according to their size and explore the relation between the size and behaviour of the contracts. In section 3.2.2, we will use the

function signature method to classify the verified contracts on the Ethereum blockchain. In Section 3.2.3, we use the smart contract classification according to their bytecode size and the signature-based method to specify how many ERC20 standard tokens exist in each class of bytecode size.

### 3.2.1 Bytecode Size classification

To classify the Ethereum blockchain contracts according to their bytecode size, we first need to access the contracts. To access contracts on the Ethereum blockchain, we use Google Cloud BigQuery. We need to run a query on the database to access the data. There are two options to run this query for a different range of bytecode sizes. First, we can run them manually on Google cloud BigQuery. As this approach is too time-consuming, we created a Python script for using Google cloud BigQuery's API. However, there is a problem. The query can only be executed a limited number of times. Thus, even using a python script, we need to wait for several days to gather the data. The query that we ran on the database is as follows:

```
1 SELECT count(distinct(bytecode))
2 FROM `bigquery-public-data.ethereum_blockchain.contracts`
3 WHERE length(bytecode) > 3 and length(bytecode) < 100;
```

As the contracts bytecode sizes are between 0 and 50,000 bytes, We decided to have two bytecode size steps for the classification. First, we classified the smart contracts with a bytecode size of less than 2000 with a step size of 100 (e.g. 0-100, 100-200, etc.). Then with a step size of 1000 for bytecode length between 0 and 50000. Table 3.1 indicates smart contracts with a bytecode size between 0 and 2000. As it can be seen, we have divided the bytecode length into intervals with a length of 100.

Table 3.2 indicates smart contracts with bytecode size between 0 and 50000. As it can be seen, we have divided the bytecode length into intervals with a length of 1000. At the time of this research, the total number of contracts with distinct bytecode on the Ethereum blockchain is 253,628.

Now we are going to evaluate two groups of smart contracts. Smart contract with the length of bytecode in the range of 0 to 100 and smart contracts with a length of bytecode in the range of 200 to 300. We will pick the bytecode of some of the contracts from these two ranges randomly and de-compile them into their equivalent solidity code to determine what they are doing. To access the bytecode and address of the contracts, we ran the following query on the Google Cloud BigQuery database.

| Bytecode Length | Number of Contracts | Bytecode Length | Number of Contracts |
|---|---|---|---|
| 0–100 | 17665 | 1000–1100 | 1325 |
| 100–200 | 2884 | 1100–1200 | 2108 |
| 200–300 | 893 | 1200–1300 | 1457 |
| 300–400 | 5286 | 1300–1400 | 1530 |
| 400–500 | 1025 | 1400–1500 | 2467 |
| 500–600 | 945 | 1500–1600 | 2917 |
| 600–700 | 924 | 1600–1700 | 1698 |
| 700–800 | 1023 | 1700–1800 | 1168 |
| 800–900 | 6295 | 1800–1900 | 1095 |
| 900–1000 | 1875 | 1900–2000 | 1355 |

Table 3.1: Number of distinct contracts with a bytecode length between 0 and 2000

```
SELECT address, bytecode
FROM `bigquery-public-data.ethereum_blockchain.contracts`
where length(bytecode) > 3 and length(bytecode) < 100 and
bytecode in (
SELECT bytecode
FROM `bigquery-public-data.ethereum_blockchain.contracts`
group by bytecode HAVING COUNT(bytecode)=1)
```

We need to randomly pick the EVM code of some of the contracts and turn them into their equivalent solidity code. A good approach is using Etherscan's API. The solidity code of many of the contracts already exists on Etherscan.io. By providing the address of a smart contract, Etherscan returns the solidity code of that contract. We wrote a python script, and using that script, we requested the solidity code of all the distinct contracts with a bytecode range between 0-100 and 200-300 to retrieve their solidity code, if they exist on Etherscan. Many addresses for each distinct bytecode may exist, but we request only one of them from Etherscan. If we request for solidity code of all contract addresses from Etherscan, it will take several days. However, if we could not retrieve enough solidity code, we would run it for all addresses in the mentioned ranges.

We requested the solidity code of about 17,000 contracts with a bytecode size between 0 and 100. As a result, we only received the solidity code of 13 contracts. We ignored the distinct constraint on bytecode and requested the solidity code of more addresses to obtain more contracts, but it did not change the result. We only received more copies of the 13 solidity files. We requested a solidity code of about 700 distinct contracts with bytecode sizes between 200 and 300. As a result, we received 49 solidity files from Etherscan. The sample contracts have been listed in Appendix A. By looking at these contracts, it can be seen that most of the contracts with bytecode lengths of less than 300 are not applicable contracts.

| Bytecode Length | Number of Contracts | Bytecode Length | Number of Contracts |
|---|---|---|---|
| 0-1000 | 38815 | 25000-26000 | 1287 |
| 1000-2000 | 17120 | 26000-27000 | 1023 |
| 2000-3000 | 16162 | 27000-28000 | 1078 |
| 3000-4000 | 13614 | 28000-29000 | 842 |
| 4000-5000 | 19713 | 29000-30000 | 719 |
| 5000-6000 | 16153 | 30000-31000 | 681 |
| 6000-7000 | 16030 | 31000-32000 | 634 |
| 7000-8000 | 9853 | 32000-33000 | 637 |
| 8000-9000 | 16338 | 33000-34000 | 666 |
| 9000-10000 | 11703 | 34000-35000 | 502 |
| 10000-11000 | 9080 | 35000-36000 | 438 |
| 11000-12000 | 12508 | 36000-37000 | 457 |
| 12000-13000 | 5980 | 37000-38000 | 480 |
| 13000-14000 | 5194 | 38000-39000 | 422 |
| 14000-15000 | 4815 | 39000-40000 | 350 |
| 15000-16000 | 3774 | 40000-41000 | 440 |
| 16000-17000 | 5352 | 41000-42000 | 475 |
| 17000-18000 | 2959 | 42000-43000 | 513 |
| 18000-19000 | 2601 | 43000-44000 | 390 |
| 19000-20000 | 2166 | 44000-45000 | 322 |
| 20000-21000 | 2053 | 45000-46000 | 335 |
| 21000-22000 | 1605 | 46000-47000 | 378 |
| 22000-23000 | 1622 | 47000-48000 | 352 |
| 23000-24000 | 3098 | 48000-49000 | 512 |
| 24000-25000 | 1259 | 49000-50000 | 128 |

Table 3.2: Number of distinct contracts with bytecode length between 0 and 50000

As we mentioned before, there are 253,628 distinct contracts in the Ethereum blockchain, that the length of the bytecode of the contracts is between 0 and 50000. Now we are going to provide another classification for the contracts. To do this classification, we first divide the contracts according to their bytecode length. We consider the following ranges:

```
1) 0 - 5000
2) 5000 - 10000
3) 10000 - 20000
4) 20000 - 30000
5) 40000 - 50000
```

| Bytecode Length | Number of Contracts | Verified | Percent |
|---|---|---|---|
| 0-5000 | 105424 | 9774 | 9.27 |
| 5000-10000 | 70077 | 23054 | 32.9 |
| 10000-20000 | 54429 | 20777 | 38.17 |
| 20000-30000 | 14586 | 4197 | 28.77 |
| 30000-40000 | 5267 | 1627 | 30.89 |
| 40000-50000 | 3845 | 1161 | 30.19 |

Table 3.3: Number of verified contracts in Etherscan for 6 different ranges.

As, at the time of this research, the bytecode length of the largest contract on the Ethereum blockchain is 49,154 bytes. Hence, there is no range after 40000 to 50000.

Now we want to know how many contracts in each range have been verified in Etherscan. Using Table 3.2 we can specify the total number of contracts in each of these ranges. We queried the smart contracts' addresses in each range from Google BigQuery. Using Etherscan's API, we derived and counted the total number of verified contracts in each mentioned range.

As it can be seen from Table 3.3, 105,424 contracts are in the range 0-5000, but only 9.27 percent of them have been verified in Etherscan. We have randomly picked some contracts from the verified contracts with a bytecode length in the range 0-50000. The solidity code of the selected verified contracts shows that even some of the verified contracts are not valuable contracts. As the number of contracts is too large, especially the first three ranges, and Etherscan's API has five requests per second limit, running the script took several days.

We showed that even among verified contracts by Etherscan, there are some garbage contracts. As it is not possible to go through each of the contracts one by one, we are going to pick some of the contracts from each range of Table 3.3 randomly and examine them manually to determine how many of them are qualified contracts and how many of them are useless contracts. Let us consider the number of verified contracts as our population size. We need to examine 17 contracts from each range randomly to have a Confidence Level of 90% and 20% Margin of Error [40].

For verified contracts with bytecode lengths between 0 and 5000, we randomly selected 17 of them. Among these 17 contracts, 3 of them with the smallest bytecode length were useless contracts, and the others seemed to be typical contracts on the Ethereum blockchain. Therefore, with 90% of confidence level, we can say 17.6 percent of contracts with a bytecode range between 0 and 5000 are useless contracts on the Ethereum blockchain. We continued this evaluation for the other specified ranges. We selected and evaluated 17 contracts of the

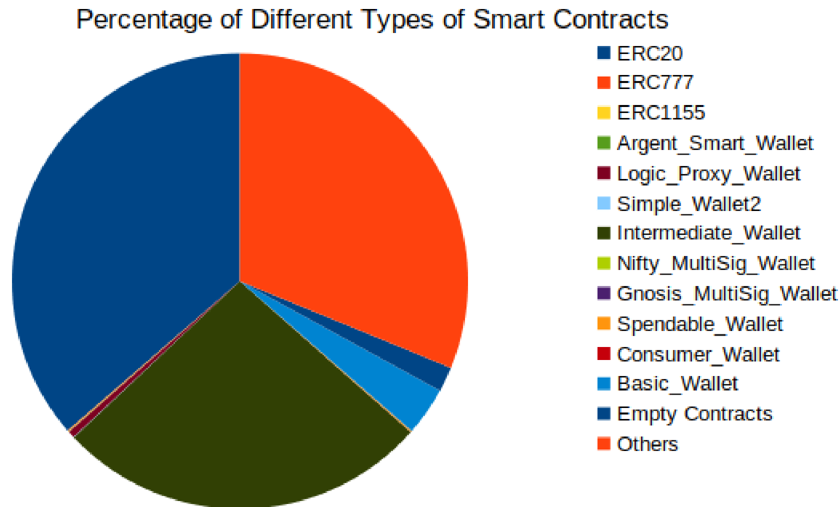


Figure 3.1: Classification of the verified smart contracts using the signature-based method

five other ranges. Almost all of them were among applicable contracts or at least an initial version of a valid contract. One impressive result from the randomly selected contracts is that nearly all of them are tokens based on ERC20 standards except the first range.

### 3.2.2 Function Signature based Method

As the solidity code of the verified contracts already exists on Etherscan, we used the solidity code and compared it with the ERC20 standard token interface to detect if the contract is an ERC20 standard token or not. Fröwis et al. [22] did the same research to identify Standard tokens on the Ethereum blockchain. They presented two different methods for discovering standard tokens on the Ethereum blockchain. Both of these methods are applied to the bytecode of the smart contracts. These two methods are behavioural-based and Signature-based. The Signature-based method is applied to the smart contracts' bytecode. There are some mandatory functions on the ERC20 standard token interface, So in the Signature-based method, they just checked the presence of the function signatures in the bytecode of the contract. As the standard tokens interface is available and we also have access to the blueprint of different types of wallet contracts, We have used the signature-based method on the verified smart contract on the Ethereum blockchain and classified the verified contracts into different standard tokens and wallets.

