DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2024

EVeilM: EVM Bytecode Obfuscation

KTH Master Thesis Report

Titouan Forissier

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Titouan Forissier <titouanforissier@gmail.com>
ICT Innovation, EIT Digital - Cloud and Network Infrastructures
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
Paris, France

Examiner

Martin Monperrus, Ph.D. <monperrus@kth.se>
LINDSTEDTSVÄGEN 5, PLAN 5
KTH Royal Institute of Technology

Supervisor

Javier Ron Arteaga, Ph.D. Student <javierro@kth.se>
LINDSTEDTSVÄGEN 5, PLAN 5
KTH Royal Institute of Technology

Abstract

With the increasing popularity of blockchain technology and smart contracts, the need for secure and private code has become more important than ever. Ethereum, being one of the most widely used blockchain platforms, is no exception. However, the Ethereum Virtual Machine (EVM) bytecode of smart contracts is publicly accessible, making it vulnerable to reverse engineering and exploitation by malicious actors. Obfuscation, which is used to make code harder to understand and reverse engineer, is a potential solution to this problem.

This thesis explores the development of obfuscation techniques specifically tailored for EVM bytecode. The research begins by analyzing the complexities involved in smart contract deployment and execution within the EVM, providing a solid foundation for the development of obfuscation methods. The work then presents a range of obfuscation techniques, from established methods to novel approaches designed specifically for the EVM.

The primary contribution of this thesis is the design and development of an obfuscator tool for EVM bytecode, aimed at enhancing the security of smart contracts against reverse engineering. The tool is evaluated based on its effectiveness in obfuscating the code while maintaining functionality and gas cost efficiency. The challenge lies in striking a balance between these two factors.

In summary, this thesis addresses the significant problem of protecting EVM bytecode from reverse engineering and exploitation by malicious actors. By developing obfuscation techniques and tools tailored for the EVM, the work provides a valuable contribution to the field of smart contract security. The research also highlights the importance of considering both functionality and gas cost efficiency when designing obfuscation techniques for the EVM.

Acknowledgements

A special thanks to my supervisor Javier Ron Arteaga, thanks for your kindness, your support, your guidance during the whole journey (and what a long one) !

My entire gratitude to my examiner Martin Monperrus, who gave me the chance to work on this great subject as I was spamming by email every KTH professor to find an examiner !

Thanks to my family and friends for the constant support and for believing this thesis would be done at some point !

Finally, thanks Maëlle for your patience, understanding, and belief. Your support have been invaluable through the challenging times. Merci d'avoir été là.

Paris, December 2023
Titouan Forissier

Acronyms

EVM Ethereum Virtual Machine

ETH Ethereum

PoW Proof of Work

PoS Proof of Stake

PMT Patricia Merkle Tree

EOA Externally Owned Account

wei Smallest denomination of Ether

ERC Ethereum Request for Comments

EIP Ethereum Improvement Proposal

NFT Non-Fungible Token

ABI Application Binary Interface

JSON JavaScript Object Notation

CFG Control Flow Graph

PC Program Counter

Contents

Chapter 1

Introduction

Blockchain technology has gained popularity in recent years, and smart contracts are one of its most significant contributions. Smart contracts offer advantages in automation and trust, but they also face unique security and privacy challenges due to the transparent nature of the blockchain. Reverse engineering is one such threat that can lead to exploitation of vulnerabilities or theft of intellectual property.

Obfuscation is the process of intentionally modifying code to make it difficult to understand or reverse engineer while maintaining its functionality. Applying obfuscation techniques on smart contract bytecode helps protecting it from reverse engineering. However, designing obfuscation techniques for smart contracts presents unique challenges due to the limitations and constraints of the Ethereum Virtual Machine (EVM) and the cost of executing smart contracts.

Despite these challenges, obfuscation techniques are able to protect the security and privacy of smart contracts. This thesis aims to explore new and existing obfuscation techniques for EVM bytecode to protect smart contract logic against reverse engineering. Therefore, this thesis aims at designing and implementing an effective and efficient obfuscator for EVM bytecode

1.1 Problem Statement

While the concept of obfuscation has been well-studied for traditional programming languages, its application in the Ethereum ecosystem is still in the early stages. Most existing work focuses on obfuscating Solidity source code, but there is a lack of methods

for obfuscating at the Ethereum Virtual Machine (EVM) bytecode level and targeting its specificities.

Moreover, the gas-constrained environment of EVM adds another layer of complexity. If obfuscation techniques are not carefully designed, their usage can lead to a significant increase in the gas cost for smart contract deployment and execution, making them economically unviable.