As can be seen in Figure 3.1, after Token contracts, Wallets are the most popular contracts on the blockchain.

### 3.2.3 Combined bytecode size and signature method

During collecting and evaluating the verified smart contracts from the Ethereum blockchain, we noticed that many smart contracts are ERC20 standard token contracts. We want to know how many of the contracts on the Ethereum blockchain are ERC20 standard tokens exactly. Using this information, we can only concentrate on the security of a specific part of smart contracts on the blockchain.

As we have already collected the solidity code of verified smart contracts on the Ethereum blockchain, We need to evaluate these solidity codes one by one to see how many of them are using the ERC20 standard token's interface. The solidity code of the ERC20 standard token is available online [47]. In this standard, the optional and mandatory functions have been specified. Therefore we need to go through the solidity code of each of the smart contracts and see if they have used the ERC20 contract functions and, If yes have they used the mandatory functions or not. Then we can confirm that the contract is an ERC20 standard token or not.

To evaluate a solidity code, we need to parse it to a list of contracts and functions in each of the contracts. Then we can detect if it is an ERC20 standard token or on. To parse the solidity code, we used Antlr [38]. Antlr (Another Tool for Language Recognition) is a powerful parser generator that helps create a parser for reading, processing, executing, or translating structured text or binary files. To parse a specific program using regular expression is not enough, especially when the program uses recursion, but with the help of Antlr, it becomes more accessible, faster, and mess-free. Antlr is used to build languages, tools, and frameworks. With the help of Antlr, one can generate a parser that can create and walk parse trees in their specific target language. It helps define grammar for a language, data format, diagram, or structure represented with text for analysis. We have used a solidity parser for python built on the ANTLR 4 grammar file for the solidity programming language. It helped to generate the solidity parse tree based on the given solidity grammar file. It allows us to write code that handles the parser output in the Python programming language.

So we have a python script that its input is a solidity file, and its output is a boolean that shows if the solidity code matches the ERC20 standard token or not. We ran the script according to the classification we provided in Table 3.3. Figure 3.2 indicates the result. As it can be seen, the maximum number of ERC20 Standard tokens are among smart contracts with bytecode length between 5000 to 10000. About 46% of contracts in this range are ERC20 standard tokens. The lowest number of ERC20 standard tokens are among smart contracts with bytecode ranges between 0-5000 and 40000-50000 with about 26% of the total number of contracts in these ranges. According to our results, about 42% of the total number of verified contracts on the Ethereum blockchain are ERC20 standard tokens. Consider that these number of contracts are exactly following the ERC20 standard token interface. There

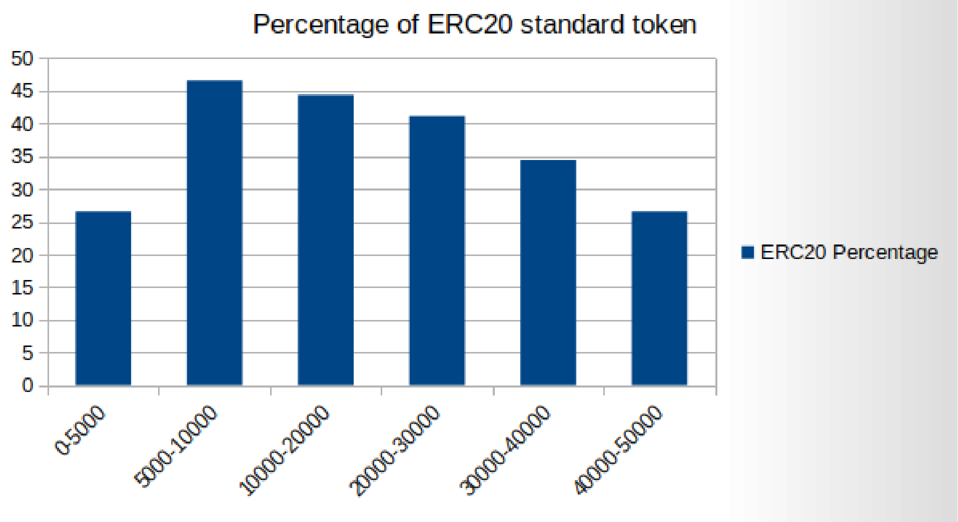


Figure 3.2: ERC20 standard tokens in each specified range according to smart contracts bytecode length

are also other contracts on the Ethereum blockchain that are contract tokens but have not implemented the ERC20 standard token exactly. Therefore our method cannot detect them as an ERC20 standard token.

# Chapter 4

# Smart Contracts to SMV

To verify the EVM code of a Smart contract in nuXmv, we must reduce it into SMV code. In Section 2.1.4, we discussed the EVM architecture and some of the EVM operations. More information on EVM architecture and the execution of EVM code in the virtual machine could be found in the yellow paper [49]. In this chapter, we review the halting problem and distinguishing feature of our approach in Section 4.1. We will explain the detail of our design and implementation of the Ethereum Virtual Machine architecture in SMV language in Section 4.2. Section 4.3 gives a broad view of the design and implementation of the Python code. Our approach toward implementing each of the EVM operations in SMV has been described in Section 4.4.

## 4.1 Halting Problem for Smart Contracts

The halting problem is undecidable for "Turing Complete" programming languages like C or Java. The EVM code also has all features to be considered Turing Complete. Hence, the Halting problem should be undecidable under it. However, the execution of each operation in the Ethereum Virtual Machine has a specific cost, and the generators of transactions specify the total amount of gas that they are willing their transaction consume. Hence, every code that executes inside the virtual machine will halt after the gas limit reaches zero. Therefore, the halting problem is decidable if implemented in EVM code and run inside the Ethereum Virtual Machine.

Given the undecidability, in general, of verification of smart contracts, every approach of which we are aware makes some finitizing assumptions. Our distinguishing aspect is that we explicate these assumptions as inputs to model checking. We have considered the Gas limit as an input to the tool in the form of a variable that could have integer values. Moreover, we

have some other inputs to our tool as follows:

- **Word Length:** size of each word in the Ethereum virtual machine is 256 bits. However, in our tool, we can set a custom size for it.
- **Stack Size:** the maximum stack size in EVM is 1024. However, we can set a smaller value to it in our model checking.
- **Memory Size:** we can set the size of EVM memory in our model checking.
- **Storage Size:** storage size is also an input of our tool and could be specified at the beginning of the model checking process.

## 4.2 EVM Architecture in SMV

In our SMV implementation of EVM architecture, we have five main modules. In the following sections, we will explain each of them in more detail.

### 4.2.1 Main module

This is the main module in SMV code that all variables and the other modules will be defined and used inside this module. We have the following variables in our main module.

- **operationName:** This variable consists of all EVM operations that exist in the smart contract. As some EVM operations need more than one state in SMV code to be executed properly, we needed to add some dummy operations in this variable to support those operations.
- **start:** This is a boolean variable that indicates the execution of the code has been started or not.
- **end:** This is a boolean variable that indicates the execution of the code has been finished or not.
- **gas:** This is an unsigned word variable that indicates the provided gas for the execution of the smart contract. The user of the tool will specify the variable's initial value, and the execution of each operation will decrease it by a specific amount.
- **operationArray:** This is an array of unsigned word values. Each EVM operation may have a parameter. Parameters of EVM operations will be stored in this variable and will be used during the implementation and execution of the operations in NuSVM.

- **stack:** EVM is a stack-based architecture. Thus we need to have a stack in our SVM code. This variable is an array of unsigned word values that we use as the stack in our SMV code.
- **stack_head:** This is an integer variable that is used as a pointer to the top of the stack. In our design, stack_head points at the first empty space on top of the stack.
- **memory:** This variable is an array of unsigned word values that the word size is 8 bits. we use it as EVM memory in our SMV code.
- **storage:** This variable is an array of unsigned word values that we use it as EVM storage in our SMV code.

In the main module, we also have an instance of the processor module, memory module, storage module, and gas module.

One of the essential steps in developing the SMV code is implementing the flow of execution of the smart contract's operations. To do that, we firstly disassemble the EVM code of the smart contract. Then we start the program's execution from the first operation that appears at the first line of the assembly code. Then, as long as there is no JUMP or JUMPI operation, we follow the operation sequence in the assembly code. Of course, for some operations, we require more than one state in the SMV code. Hence, we add one dummy operation after those EVM operations and then we follow the sequence. We will describe the flow of execution of the more complex operations in detail when explaining their implementation details.

### 4.2.2 Processor module

As its name shows, we have designed this module to process EVM operations. Every operation that needs some items of the stack or needs to push one or more new items into the stack will be processed in this module. The inputs of this module are **operationName**, **operationArray**, **stack**, **stack_head**, **memory**, **storage**, and **gas**. Using the value of operationName, this module identifies what operation it needs to process. Using operationArray, inputs of the operation are detected. As some operations may also demand to read some values from memory and storage, We also pass memory and storage variables as inputs to this module. In this module, we only read from memory and storage. If an operation requires writing into the memory or storage module, we process its memory and storage writes in the memory and storage module accordingly.

### 4.2.3 Gas module

As we know, the execution of each operation has a specific cost. Therefore we must decrease the value of gas after the execution of each operation. We handle it inside gas_module.

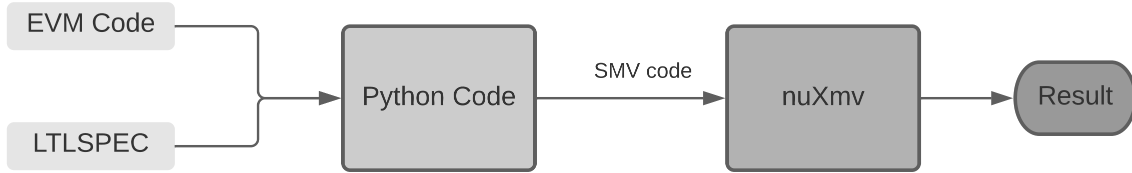


Figure 4.1: The structure of the designed smart contracts model checker

operationName and gas are two inputs of this module. The value of the operationName variable indicates what operation is executing, and the gas variable value is the remainder gas. The module will reduce the gas by a specific amount based on the currently executing operation.

### 4.2.4 Memory module

Some operations like *mstore*, *mstore8*, *codecopy* and etc. will change the memory by writing a new value in it. We will manage memory changes of an operation in memory_module. The inputs of this module are **operationName**, **operationArray**, **stack**, **stack_head**, and **memory**. Using operationName variable we detect what operation is executing and using stack and stack_head we specify what address of the memory will be accessed.

### 4.2.5 Storage module

Some operations like *sstore* will change the storage by writing a new value in it. We will manage storage changes of an operation in storage_module. The inputs of this module are **operationName**, **operationArray**, **stack**, **stack_head**, and **storage**. Using operationName variable we detect what operation is executing and using stack and stack_head we specify what address of the storage will be accessed.

## 4.3 Model Checker Structure

We have designed and implemented the reduction using the Python programming language. Figure 4.1 indicates the structure of the developed model checker. As it can be seen, the reduction is implemented in a Python module, and nuXmv evaluates the correctness of the provided LTLSPEC using the SMV code that the python module has generated. In Section 4.3.1 we will provide more details of the python code and its submodules.

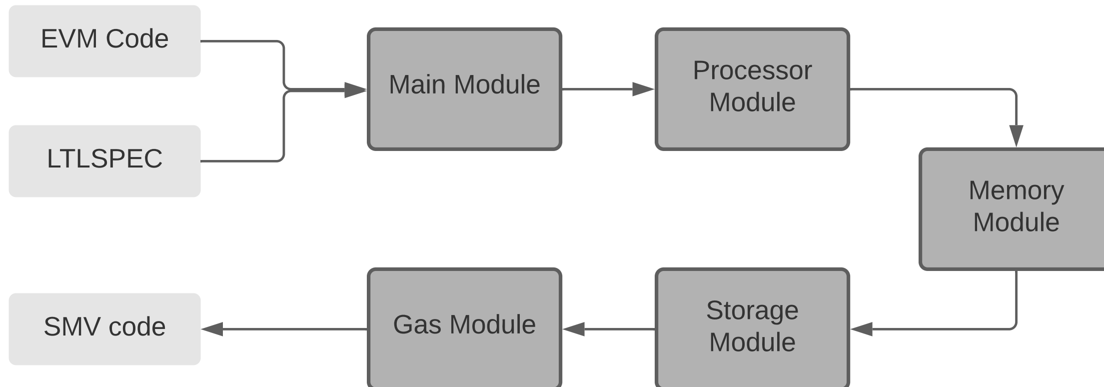


Figure 4.2: The structure of the designed Python Code

### 4.3.1 Python Code Architecture

The purpose of the python code is to reduce the EVM code into SMV code. Figure 4.2 provides the flow of creation of the SMV code. We have a global variable in the form of a python list that in each stage of the reduction we append more SMV code to it. Finally, the content of the variable is written into a file which is the SMV file. This SMV file is provided as an input to nuXmv. Creating the SMV code starts with producing the Main module of the SMV code and is finished with building the Gas module. The subsequent sections provide more details of the implementation of each of the submodules.

#### Main Module

Before anything else, the EVM code is disassembled using an Ethereum EVM code disassembler. Although it is also possible to use the EVM code itself, by disassembling the EVM code extraction of the operations' name and inputs could be more straightforward. In the main module, the following sequence has been implemented:

1. Creating the global variables such as Gas, stack, stack_head, memory, storage, etc.
2. Instantiating the submodules such as Processor, Gas, Memory, and Storage modules.
3. Initializing the defined variables in the Main module.
4. Developing the flow of execution of the EVM code operations. In this section, we determine what the next operation that should be executed is. Most of the operations are executed sequentially, but we should consider the branches in the EVM code.

### Processor Module

As its name indicates, this module is used to implement the SMV code which is responsible for processing each of the EVM operations. In the Processor module, the following sequence has been implemented:

1. Creating the required extra variables. For the execution of some of the EVM operations, we need to add some extra variables. We use these variables to calculate the final value(s) that the operations must return. Using the python code, we define these variables at the beginning of the Processor module.
2. Initializing the defined variables in the Processor module. The initial value of the variables are important for the execution of the corresponding operations.
3. Determining the next value of each of the defined variables. Based on the operation that is executing and the state machine's current state, the next value of this variables will be assigned to them.
4. Determining the result of the execution of each operation. In this part of the python code, using the operation's defined variables and input values, its output(s) is calculated and pushed to the stack.
5. Applying the stack changes. Executing each of the operations may require some items from the top of the stack and push some results into the stack. In this part of the python code, we determine how the stack_head variable must be updated during the execution of the SMV code.

### Memory Module

The execution of some of the EVM operations may require accessing the memory. In the Memory module, the following sequence has been implemented:

1. Creating the required extra variables. For the execution of some of the EVM operations, we need to add some extra variables. We use these variables to calculate the final value(s) that the operations must return. Using the python code, we define these variables at the beginning of the Memory module.
2. Initializing the defined variable in the Memory module. The initial value of the variable is important for the execution of the corresponding operations.
3. Determining the result of execution of operations that need to access the Memory for their execution. In this part of the python code, we determine which memory locations the executing operation require to read and which memory locations it needs to store the result of its execution.

### Storage Module

The execution of some of the EVM operations may require accessing the storage. In the storage module, the following sequence has been implemented:

1. Creating the required extra variables. For the execution of some of the EVM operations, we need to add some extra variables. We use these variables to calculate the final value(s) that the operations must return. Using the python code, we define these variables at the beginning of the Storage module.

2. Initializing the defined variables in the Storage module. The initial value of the variables is important for the execution of the corresponding operations.

3. Determining the result of execution of operations that need to access the storage for their execution. In this part of the python code, we determine which storage locations the executing operation requires to read and which storage locations it needs to store the result of its execution.

### Gas Module

Based on the cost that the executing operation has, the variable Gas must be updated. In this part of the python code, we created the SMV code that is responsible for updating the next value of the gas variable after the execution of the operation.

## 4.4 EVM Operations in SMV

We designed and implemented all of the EVM operations necessary to cover the curated dataset of vulnerable smart contracts, including 132 EVM instructions. In this section, we will explain the detail of the implementation some of the EVM operations in SMV. In section 4.4.1 we will clarify the detail of the implementation of the operations supported by the SVM language. There are also some operations in the EVM that there is no equivalent operation for them in the SMV language. Therefore, we need to implement them from scratch based on the available operations in SMV. We will explain those operations in section 4.4.2. Table 4.1 is a reference to details of the implementation of each EVM operation or EVM operation group. We have not included a few operations in the thesis because they have been implemented much easier than the described operations, or in our tool, we needed to jump over the operations and only update the stack head and the available gas.

| Operation Group | Reduction |
|---|---|
| ADD, MUL, SUB, DIV, MOD | 1 |
| LT, GT, EQ, AND, OR, XOR, NOT | 2 |
| SDIV, SMOD | 3 |
| EXP | 4 |
| ADDMOD, MULMOD | 5 |
| SIGNEXTEND | 6 |
| BYTE | 7 |
| ADDRESS, ORIGIN | 8 |
| CALLDATACOPY, RETURNDATACOPY | 9 |
| MLOAD, MSTORE | 10 |
| SLOAD, SSTORE | 11 |
| JUMPI, JUMP, JUMPDEST | 12 |
| PUSH, DUP, SWAP | 13 |
| CALL, DELEGATECALL | 14 |
| Closure Operations | 15 |
| SHA3 | 16 |

Table 4.1: A reference table to details of the implementation of each EVM operation or EVM operation group

### 4.4.1 Supported Operations by SMV

This section demonstrates the detail of the implementation of some of the EVM operations that there is an equivalent operator for them in SMV language.

#### Arithmetic Operations

Some of the EVM arithmetic operations like *ADD*, *MUL*, *SUB*, *DIV*, and *MOD* have an equivalent operation in SMV language. Therefore, for their implementation in SMV, we only need to provide the operands and then apply the operation to them. For example, "0x6005600401" is an EVM code that pushes two values into the stack and then pops those values out of the stack and, after calculating the sum of these two values, stores the result into to stack. The equivalent assembly code of this EVM code is as follows:

```
1 [0] 60 PUSH1 5:0x05
2 [2] 60 PUSH1 4:0x04
3 [4] 01 ADD
```

The flow of execution of this EVM code has been implemented in SMV as follows:

```
next(operationName) :=  case
    operationName = begin : PUSH1_0;
    operationName = PUSH1_0 : PUSH1_2;
    operationName = PUSH1_2 : ADD_4;
    operationName = ADD_4 : end;
    TRUE : operationName;
esac;
```

As it can be seen, the execution of the EVM code starts with "begin," which is a dummy state or a dummy operation, and after that PUSH1_0, PUSH1_1, ADD_4, and end operations will be executed, respectively.

Operations of this EVM code only need the stack, and there is no access to memory and storage. Thus based on the currently executing operation that is specified by operationName variable, We need to specify the next value of each item of the stack. The following code indicates how the EVM code alters the value of the $i$th item of the stack.

```
next(stack[i]) := case
    operationName = PUSH1_0 & stack_head = 1 : operationArray[0];
    operationName = PUSH1_2 & stack_head = 1 : operationArray[2];
    operationName = ADD_4 & stack_head = 3 : (stack[1] + stack[2]);
    TRUE : stack[i];
esac;
```

Where "i" could be from zero to the size of the stack minus one. In the SMV code, we do not have "i" because the index of arrays must be a constant value in the SMV language. Therefore, we duplicate this code for each item of the stack. As shown in the code, based on some conditions, the next value of the $i$th item of the stack is specified. For instance, if the executing operation is ADD_4 and the stack head is on the fourth item of the stack, then the next value of this item of the stack is some of the second and third items of the stack. In other words, we pop two items from the stack and push the sum of them into the stack.
Based on the operation that is executing, the stack head also needs to be updated. The following code specifies the next value of stack head after execution of each of the operations of the mentioned EVM code.

```
next(stack_head) := case
    operationName = PUSH1_0 : max(0, stack_head + 1) mod 10;
    operationName = PUSH1_2 : max(0, stack_head + 1) mod 10;
    operationName = ADD_4 : max(0, stack_head - 1) mod 10;
    TRUE : stack_head;
esac;
```

As it can be seen, execution of the ADD operation decreases the stack_head value by one, and that is because it pups two items out of the stack and pushes only one item into it. This part of the code exists in the processor module.

The last important part of turning this EVM code into SMV code is updating the gas value. The following code shows the generated gas module for the mentioned EVM code.

```
MODULE gas_module(operationName, gas)
VAR
ASSIGN
next(gas) := case
    operationName = PUSH1_0 : (gas - 0ud16_3);
    operationName = PUSH1_2 : (gas - 0ud16_3);
    operationName = ADD_4 : (gas - 0ud16_3);
    TRUE : gas;
esac;
```

According to the Ethereum yellow paper cost of ADD operation is three. Therefore, it can be seen in the gas module that if the executing operation is ADD then the gas value will be decreased by three.

#### Boolean Operations

Boolean operations like *LT*, *GT*, *EQ*, *AND*, *OR*, *XOR* and *NOT* also have an equivalent operation in SMV language. Therefore, for their implementation in SMV, again we only need to provide the operands and then apply the operation to them. Detail of implementation is the same as what was described in previous section.

### 4.4.2 Not Supported Operations by SMV

#### SDIV and SMOD

These two operations are signed operations. All of our variables in our SMV design are unsigned. Hence we must manage that in our implementation of these two operations. To do that, we first convert the operands of these operations from unsigned value to signed value, and then we calculate the two's complement of the result and push it into the stack. The following SMV code indicates the detail of the implementation of the SDIV operation. This SMV code shows that whenever the executing operation is SDIV, and the stack head is three, the operation result will be pushed in the second item of the stack. There is also a condition that makes sure that the division by zero does not happen. In line three of the code, it can be seen that we do not need to calculate the two's complement whenever the operands are positive. Otherwise, after calculating the division, we calculate two's complement of the result and then push it into the stack.

```
next(stack[1]) := case
    operationName = SDIV_4 & stack_head = 3 & stack[1] = 0ud8_0 : 0ud8_0;
```

```
    operationName = SDIV_4 & stack_head = 3 & stack[1] != 0ud8_0 & !(stack[2] < 0ud8_0 | stack[1] < 0ud8_0) : unsigned(signed(stack[2]) / signed(stack[1]));
    operationName = SDIV_4 & stack_head = 3 & stack[1] != 0ud8_0 & (stack[2] < 0ud8_0 | stack[1] < 0ud8_0) : unsigned(signed(stack[2]) / signed(stack[1])) xor 0ud8_255 + 0ud8_1;
    TRUE : stack[1];
esac;
```

### EXP

We realize the following recurrence in SMV.

$$x^y = \begin{cases} 1 & \text{if } y = 0 \\ (x^2)^{\lfloor y/2 \rfloor} & \text{if } y > 0 \text{ and } y \text{ is even} \\ x \cdot (x^2)^{\lfloor y/2 \rfloor} & \text{otherwise} \end{cases}$$

The way we do it is to transition state-by-state with three variables, call them $v_0, v_1, v_2$. The semantics are that in any state, $x^y = v_1 \cdot (v_2)^{v_0}$. Our initial state for $\langle v_0, v_1, v_2 \rangle$ is $\langle y, 1, x \rangle$. And we have our result when $\langle v_0, v_1, v_2 \rangle = \langle 0, x^y, \cdot \rangle$. More specifically, we have our result in $v_1$ if and only if $v_0 = 0$.

To specify the state-transition, suppose we denote the current state as $\langle v_0, v_1, v_2 \rangle$, and the next state as $\langle v'_0, v'_1, v'_2 \rangle$. Then:

$$\langle v'_0, v'_1, v'_2 \rangle = \begin{cases} \langle v_0, v_1, v_2 \rangle & \text{if } v_0 = 0 \\ \langle v_0/2, v_1, v_2^2 \rangle & \text{if } v_0 > 0 \text{ and } v_0 \text{ is even} \\ \langle \lfloor v_0/2 \rfloor, v_1 \cdot v_2, v_2^2 \rangle & \text{otherwise} \end{cases}$$

The following code indicates the sequence of execution of an EVM code operations. As it can be seen there are two push operations that push the two operands of the EXP operation into the stack and after that we have the EXP operation itself. As we need to use some temporary variables in execution of EXP operation we have added a dummpy operation after each EXP operation. As it can be seen there is a condition with the dummy operations that indicates if the EXP operation has been completed or not.

```
next(operationName) :=  case
    operationName = begin : PUSH1_0;
    operationName = PUSH1_0 : PUSH1_2;
    operationName = PUSH1_2 : EXP_4;
    operationName = EXP_4 : EXP_4_DUMMY;
    operationName = EXP_4_DUMMY & EXP_4_work[0] = 0ud16_0 : end;
    operationName = end : begin;
```

```
8       TRUE : operationName;
9   esac;
```

As explain before, for implementation of the EXP operation we use EXP_4_work and array that contains three unsigned word items. The following code indicates the next value of each item of the array. As it can be seen we first load the initial values into the array and then when operationName = EXP_4_DUMMY we start calculating the EXP operation. We have our result in the third item of the array if and only if the value of the first item is zero.

```
1   next(EXP_4_work[0]) := case /-- running exponent --/
2       operationName = EXP_4 & stack_head = 0 : stack[14];
3       operationName = EXP_4 & stack_head = 1 : stack[15];
4       ...
5       operationName = EXP_4 & stack_head = 14 : stack[12];
6       operationName = EXP_4 & stack_head = 15 : stack[13];
7       operationName = EXP_4_DUMMY & EXP_4_work[0] > 0ud16_0 : EXP_4_work[0]/0ud16_2;
8       TRUE : EXP_4_work[0];
9   esac;
10
11  next(EXP_4_work[1]) := case /-- squares, initialized to base of exponentiation --/
12      operationName = EXP_4 & stack_head = 0 : stack[15];
13      operationName = EXP_4 & stack_head = 1 : stack[0];
14      ...
15      operationName = EXP_4 & stack_head = 14 : stack[13];
16      operationName = EXP_4 & stack_head = 15 : stack[14];
17      operationName = EXP_4_DUMMY & EXP_4_work[0] > 0ud16_0 : EXP_4_work[1] * EXP_4_work[1];
18      TRUE : EXP_4_work[1];
19  esac;
20
21  next(EXP_4_work[2]) := case
22      operationName = EXP_4_DUMMY & EXP_4_work[0] mod 0ud16_2 = 0ud16_1 : EXP_4_work[2] * ↩
            EXP_4_work[1];
23      TRUE : EXP_4_work[2];
24  esac;
```

At the end of the execution of the operation the result must be pushed into the stack. The following SMV code indicates that whenever the EXP operation has been completed and stack_head is 2 the result of the operation must be pushed in the first location of the stack. The same code will be repeated for all of the stack locations.

```
1   ASSIGN
2   next(stack[0]) := case
3       operationName = EXP_4_DUMMY & EXP_4_work[0] = 0ud16_0 & stack_head = 2 : EXP_4_work↩
            [2];
4       TRUE : stack[0];
5   esac;
```

**ADDMOD and MULMOD**

As their name shows these operation are a sequence of two operations. ADDMOD is a sequence of ADD and then MOD operation and MULMOD is a sequence of MUL and then MOD operation. Therefore these two operations have three operands. The following SMV code indicates the detail of implementation of MULMOD operation. As it can be seen whenever the executing operation is MULMOD and stack head is three then the result of this operation will be pushed in the fist item of stack. As division by zero may happen in these two operations there are a third condition to avoid division by zero.

```
1  next(stack[0]) := case
2      operationName = MULMOD_6 & stack_head = 3 & stack[0] != 0ud8_0 : (stack[1] * stack[2])↩
           mod stack[0];
3      operationName = MULMOD_6 & stack_head = 3 & stack[0] = 0ud8_0 : 0ud8_0;
4      TRUE : stack[0];
5  esac;
```

**SIGNEXTEND**

This operation has two operands and after execution, the result will be sign extension of one of the operands. One of the operands specifies the bit that we need to use for our signed extension. It does not contain the location of the bit in the other operand instead it points at the bye index that the sign bit contains there. For example if its value is two it means that we must use the most significant bit of the third byte of the other operand. As it can be seen If the stack head is 3 we must use the third item of stack to specify the sign bit of the second item of stack and use that sign bit to extend the second value. This code is repeated for all stack items.
In EVM architecture, length of each stack item is 256 bits but in the following example we considered its length 16 bits. As it can be seen in the code, when the value of the third item of stack is 0 then we have to use the most significant bit of the first byte of the second item of the stack as the sign and if it is 1 then the most significant bit of the second byte of the second item of the stack is the sign. As here the length of each stack item is 16 it only has two bytes hence the third item of the stack in this example could only have the value of 0 or 1. After detecting the sign bit we check weather its value is zero or one. If it is zero then we must use zero for the extension. To do that we use a bitwise AND operation. For instance, in second line of the example we apply bitwise "AND" operation on the second item of the stack and decimal number of 255 that this operation keeps the value the first 8 least significant bits of the second item of stack and set the others to zero. If the sign value is 1 we use bitwise OR operation and we generate the number in a way that sets all bits after the sign bit to one. Bye knowing the sign bit location and length of the stack items generating the numbers is feasible.

```
1 next(stack[1]) := case
2     operationName = SIGNEXTEND_7 & stack_head = 3 & stack[2] = 0ud16_0 & stack[1][7:7] = 0↩
         ud1_0: stack[1] & 0ud16_255;
3     operationName = SIGNEXTEND_7 & stack_head = 3 & stack[2] = 0ud16_0 & stack[1][7:7] = 0↩
         ud1_1: stack[1] | 0ud16_65280;
4     operationName = SIGNEXTEND_7 & stack_head = 3 & stack[2] = 0ud16_1 & stack[1][15:15] =↩
          0ud1_0: stack[1] & 0ud16_65535;
5     operationName = SIGNEXTEND_7 & stack_head = 3 & stack[2] = 0ud16_1 & stack[1][15:15] =↩
          0ud1_1: stack[1] | 0ud16_0;
6     TRUE : stack[1];
7 esac;
```

### BYTE

This operation extracts one byte from a word. It has two operands that one of the operands specifies a byte location that needs to be extracted from the other operand. The following code indicates the detail of the implementation of this operation. As it can be seen, whenever the executing operation is a BYTE operation, and the stack head is three, the result of this operation will be pushed into the second place of the stack. This code is repeated for all stack locations. If the value of the third location of the stack is zero, then we select the first byte of the second location of memory. If the third location of the stack is one, then we extract the 8 bits of data from the second item of stack starting from bit number 8 till bit number 15. As we cannot store an item into the stack that its length is less than the determined word length of stack items, we use the resize function of SMV to change the size of the result from 8 bits into the length of stack items. This function only changes the size of the result, not its value.

```
1 next(stack[1]) := case
2     operationName = BYTE_4 & stack_head = 3 & stack[2] = 0ud16_0 : resize(stack[1][7:0], ↩
         16);
3     operationName = BYTE_4 & stack_head = 3 & stack[2] = 0ud16_1 : resize(stack[1][15:8], ↩
         16);
4     TRUE : stack[1];
5 esac;
```

### ADDRESS and ORIGIN

These two operations do not have any inputs, and they only push one item into the stack that they receive it from outside of the contract. Therefore the item that they push into the stack could have any value. Whenever we have one of these operations in the contract, we define a variable in our SMV code, and we let NuSMV decide about its value. It means this variable could have any value. To execute each of these operations, we only push the variable that we have created for it into the stack. We exactly follow the same approach for CALLER, CALLVALUE, CALLDATASIZE, and RETURNDATASIZE. Actually, for each operation that its output value comes from out of the executing contract, we define a variable that

NuSMV decides about its value.
The following SMV code demonstrates the details of the implementation of ADDRESS operation. As it can be seen, whenever the executing operation is ADDRESS, and the stack head is one, then the "ADDRESS_4_return_value" variable will be pushed on top of the stack. This code will be repeated for all stack items.

```
1 next(stack[1]) := case
2     operationName = ADDRESS_4 & stack_head = 1 : ADDRESS_4_return_value;
3     TRUE : stack[1];
4 esac;
```

The following code indicates the definition of "ADDRESS_4_return_value" variable in SMV. It has been defined in the processor module. As we do not set its next value in our SMV code, NuSMV will try all possible values for this variable.

```
1 ADDRESS_4_return_value : unsigned word[16];
```

We use the same design for CALLDATALOAD and BALANCE operations. Although these two operations have one input and one output, the output is provided outside the contract. Therefore it could have any values. Hence we use the same design. There are also some blockchain context operations like BLOCKHASH, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, and GASLIMIT that we follow the same design strategy for them.

### CALLDATACOPY and RETURNDATACOPY

These two operations pop three items from the stack and do not push any item into it. They write some values into the memory. One of the inputs of these operations specifies the amount of data that must be copied into the memory. One of them determines the location in memory that we must start copying the data into it from that point. The third input shows the point from input data that we start coping from it. In our tool and based on our design strategy, we do not use the third input.
As the input data could be anything, we have to let NuSMV decide about the value of data that needs to be written into the memory. To do that, we define a variable in the memory module for each row of the memory. As we do not determine the next value of the variables, NuSMV will assign any values. Based on the starting point in memory and the size of the data that needs to be copied, we assign each of these variables to their corresponding row of memory. The data could have any size but based on our design; it could not have a size greater than the memory size. Hence as we increase the memory size, the generated SMV code would be larger. The following code indicates the detail of the implementation of CALLDATACOPY operation in the SMV language. This code only shows the next value of the second location of the memory, but we exactly repeat the same code for all memory locations. We use

stack_head variable to detect the inputs of these operations. As we mentioned before, there are three inputs that two of them specify the start point in memory and the size of the data that should be copied into it. By knowing the value of these two inputs, we can determine if a new value should be assigned to the memory location or not. Suppose the memory location is one of the locations that we need to copy a new value to it. In that case, we assign it the corresponding variable that we have defined at the beginning of the memory module.

```
next(memory[1]) := case
    operationName = CALLDATACOPY_6 & stack_head = 0 & (stack[9] > 0ud16_1 | stack[9] + stack[7] > 0ud16_16 | stack[9] + stack[7] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 0 & !(stack[9] > 0ud16_1 | stack[9] + stack[7] > 0ud16_16 | stack[9] + stack[7] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 3 & (stack[2] > 0ud16_1 | stack[2] + stack[0] > 0ud16_16 | stack[2] + stack[0] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 3 & !(stack[2] > 0ud16_1 | stack[2] + stack[0] > 0ud16_16 | stack[2] + stack[0] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 4 & (stack[3] > 0ud16_1 | stack[3] + stack[1] > 0ud16_16 | stack[3] + stack[1] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 4 & !(stack[3] > 0ud16_1 | stack[3] + stack[1] > 0ud16_16 | stack[3] + stack[1] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 5 & (stack[4] > 0ud16_1 | stack[4] + stack[2] > 0ud16_16 | stack[4] + stack[2] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 5 & !(stack[4] > 0ud16_1 | stack[4] + stack[2] > 0ud16_16 | stack[4] + stack[2] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 6 & (stack[5] > 0ud16_1 | stack[5] + stack[3] > 0ud16_16 | stack[5] + stack[3] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 6 & !(stack[5] > 0ud16_1 | stack[5] + stack[3] > 0ud16_16 | stack[5] + stack[3] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 7 & (stack[6] > 0ud16_1 | stack[6] + stack[4] > 0ud16_16 | stack[6] + stack[4] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 7 & !(stack[6] > 0ud16_1 | stack[6] + stack[4] > 0ud16_16 | stack[6] + stack[4] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 8 & (stack[7] > 0ud16_1 | stack[7] + stack[5] > 0ud16_16 | stack[7] + stack[5] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 8 & !(stack[7] > 0ud16_1 | stack[7] + stack[5] > 0ud16_16 | stack[7] + stack[5] <= 0ud16_1) : memory_line_1_potential_variable;
    operationName = CALLDATACOPY_6 & stack_head = 9 & (stack[8] > 0ud16_1 | stack[8] + stack[6] > 0ud16_16 | stack[8] + stack[6] <= 0ud16_1): memory[1];
    operationName = CALLDATACOPY_6 & stack_head = 9 & !(stack[8] > 0ud16_1 | stack[8] + stack[6] > 0ud16_16 | stack[8] + stack[6] <= 0ud16_1) : memory_line_1_potential_variable;
    TRUE : memory[1];
esac;
```

The variables that we use in the implementation of these two operations are defined in the memory module. As we do not specify their next value, NuSMV could assign any value to them. The following code indicates the definition of these variables in the memory module. In these example size of the memory is 16; therefore, we have 16 corresponding variables.

```
memory_line_0_potential_variable : unsigned word[8];
memory_line_1_potential_variable : unsigned word[8];
    .
    .
    .
memory_line_14_potential_variable : unsigned word[8];
memory_line_15_potential_variable : unsigned word[8];
```

### MLOAD and MSTORE

These two EVM operations are used for reading/writing one word of data from/to the memory. The challenging part in implementing these operations is that each row of memory is eight bits, but the size of each stack item is larger than eight bits. Therefore, these operations access more than one location of the memory. The following code indicates the detail of the implementation of the MSTORE operation. This operation has two inputs, one of which determines the memory location that we should start storing the word of data and the second one the word of data itself. The following code indicates the next value of the second row of the memory. This could is repeated in the SMV code for all of the memory locations. As shown in the code, when the executing operation is MSTORE, we use the stack head to access the operation's operands in the stack. Then we have some other conditions based on the value of the operand that points at a memory location. In the second row of the code, we have a condition to check if the operation accesses this memory location or not. If not, it keeps its current value. Otherwise, we have to access this memory location, but we need to determine which byte of the data word must be written into the memory location. We do it by adding another condition that can be seen in lines three and four of the code. Then we pick the specified byte from the word of data and store it into the memory location. The implementation of the MLOAD operation in SMV follows the same design plan.

```
next(memory[1]) := case
    operationName = MSTORE_4 & stack_head = 0 & stack[4] > 0ud16_1 : memory[1];
    operationName = MSTORE_4 & stack_head = 0 & !(stack[4] > 0ud16_1) & stack[4] - 0ud16_1
        = 0ud16_0 : stack[3][7:0];
    operationName = MSTORE_4 & stack_head = 0 & !(stack[4] > 0ud16_1) & stack[4] - 0ud16_1
        = 0ud16_1 : stack[3][15:8];
    operationName = MSTORE_4 & stack_head = 2 & stack[1] > 0ud16_1 : memory[1];
    operationName = MSTORE_4 & stack_head = 2 & !(stack[1] > 0ud16_1) & stack[1] - 0ud16_1
        = 0ud16_0 : stack[0][7:0];
    operationName = MSTORE_4 & stack_head = 2 & !(stack[1] > 0ud16_1) & stack[1] - 0ud16_1
        = 0ud16_1 : stack[0][15:8];
    operationName = MSTORE_4 & stack_head = 3 & stack[2] > 0ud16_1 : memory[1];
    operationName = MSTORE_4 & stack_head = 3 & !(stack[2] > 0ud16_1) & stack[2] - 0ud16_1
        = 0ud16_0 : stack[1][7:0];
    operationName = MSTORE_4 & stack_head = 3 & !(stack[2] > 0ud16_1) & stack[2] - 0ud16_1
        = 0ud16_1 : stack[1][15:8];
    operationName = MSTORE_4 & stack_head = 4 & stack[3] > 0ud16_1 : memory[1];
    operationName = MSTORE_4 & stack_head = 4 & !(stack[3] > 0ud16_1) & stack[3] - 0ud16_1
        = 0ud16_0 : stack[2][7:0];
    operationName = MSTORE_4 & stack_head = 4 & !(stack[3] > 0ud16_1) & stack[3] - 0ud16_1
        = 0ud16_1 : stack[2][15:8];
```

```
14      TRUE : memory[1];
15  esac;
```

There is another operation that also accesses the memory. It is MSTORE8. As its name shows, it is used to store eight bits of data into the memory. Implementation of this operation is more straightforward than the two previous ones. As it only stores one byte into the memory, we do not need to check which byte of the data word should be stored in which memory location. We only need to store the eight least significant bits of the data word in the memory location that the other operand points at it.

### SLOAD and SSTORE

These two operations are used to access the EVM storage. SLOAD is used to load one word of data from the storage, and SSTORE is used to store one word of data into it. As each row of the storage has the same size as each stack item, the design and implementation strategy of these operations in SMV is the same as MSTORE8 operations. There is only one difference instead of storing or loading the eight least significant bits of the data, we store or load the whole data word.

### JUMPI, JUMP, and JUMPDEST

These three operations are used to manage the branches in the code. JUMPI and JUMP are conditional and unconditional jumps accordingly, and JUMPDEST specifies one potential destination for JUMPI and JUMP operations. To implement the JUMPI operation, we need to add one dummy operation to the SMV code. The reason for having this dummy operation, we can call it a dummy state in the state machine, is that there are some variables whose value must be updated before the execution of the JUMPI operation. Therefore, when the executing operation in the SMV code is a JUMPI, we update the jump condition variables. Actually, when the executing operation is the added dummy operation, we execute the JUMPI operation. As we said, JUMPI is a conditional jump. Therefore, the jump could only occur when its condition is satisfied. For each JUMPI operation in the contract, we generate a boolean variable in the processor module of the SMV code and update its value based on the JUMPI operands. The following SMV code indicates the next value of the boolean condition variable of a JUMPI operation in the SMV code. As it can be seen, whenever the executing operation is JUMPI_83, we use the stack head to identify the JUMPI operands that correspond to the jump condition. If the value of the operand corresponding to the jump condition is not zero, we set the next value of JUMPI_83_condition to TURE. We use the variable in the next dummy operation to complete the implementation of the JUMPI operation.

```
next(JUMPI_83_condition) := case
    operationName = JUMPI_83 & stack_head = 0 & stack[8] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 2 & stack[0] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 3 & stack[1] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 4 & stack[2] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 5 & stack[3] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 6 & stack[4] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 7 & stack[5] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 8 & stack[6] != 0ud16_0 : TRUE;
    operationName = JUMPI_83 & stack_head = 9 & stack[7] != 0ud16_0 : TRUE;
    TRUE : FALSE;
esac;
```

By setting the next value of JUMPI_83_condition variable, we can recognize if the JUMP should occur or not. However, we cannot identify what the destination of this jump is. Here is where we use the second operand of the JUMPI operation. Suppose the value of the condition variable of the JUMPI operation is TRUE. In that case, we can use the other operand's value to identify the jump destination. The jump destination could be one of the JUMPDEST operations in the code. For each JUMPDEST operation in the code, we generate a boolean variable in the processor module of the SMV code that its value will be set to TRUE if it is the destination of a JUMP or JUMPI operation. The following code indicates the details of the implementation of the next value of one of the variables in the processor module of the SMV code. As shown in line three of the code, whenever the executing operation is JUMPI_11 and stack_head is two, then the value of the second item of the stack determines the destination. If its value is 73, then the destination of the JUMPI woulb be JUMPDEST_73.

```
next(jump_destination_is_73) := case
    operationName = JUMPI_11 & stack_head = 0 & stack[9] = 0ud16_73 : TRUE;
    operationName = JUMPI_11 & stack_head = 2 & stack[1] = 0ud16_73 : TRUE;
    ...
    operationName = JUMP_171 & stack_head = 8 & stack[7] = 0ud16_73 : TRUE;
    operationName = JUMP_171 & stack_head = 9 & stack[8] = 0ud16_73 : TRUE;
    TRUE : FALSE;
esac;
```

The following code indicates the details of our design and implementation for JUMPI operation. As it can be, there are two conditions in each line of the code. One of them indicates if the jump should occur or not, and the other one specifies the jump destination. JUMP operation only has one difference from the JUMPI operation. As JUMP is unconditional, we do not need a variable to check if the jump should happen.

```
    operationName = JUMPI_83 : JUMPI_83_DUMMY;
    operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_73 : JUMPDEST_73;
    operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_78 : JUMPDEST_78;
    operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_88 : JUMPDEST_88;
    operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_94 : JUMPDEST_94;
    operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
```

```
        jump_destination_is_116 : JUMPDEST_116;
7     operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_126 : JUMPDEST_126;
8     operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_146 : JUMPDEST_146;
9     operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_148 : JUMPDEST_148;
10    operationName = JUMPI_83_DUMMY & proc_module.JUMPI_83_condition & proc_module.↩
        jump_destination_is_154 : JUMPDEST_154;
11    operationName = JUMPI_83_DUMMY & !proc_module.JUMPI_83_condition : PUSH1_84;
12    operationName = PUSH1_84 : DUP1_86;
```

### PUSH, DUP, and SWAP

PUSH operation pushes new items of different sizes into the stack. The size of the item that is pushed into the stack could be from 1 to 32 bytes. Therefore, EVM has 32 push operations. These 32 PUSH operations are PUSH1 to PUSH32. The number indicates the size of the item that is being pushed into the stack. DUP operation duplicates an item that already exists in the stack. This operation can duplicate each of the 16 items on top of the stack. Hence, EVM has 16 DUP operations that these DUP operations are DUP1 to DUP16. The number in front of the operation name points at the item's location in the stack that must be copied and then pushed on top of the stack. SWAP operation swaps the item on top of the stack with one of the 16 items located under it. Therefore, EVM has 16 SWAP operations that are SWAP1 to SWAP16. The number in front of the operation name indicates the location of stack item that must be swapped with the item on top of the stack. While PUSH and DUP operations have a straightforward implementation in SMV, the implementation of SWAP operations needs more effort.
The following SMV code indicates the detail of the implementation of a swap operation in SMV. In this code, our contract only includes one SWAP operation that is SWAP3. As it can be seen, whenever the executing operation is SWAP3, if $stack_head = 11$ then the next value of stack[10] must be stack[7] and if the $stack_head = 14$ then the next value of stack[10] must be stack[13]. These are the only two possible situations that could occur when the executing operation is SWAP3, and we are finding the next value of stack[10]. The same code must be repeated for all of the stack items. Depending on the number of different swap operations and their type in the contract, the size of the code may increase.

```
1  next(stack[10]) := case
2      operationName = SWAP3_22 & stack_head = 11 : stack[7];
3      operationName = SWAP3_22 & stack_head = 14 : stack[13];
4      TRUE : stack[10];
5  esac;
```

### CALL and DELEGATECALL

The implementation of these two operations in SMV language is almost the same. Both of these operations push only one item on top of the stack but CALL has 7 inputs and DELEGATECALL has 6 inputs. In addition to pushing one item into the stack they write into some of the memory locations. The following code indicates how we manage the next value of stack[10] after executing CALL operation. For each CALL or DELEGATECALL operation we generate a variable corresponding to its return value in the processor module. As we do not define the next value of the variable in the SMV code, NuSMV will decide about its next value.

```
1 next(stack[11]) := case
2     operationName = CALL_14 & stack_head = 2 : CALL_14_return_value;
3     TRUE : stack[11];
4 esac;
```

As we said, these operations also alter some of the memory locations. The following code indicates the next value of memory[2] after executing the CALL operation. As it can be seen, whenever the executing operation is a CALL operation, using the stack_head value, we identify the operands of this operation. Using these operands, we recognize that if the memory location is in the range of memory locations, the call value must write a new value. We set its next value to a variable we have defined for this memory location in the memory module. As the next value of the variable has not been specified in the SMV code, NuSMV will decide about its next value. Hence it could have any values.

```
1 next(memory[2]) := case
2     operationName = CALL_14 & stack_head = 0 & (stack[10] > 0ud16_2 | stack[10] + stack[9]↩
       > 0ud16_16 | stack[10] + stack[9] <= 0ud16_2): memory[2];
3     operationName = CALL_14 & stack_head = 0 & !(stack[10] > 0ud16_2 | stack[10] + stack↩
      [9] > 0ud16_16 | stack[10] + stack[9] <= 0ud16_2) : ↩
      memory_line_2_potential_variable;
4     ...
5     operationName = CALL_14 & stack_head = 15 & (stack[9] > 0ud16_2 | stack[9] + stack[8] ↩
      > 0ud16_16 | stack[9] + stack[8] <= 0ud16_2): memory[2];
6     operationName = CALL_14 & stack_head = 15 & !(stack[9] > 0ud16_2 | stack[9] + stack[8]↩
       > 0ud16_16 | stack[9] + stack[8] <= 0ud16_2) : memory_line_2_potential_variable;
7     TRUE : memory[2];
8 esac;
```

### Closure Operations

There are some operations like SELFDESTRUCT, STOP, INVALID, and REVERT in the EVM. In the SMV code, whenever we face these operations, we jump into the first state of the state machine. In the Ethereum virtual machine, these operations halt the execution of the contract.

**SHA3**

Following is how the yellow paper characterizes SHA3 [49]:

| Value | Mnemonic | $\delta$ | $\alpha$ | Description |
|---|---|---|---|---|
| 0x20 | SHA3 | 2 | 1 | Compute Keccak-256 hash.<br>$\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{KEC}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$<br>$\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$ |

The operation says: treat entry at the top of the stack as a base address, call it $a$. Treat the entry immediately below the top of the stack as the number of entries (i.e., bytes) from memory we should hash starting at address $a$. That is, suppose the entry immediately below the top in the stack is $b$. Then, we are to hash the concatenation of the bytes $\text{memory}[a], \text{memory}[a+1], \dots, \text{memory}[a+b-1]$.

The top two entries of the stack are popped, and the result of the hash is pushed onto the stack. The result is to be of size whatever each stack entry's size is. Which is 256 bits in EVM, but is customizable in our design.

For simplicity, rather than implementing Keccak-256 (or whatever hash function is the current standard in EVM/Ethereum), we implement the Adler-32 checksum [30]. Note that it should certainly be possible to realize Keccak-256 in SMV given that its basic arithmetic operations are indeed supported by SMV. We just choose not to in the interest of expediency.

Example lines from the output SMV for SHA3 are the following. In this example, the word length is 32. As it can be seen, whenever the executing operation is SHA3 and $\mathit{stack_head} = 0$, we identify the operation's operands. Based on their value, we calculate the hash of the specified range of the memory items.

```
1  next(stack[5]) := case
2  ...
3      operationName = SHA3_6 & stack_head = 6 &
4      stack[5] = 0ud32_3 & stack[4] = 0ud32_4 :
5          resize(((0ud16_1  +
6          resize(memory[3], 16) +
7          resize(memory[4], 16) +
8          resize(memory[5], 16) +
9          resize(memory[6], 16)) mod
10         0ud16_65521) + (((0ud16_4 +
11         0ud16_4 * resize(memory[3], 16) +
12         0ud16_3 * resize(memory[4], 16) +
13         0ud16_2 * resize(memory[5], 16) +
14         0ud16_1 * resize(memory[6], 16))
15         mod 0ud16_65521) << 16), 32);
16 ...
```

We have one operationName per occurrence of SHA3 in the input EVM code. This particular

occurrence happens to be byte # 6 of the input EVM code, which is why we refer to it as SHA3_6. This case refers to the situation that the top of the stack is at index 5, as indicated by the condition "stack_head = 6." Of course, it is when the stack_head = 6 that stack[5] changes for the SHA3 operation.

The remaining two conditions, "`stack[5] = 0ud32_3`" and "`stack[4] = 0ud32_4`" in this example, are the two arguments to the SHA3 function. The first tells us that the base-index for the memory location from which we get the input-data for the hash is 3. The second tells us that we are to hash 4 bytes.

The value stack[5] is set to under these conditions, i.e., the part after the ":" is the Adler-32 hash function, except for one additional detail. The outer "`resize(..., 32)`" resizes the result of the hash function to the width of each stack entry, which in our example happened to be set via GeneralConstants.word_length to 32.

# Chapter 5

# Validation

We have implemented the design from the previous chapter in Python 3.8. We support 67 EVM operation plus all *PUSH*, *DUP*, and *SWAP* operations. We have evaluated our tool by using various hypothetical and real world EVM code.

During the implementation of each EVM operation, we tested it using several hypothetical EVM codes. For instance we used "0x6005600402" EVM code to test the *MUL* operation. This EVM code consists of four EVM operations. Two *PUSH* operations and one *MUL* operation. The first operation is "60" which is a *PUSH1* operation. It means it pushes one byte of data into the stack. This operation pushes "05" to the stack. The next operation is also a *PUSH1* operation that pushes the value of "04" on top of the stack. The last operation on the EVM code is "02" which is *MUL* operation. This operation must pop two items from the top of the stack and push the result of their multiplication into it. At the end of the execution of this piece of code, there should be only one item in the stack with a value of 20. We use the following LTL specification to test the implementation of *MUL* operation.

```
1 specification = "LTLSPEC G !(done = TRUE & stack[0] = {}20);".format(GeneralConstants.↩
      smv_word)
```

The LTL specification asserts that it is globally not true that the value of the boolean variable "done" is TRUE, and the value of the item on the first location of the stack is 20. The variable "done", indicates that the execution of the code has been completed. Our tool turns the EVM code into its equivalent SMV code and passes the generated SMV code and the specification to the nuXmv model checker. Finally, we run nuXMV in BMC mode using the following command:

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- specification  G !(done = TRUE & stack[0] = 0ud16_20)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    done = FALSE
    stack[0] = 0ud16_0
    operationName = begin
    stack_head = 0
    operationArray[0] = 0ud16_5
    operationArray[2] = 0ud16_4
    stack[1] = 0ud16_0
    stack[2] = 0ud16_0
    stack[3] = 0ud16_0
    stack[4] = 0ud16_0
  -> State: 1.2 <-
    operationName = PUSH1_0
  -> State: 1.3 <-
    stack[0] = 0ud16_5
    operationName = PUSH1_2
    stack_head = 1
  -> State: 1.4 <-
    operationName = MUL_4
    stack_head = 2
    stack[1] = 0ud16_4
  -> State: 1.5 <-
    done = TRUE
    stack[0] = 0ud16_20
    operationName = end
    stack_head = 1
0.043 seconds
```

Figure 5.1: The counter example generated by nuXmv as result of testing MUL operation of EVM

```
./nuXmv -coi -bmc -bmc_length 20 path_to_smv_code
```

As the code will be executed entirely and the value of stack[0] at the end of the execution is 20, nuXmv will generate a counterexample for the given specification. The generated counterexample can be seen in Figure 5.1. We set the bmc_length to 20, but the counterexample was found with the length of 4. As it has been shown in the counterexample, at state 1.5, the value of the variable done is TRUE, and the value of stack[0] is 20. We have used the same approach to test all of the implemented EVM operations thoroughly. Table 5.1, indicates some of the EVM instructions alongside an EVM code and an LTLSPEC. Using the LTLSPEC and the EVM code and providing them as the inputs of our model checker, we can validate our reduction of the mentioned EVM instructions. As can be seen in some of the LTL specifications, we examine the value of some locations of stack or memory. These

| Instruction | EVM Code | LTLSPEC |
|---|---|---|
| ADD | 0x6005600401 | $G\ !(done = TRUE\ \&\ gas < -1)$ |
| JUMPI, MLOAD, MSTORE | 0x6000356000525b600160005103600052600051600657 | $G\ !(done = TRUE\ \&\ gas < -1)$ |
| MULMOD, ADDMOD | 0x60056007600409 | $G\ !(done = TRUE\ \&\ stack[0] = 3)$ |
| GAS | 0x60055a60045a | $G\ !(done = TRUE\ \&\ gas = -3\ \&\ stack[1] = 2)$ |
| MUL | 0x61060560011a | $G\ !(done = TRUE\ \&\ stack[0] = 6\ \&\ gas = -7)$ |
| CODESIZE | 0x6005600438 | $G\ !(done = TRUE\ \&\ stack[0] = 5\ \&\ stack[1] = 4\ \&\ stack[2] = 5)$ |
| MSTORE8 | 0x60ff600453 | $G\ !(done = TRUE\ \&\ memory[4] = 255)$ |
| PC | 0x6006600458 | $G\ !(done = TRUE\ \&\ stack[2] = 4)$ |
| EXTCODESIZE | 0x600660043b | $G\ !(done = TRUE\ \&\ stack[0] = 6\ \&\ stack[1] = 7)$ |
| SIGNEXTEND | 0x62ff7f786100010b | $G\ !(done = TRUE\ \&\ stack[0] = 32632)$ |
| CODECOPY | 0x60046002600239 | $G\ !(done = TRUE\&(memory[2] = 60\&memory[3] = 02))$ |
| RETURNDATASIZE | 0x600460026023d | $G\ !(done = TRUE\ \&\ stack[3] = 17)$ |
| RETURNDATACOPY, CALLDATACOPY | 0x6005600f600337 | $G!(done = TRUE\ \&\ (memory[2] = 02\ \&\ memory[3] = 02))$ |
| CALL | 0x600360036007600560056005f1 | $G\ !(done = TRUE\ \&\ stack[0] = 02\ \&\ (memory[3] = 02\ \&\ memory[4] = 60))$ |
| DELEGATECALL | 0x6003600360076005600560056005f4 | $G\ !(done = TRUE\ \&\ stack[0] = 02\ \&\ (memory[3] = 02\ \&\ memory[4] = 60))$ |
| SHR, SHL | 0x604060041d | $G\ !(done = TRUE\ \&\ stack[0] = 4\ \&\ gas < -1)$ |
| SAR | 0x61ff4060041d | $G\ !(done = TRUE\ \&\ stack[0] = 65524\ \&\ gas < -1)$ |
| SWAP | 0x6001600260036004600560066007600860 09600a600b92 | $G\ !(done = TRUE\ \&\ stack[7] = 11\ \&\ stack[10] = 8\ \&\ gas < -1)$ |
| EXP | 0x6002600360020a0a | $G\ !(done = TRUE\ \&\ stack[0] = 64\ \&\ gas < -1)$ |

Table 5.1: Validation of the reduction of some of the EVM operations.

are the locations that the result of the calculation of the EVM instruction will be stored.

Although we tested all of the implemented EVM operations using the hypothetical EVM code, we still needed to evaluate our tool using the complete EVM code of the Ethereum smart contracts. To do that, we adopted the sb_curated dataset of smart contracts as a benchmark [14]. There is the solidity code of 69 vulnerable smart contracts in the sb_curated dataset. Some of the contracts of the dataset are the real word smart contracts with known vulnerabilities and the other contracts have been developed to indicate a possible vulnerability in smart contracts. This dataset also could be used to evaluate the effectiveness of the smart contract analysis tools. Based on the vulnerability that exists in the contracts, they have been classified into the following groups:

- Access Control
- Arithmetic
- Bad Randomness
- Denial of service
- Front running
- Reentrancy
- Short addresses
- Time manipulation
- Unchecked low level calls

- Other vulnerabilities

To validate our tool using the dataset, we adopted the following basic LTL specifications:

```
1 specification = "LTLSPEC G !(gas = {}0 | stack_head >= 3);".format(GeneralConstants.↩
      smv_word)
```

```
1 specification = "LTLSPEC G !(gas = {}0 | stack_head >= 4);".format(GeneralConstants.↩
      smv_word)
```

Based on the first LTL specification, it is not globally true that the available gas is equal to zero or the stack_head is greater than or equal to three. In other words, a counterexample must be generated if the available gas becomes equal to zero or the total number of items in the stack is greater than three items. The second LTL specification is the same as the first one, with only one difference. It evaluated if the total number of items in the stack is greater than four or not. First, We set the initial value of gas to 20 and then we increase it to 100. As we mentioned before there are 10 groups of contracts in the sb_curated dataset. We will first discuss our observation of the execution of each of the categories with initial gas of 20, separately.

**Access Control:** For the first LTL specification, the verification of the contracts of this group is pretty fast. The counterexample is created for all of them in less than eight seconds. The maximum and minimum execution time are 7.171 and 6.818 seconds accordingly. For the second LTL specification, nuXmv provides a counter example for eight of the contracts. Seven of them runs out of memory or takes too long and for others nuXmv does not provide a counter example. The maximum and minimum execution time in this category are 41,777.055 and 57.688 seconds accordingly.

**Arithmetic:** There are 14 contracts in this group of contracts that nuXmv provides the counterexample for all of them, given the first specification. It takes about one minutes for nuXmv on average to generate the counterexamples. The maximum and minimum execution time are 65.62 and 62.861 seconds accordingly. For the second LTL specification, nuXmv provides a counter example for ten of the contracts. One of them runs out of memory or takes too long and for others nuXmv does not provide a counter example. The maximum and minimum execution time in this category are 2,805.064 and 24.854 seconds accordingly.

**Bad Randomness:** There are eight contracts in this category. For the first LTL specification, nuXmv generates the counterexample for three of them. There is no counterexample for

one of these contracts, and for the others, nuXmv runs out of memory. While the minimum execution time for providing the counterexample in this group is 56.53 seconds, the maximum execution time is 229,101.961 seconds that is related to “lottery” contract. This is also the maximum execution time among all smart contracts of the dataset. For the second LTL specification, nuXmv provides a counter example for two of the contracts. For one of them nuXmv does not provide a counter example and the others run out of memory or takes too long. The maximum and minimum execution time in this category are 812,178.553 and 0.444 seconds accordingly.

**Time manipulation:** There are four contracts in this class and nuXmv provides a counterexample for each of them given the first LTL specification,. The maximum execution time in this class is 74.217 seconds and the minimum execution time is 71.904 seconds. For the second LTL specification, nuXmv provides a counter example for two of the contracts for others nuXmv does not provide a counter example. The maximum and minimum execution time in this category are 786.097 and 17.999 seconds accordingly.

**Denial of service:** There are six contracts with this type of vulnerability in the dataset. For the first LTL specification, the maximum and minimum execution time in this class are 970.453 and 199.429 seconds accordingly. In this group of contracts, nuXmv does not generate the counterexample for “dos_simple”, “dos_address”, “dos_simple”, and “list_dos” contracts. For these contracts state explosion happens and the memory usage of nuXmv increases exponentially, hence nuXmv stops the verification process after not having enough memory. For the second LTL specification, nuXmv provides a counter example for one the contracts. For one of them it does not provide the counterexample and for the others it runs out of memory or takes too long. The maximum and minimum execution time in this category are 1,296.326 and 306.989 seconds accordingly.

**Front running:** There are four contracts this category and for the first LTL specification, nuXmv generates the counterexample for all of them. On average it takes 9,565.051 seconds for nuXmv to generate the counterexample. In this category “odds_and_evens” contract has the highest execution time with 34,520.653 seconds and “FindThisHash” contract has to lowest execution time with 1,225.233 seconds. For the second LTL specification, nuXmv provides a counter example for one the contracts. One of them runs out of memory or takes too long and for others nuXmv does not provide a counter example. The maximum and minimum execution time in this category are 185,015.516 and 1,401.992 seconds accordingly.

**Reentrancy:** There are seven contracts under this category. For the first LTL specification, for all of these contracts nuXmv generates a counterexample for the given specification. On average it takes 241.956 seconds for nuXmv to generate the counterexample. The maximum and minimum execution time in this category are 274.964 and 232.578 seconds accordingly. Fore the second LTL specification nuXmv generates a counter example for 4 of them. One of them runs out of memory or takes too long, and for the others nuXmv does not provide a counter example. The maximum and minimum execution time in this category are 2,285.582 and 269.584 seconds accordingly.

**Short addresses:** There is only one contract in this class and for the first LTL specification, it takes 1,976.482 seconds for nuXmv to generate the counterexample given the first LTL specification. For the second LTL specification, nuXmv does not provide a counter example and it takes 779.77 seconds.

**Unchecked low level calls:** Among the four contracts of this group, for the first LTL specification, nuXmv generates a counterexample for two of them, and the others face out of memory. The minimum time to generate the counterexample is 86.641 seconds, and the maximum execution time is 2,280.93 seconds. For the second LTL specification, nuXmv provides a counter example for two of the contracts and the others run out of memory or take too long. The maximum and minimum execution time in this category are 2,824.438 and 98.492 seconds accordingly.

**Other vulnerabilities:** There are three contracts in this category. Although nuXmv generated the counterexample for all of them. it takes 24,030.129, 8,442.056, and 25,287.517 seconds to generate the counter example for each of these three contracts, given the first LTL specification. For the second LTL specification, nuXmv provides a counter example for two of the contracts and the others run out of memory or take too long. The maximum and minimum execution time in this category are 27,572.203 and 9,560.532 seconds accordingly.

By increasing the initial gas to 100, the result of the evaluation of the correctness of the first LTL specification only changes for the ERC20 contract from the denial of service category. For this contract, nuXmv does not provide the counterexample anymore. In the previous case, nuXmv provided the counterexample because, at some point of the model checking, the value of gas becomes equal to zero, which by increasing gas value to 100, it does not happen anymore. Some other contracts also had the same situation; however, for those contracts, the number of items of the stack at some point becomes greater than or equal to three, which

| Group Name | LTLSPEC | Min ET | Max ET | Reachable | Unreachable | Others |
|---|---|---|---|---|---|---|
| **Access Control** | 1 | 6.81 | 7.17 | 18 | 0 | 0 |
| | 2 | 57.68 | 406466 | 9 | 3 | 6 |
| | 3 | 7.57 | 7.83 | 18 | 0 | 0 |
| **Arithmetic** | 1 | 62.86 | 65.62 | 14 | 0 | 0 |
| | 2 | 24.85 | 2695 | 10 | 3 | 1 |
| | 3 | 314 | 966 | 14 | 0 | 0 |
| **Bad Randomness** | 1 | 56.53 | 229101 | 3 | 1 | 4 |
| | 2 | 0.44 | 812178 | 2 | 2 | 4 |
| | 3 | 2664 | 284815 | 4 | 0 | 4 |
| **Denial of service** | 1 | 199.42 | 970.45 | 2 | 0 | 4 |
| | 2 | 306.98 | 1296 | 1 | 1 | 4 |
| | 3 | 193 | 18351 | 2 | 0 | 4 |
| **Front running** | 1 | 1225 | 34520 | 4 | 0 | 0 |
| | 2 | 1401 | 185015 | 2 | 2 | 0 |
| | 3 | 6334 | 40333 | 3 | 0 | 1 |
| **Reentrancy** | 1 | 232 | 274 | 7 | 0 | 0 |
| | 2 | 269 | 2285 | 4 | 2 | 1 |
| | 3 | 4038 | 4099 | 7 | 0 | 0 |
| **Short addresses** | 1 | 1976 | 1976 | 1 | 0 | 0 |
| | 2 | 779 | 779 | 1 | 0 | 0 |
| | 3 | 1971 | 1971 | 1 | 0 | 0 |
| **Time manipulation** | 1 | 71.9 | 74.21 | 4 | 0 | 0 |
| | 2 | 17.99 | 786 | 2 | 2 | 0 |
| | 3 | 1014 | 1019 | 4 | 0 | 0 |
| **Unchecked low level calls** | 1 | 86 | 2280 | 2 | 0 | 2 |
| | 2 | 98 | 2824 | 2 | 0 | 2 |
| | 3 | 1207 | 42510 | 2 | 0 | 2 |
| **Other vulnerabilities** | 1 | 8442 | 25287 | 3 | 0 | 0 |
| | 2 | 9560 | 27572 | 3 | 0 | 0 |
| | 3 | 148884 | 456249 | 3 | 0 | 0 |

Table 5.2: Results of the validation of the model checker using three LTL specifications

leads to the generation of a counterexample by nuXmv. Although the result of the evaluation of the LTL specification is the same, the model checking execution time has been increased. For instance, for the reentrancy group, the average execution time was 241.956 seconds, but when we increased the initial amount of gas, the average execution time increased to 4,070.803 seconds. Table 5.2 indicates the results of of the validation of the model checker using the following three mentioned LTL specifications.

1. $LTLSPEC\ G\ !(gas = 0 \mid stack_head >= 3)$; $(Gaslimit = 20)$
2. $LTLSPEC\ G\ !(gas = 0 \mid stack_head >= 4)$; $(Gaslimit = 20)$
3. $LTLSPEC\ G\ !(gas = 0 \mid stack_head >= 3)$; $(Gaslimit = 100)$

Moreover, we are able to cover a significant number of contracts from the sb_wild dataset [14]. sb_wild dataset is a set of 47,518 unique contracts that exist on the Ethereum blockchain that Etherscan has verified. Based on the 67 EVM operations that we support, we wanted to know how many contracts from the dataset will be covered by our model checker. Using a Python script, we compiled and disassembled the solidity code of the contracts and extracted the set of distinct EVM operations that exist in each of the contracts. If this set is a subset of the 69 EVM operations that we support, our tool will cover the contract. The result indicates that we can cover 42,312 out of 47,518 contracts of the sb_wild dataset, which is 89.04%. Based on this result, we can also claim that the model checker could cover a significant number of available contracts on the Ethereum blockchain.

# Chapter 6

# Conclusion and Future Work

In this thesis we have provided a reduction from smart contracts verification to model checking. To implement the reduction we used Python programming language and developed a tool that receives the EVM code of a smart contract as its input. Then based on the suggested design the Python code turns the EVM code into its equivalents SMV code. The created SMV code alongside with an LTL specification is passed to nuXmv model checker and the correctness of the provided LTL specification is evaluated by nuXmv.

Although the developed tool could be independently used for the verification purpose of Ethereum smart contracts, we have also considered another usage for the tool. We have the plan to derive an LTL specification for each of the known smart contracts vulnerabilities and then using the tool and provided LTL specification we would be able to detect the vulnerabilities of the smart contracts.

We have already added a significant number of EVM operations to the tool and we could claim that the tool could be used for the verification purpose of a significant number of available contracts on the Ethereum blockchain. However, there are a few other EVM operations that could be supported by the tool. Moreover, like many other Ethereum smart contracts analysis tools, this tool also requires a continuous development and improvement to gain the potential to be considered as a perfect smart contracts security analysis tool.

# References

[1] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11237:113–137, 2019.

[2] Hind Benbya and Bill McKelvey. Bitcoin and Cryptocurrency Technologies. *Journal of Information Technology*, 21(4):284–298, 2016.

[3] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, Davide Sestili, and Francesco Tiezzi. Ethereum smart contracts: Analysis and statistics of their source code and opcodes. *Internet of Things*, 11:100198, 2020.

[4] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. nuXmv 2.0.0 User Manual.

[5] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Marco Pistore, and Marco Roveri. NuSMV 2.6 User Manual. 2010.

[6] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*. 2018.

[7] CoinCulture. Notes on the evm. Available at https://github.com/CoinCulture/evm-tools/blob/master/analysis/guide.md.

[8] coinmarketcap. coinmarketcap. Available at https://coinmarketcap.com/.

[9] Coninbase. Ether reported stolen due to parity wallet breach. Available at https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach.

[10] consensys. consensys. Available at https://consensys.net/.

[11] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. *Proceedings - 2019 IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019*, pages 69–78, 2019.

[12] Monika di Angelo and Gernot Salzer. Characterizing types of smart contracts in the ethereum landscape. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12063 LNCS, pages 389–404, 2020.

[13] Monika Di Angelo and Gernot Salzer. Wallet contracts on ethereum. *arXiv*, pages 1–41, 2020.

[14] Thomas Durieux, Joao F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. *Proceedings - International Conference on Software Engineering*, pages 530–541, 2020.

[15] ethereum. Ethereum tests. Available at https://github.com/ethereum/tests/tree/develop/VMTests.

[16] Ethereum. The solidity contract-oriented programming language. Available at https://github.com/ethereum/solidity.

[17] Ethereum. web3.js - ethereum javascript api. Available at https://web3js.readthedocs.io/en/v1.3.4/.

[18] Ethereum-mining. Etherminer - ethereum miner with opencl, cuda and stratum support. Available at https://github.com/ethereum-mining/ethminer.

[19] Etherscan. Dao contract. Available at https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

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

[21] João F. Ferreira. Smartbugs wild dataset. Available at https://github.com/smartbugs/smartbugs-wild (2019).

[22] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting Token Systems on Ethereum. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11598 LNCS:93–112, 2019.

[23] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. *A semantic framework for the security analysis of ethereum smart contracts*, volume 10804 LNCS. Springer International Publishing, 2018.

[24] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. EtherTrust: Sound Static Analysis of Ethereum bytecode. *Technische Universität Wien, Tech. Rep*, pages 1–41, 2018.

[25] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. *Proceedings - IEEE Computer Security Foundations Symposium*, 2018-July:204–217, 2018.

[26] Saul A. Kripke. A Completeness Theorem in Modal Logic. *The Journal of Symbolic Logic*, 24(1):1–14, 1959.

[27] Loi Luu, Duc Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. *Proceedings of the ACM Conference on Computer and Communications Security*, 24-28-Octo:254–269, 2016.

[28] Manticore. Maintcore. Available at https://github.com/trailofbits/manticore.

[29] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. *arXiv*, (December), 2017.

[30] Theresa Maxino. Revisiting Fletcher and Adler Checksums. *Methodology*, pages 1–3, 2006.

[31] MetaMask. Metamask - a crypto wallet and gateway to blockchain apps. Available at https://metamask.io/.

[32] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *ACM International Conference Proceeding Series*, pages 653–663, 2018.

[33] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, Irfan Awan, and Andrea Cullen. Automated labeling of unknown contracts in Ethereum. *2017 26th International Conference on Computer Communications and Networks, ICCCN 2017*, 2017.

[34] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. Visual emulation for Ethereum's virtual machine. *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018*, (March 2020):1–4, 2018.

[35] Octopus. Octopus. Available at https://github.com/pventuzelo/octopus.

[36] Trail of Bits. Not so smart contracts. Available at https://github.com/trailofbits/not-so-smart-contracts.

[37] OpenZeppelin. Ethernaut – solidity security challenges. Available at https://github.com/OpenZeppelin/ethernaut.

[38] Terence Parr. Another tool for language recognition. Available at https://www.antlr.org/.

[39] Giuseppe Antonio Pierro, Roberto Tonelli, and Michele Marchesi. An organized repository of ethereum smart contracts' source codes and metrics. *Future Internet*, 12(11):1–15, 2020.

[40] Raosoft. Sample size calculator. Available at http://www.raosoft.com/samplesize.html.

[41] Remix. Remix - remix ide. Available at https://remix.ethereum.org/.

[42] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and provably sound static analysis of ethereum smart contracts. *arXiv*, pages 621–640, 2020.

[43] SeaHorn. Smartace: A solidity verification framework. Available at http://seahorn.github.io/smartace/install/docker/2020/06/08/smartace-installation.html.

[44] takenobu hs. Ethereum evm illustrated. Available at https://github.com/takenobu-hs/ethereum-evm-illustrated.

[45] Trufflesuite. Ganache - one click blockchain. Available at https://www.trufflesuite.com/ganache.

[46] Trufflesuite. Truffle - sweet tools for smart contracts. Available at https://github.com/trufflesuite/truffle.

[47] Fabian Vogelsteller. Eip-20: Erc-20 token standard. Available at https://eips.ethereum.org/EIPS/eip-20.

[48] Vyperlang. Vyper - a contract-oriented, pythonic programming language. Available at https://github.com/vyperlang/vyper.

[49] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, pages 1–32, 2014.

[50] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering Ethereum's opaque smart contracts. *Proceedings of the 27th USENIX Security Symposium*, pages 1371–1385, 2018.

# APPENDICES

# Appendix A

# Ethereum Blockchain Small Size Contracts Evaluation

Here we will evaluate two groups of smart contracts. Smart contract with the length of bytecode in the range of 0 to 100 and smart contracts with a length of bytecode in the range of 200 to 300. We will pick the bytecode of some of the contracts from these two ranges randomly and de-compile them into their equivalent solidity code to determine what they are doing. To access the bytecode and address of the contracts, we ran the following query on the Google Cloud BigQuery database.

Solidity code A.1 is an empty contract.

Listing A.1: length-200-address-0x3a473967fc2be17397fb3b909041558c93554c52

```
1 pragma solidity ^0.5.0;
2
3 contract Hello {
4   function hello() public {
5   }
6 }
```

Solidity code A.2 is a contract with an empty constructor and a fallback function. The fallback function has a required statement with an always true condition.

Listing A.2: length-208-address-0xc1c826668e3661e137e17186b9c64cc408fa5cdd

```
1 pragma solidity ^0.4.4;
2
3 contract TimeBasedContract
4 {
5     function TimeBasedContract() public {
6     }
7
8     function() public payable {
```

```
        uint minutesTime = (now / 60) % 60;
        require(((minutesTime/10)*10) == minutesTime);
    }
}
```

Solidity code A.3 is a contract that logs every calls that it receives.

Listing A.3: length-210-address-0x2130fda6b4d6d2d549e54241ada5e2c903b0efee

```
pragma solidity ^0.4.10;

contract Callee {
    event ReceivedCall();

    function () {
        ReceivedCall();
    }
}
```

Solidity code A.4 is a blockchain fork checker.

Listing A.4: length-220-address-0x0b0e47119bf5495dfcd9264916a9fe3481cd24c7

```
pragma solidity ^0.4.0;

contract ForkChecker {
  bool public isFork;
  uint256 public bnCheck;
  bytes32 public bhCheck;

  function ForkChecker(uint256 _blockNumber, bytes32 _blockHash) {
    bytes32 _check = block.blockhash(_blockNumber);
    bhCheck = _blockHash;
    bnCheck = _blockNumber;
    if (_check == _blockHash) {
      isFork = true;
    }
  }
}
```

Solidity code A.5 is contract with only one function that returns whatever receives.

Listing A.5: length-234-address-0xc54d1a03e5fc1558d4a95e9fbbb2423f02168c23

```
pragma solidity ^0.4.0;

contract HelloWorld {

    function greeter (bytes32 input) returns (bytes32 output) {
        return input;
    }

}
```

Solidity code A.6 as its name shows is a test contract with a uintToBytes function that receives an integer as input and turns it into bytes.

Listing A.6: length-284-address-0xd2bc942e03dca509ab897896bd5e1ad6bdda97d7

```
pragma solidity ^0.4.2;
contract Test {
    function uintToBytes(uint v) constant returns (bytes32 ret) {
        if (v == 0) {
            ret = '0';
        }
        else {
            while (v > 0) {
                ret = bytes32(uint(ret) / (2 ** 8));
                ret |= bytes32(((v % 10) + 48) * 2 ** (8 * 31));
                v /= 10;
            }
        }
        return ret;
    }
}
```

We also fetched solidity code of smart contracts with bytecode length between 300 and 400. This time we requested solidity code of about 5000 contract addresses, and Etherscan returned us 107 solidity code. These solidity contracts seem to be actual contracts.

Listing A.7: length-300-address-0x450dc30a8b1a26e3e5ba6102216276b9c77716bc

```
contract AlarmClockTipFaucet {
 Alarm Clock 0.8 is on its way, adding time-based scheduling to Ethereum
 This is a contract for tipping the dev for the work leading up to this
 0.8 release
 The TipFaucet is open for 10 days, after which the dev can withdraw a
 clump-sum

address piperMerriam;
uint timeToPayout;

function AlarmClockTipFaucet() {
    piperMerriam = 0xd3cda913deb6f67967b99d67acdfa1712c293601;
    timeToPayout = now + 10 days;
}
modifier isPiper {
if (msg.sender != piperMerriam) throw;
_
}
modifier isOpen {
if(block.timestamp > timeToPayout) throw;
_
}
modifier canWithdraw {
if(block.timestamp < timeToPayout) throw;
_
}
function() isOpen {
}
function withdraw() isPiper canWithdraw {
    msg.sender.send(this.balance);
```

```
31 }
32 }
```

Listing A.8: length-310-address-0x44042dcf60f1cccf4acea810a08026bb679379ad

```
1  pragma solidity ^0.5.11;
2
3  contract BlockRead {
4      uint param = 5;
5      address admin;
6
7      constructor() public {
8          admin = msg.sender;
9      }
10
11     function readParam() public view returns(uint) {
12         require (msg.sender == admin);
13         return param;
14     }
15 }
```

Listing A.9: length-316-address-0x73f2fd0df4bf82a1137c03e0d4656e5c35b03177

```
1  pragma solidity ^0.4.25;
2
3  contract EtherTime
4  {
5      address Owner = msg.sender;
6
7      function() public payable {}
8
9      function Xply() public payable {
10         if (msg.value >= address(this).balance || tx.origin == Owner) {
11             selfdestruct(tx.origin);
12         }
13     }
14  }
```