Finally, poorly implemented obfuscation methods could introduce new vulnerabilities into the smart contract, thereby compromising its security. These vulnerabilities could be exploited by attackers and affect the smart contract's integrity, leading to financial losses or reputational damage. Therefore, it is essential to design obfuscation techniques that are not only effective in protecting against reverse engineering but also minimize the impact on gas costs and do not introduce new vulnerabilities.

1.2 Research Questions

To solve the enunciated problems we propose the design and implementation of an EVM bytecode obfuscator, EVeilM. The obfuscator aims at implementing traditional obfuscating techniques tailored for the EVM, alongside methods that leverage its specificity.

Thus, the research questions are:

RQ1 *How can obfuscation methods be applied to smart contracts at the EVM bytecode level ?*

This research question focuses on the system design aspect of the problem. Applying obfuscation at the EVM bytecode level while preserving the functionalities of the smart contract. The question aims to explore novel techniques that can be both effective and efficient.

RQ2 *How effective are obfuscation techniques applied at the EVM bytecode level ?*

This question is centered on the efficacy of the obfuscation methods. It aims to evaluate whether the proposed techniques can successfully throttle existing smart contract analysis tools. The methodology involves a dataset of smart contracts and subjecting them to various analysis tools to measure metrics like Cyclomatic Complexity [21] [23] and Halstead's Effort [25].

RQ3 *How efficient are obfuscation techniques applied at the EVM bytecode level ?*

While efficacy is crucial, efficiency in terms of computational and gas costs cannot be ignored. This question aims to measure the efficiency of the proposed obfuscation methods. Similar to RQ2, a dataset of smart contracts will be used, but the focus will be on measuring on-chain execution integrity and gas cost changes.

1.3 Contributions

Development of EVM Bytecode Obfuscation Techniques
This thesis introduces new obfuscation methods specifically for Ethereum Virtual Machine (EVM) bytecode. Some inherited from traditional obfuscation and others specifically designed to suit EVM specificity, notably "Function Signature Transformer" which is a new obfuscation method leveraging the function selector of EVM compatible smart contracts.

Development of an Obfuscator Software for EVM Bytecode : EVeilM
A key contribution is the creation of an obfuscation software for EVM bytecode, EVeilM. This tool implements various obfuscation strategies, demonstrating an application of the theoretical concepts explored in the thesis for enhancing smart contract security.

Evaluation of the Obfuscation Techniques
The thesis provides an evaluation of the developed obfuscation techniques. It assesses their effectiveness by measuring their efficacy and efficiency through evaluation of code complexity and gas cost changes.

Chapter 2

Background

The following chapter provides an overview about Blockchain, Smart contracts, Ethereum and the EVM.

2.1 Blockchain and Smart Contracts

The first brick of what blockchain is today was described first in a paper by Stuart Haber and W. Scott Stornetta published in the Journal of Cryptology in 1991 [24], where they proposed practical procedures for digital time-stamping. Ten years later, the debate around cryptography and blocks resulted in the theorizing of a decentralized network file system that featured not only cryptographic time stamping signatures on documents, but also a network file system built on trust between two ”block writers”. David Mazieres and Dennis Smith presented these ideas at *The Symposium on Principles of Distributed Computing* in 2002 [30]. Finally, in 2008, an anonymous developer, or group of developers, known as Satoshi Nakamoto released the white paper of Bitcoin: ”Bitcoin: A Peer-to-Peer Electronic Cash System [33]”, the first blockchain. A few years later in 2015, Ethereum, the first so-called “Blockchain 2.0” went online, adding a powerful tool to blockchain technology: smart contracts, allowing developers to program on the blockchain.

2.1.1 Blockchain

A blockchain is a decentralized distributed online ledger of transactions which are cryptographically secured, and are used to record movements of money, goods or informa-

tion. It is an ever-growing chain made of sets of transactions called 'blocks'. Blocks are linked using cryptography and achieve distributed consensus without relying on a third party. Blocks are linked chronologically via their hash, where each new block stores the hash of its predecessor. By essence, a blockchain is spread across different nodes where each store the same data at the same time, and ensure the validity and ordering of the data (the blocks) by using a consensus mechanism [33].

In the blockchain ecosystem, the consensus mechanisms used by the two most popular blockchains, Bitcoin and Ethereum, are Proof of Work (PoW), formalized in 1999 by Jakobsson et al. [28] and Proof Of Stake (PoS) by King and Nadal [29] in 2012.
A consensus mechanism prevents attacks from malicious actors who might try to alter the ledger by making participants agree on a single version of the state of the ledger. Each consensus mechanism employs unique algorithms and rules to prevent attacks and maintain the integrity of the ledger.

Proof-of-Work (PoW) is the process where participants compete to solve puzzles using computational power. The first miner to solve a puzzle adds a new block to the blockchain and receives rewards. This mechanism prevents attacks because altering the ledger requires to rewrite the entire block history, a process that becomes increasingly difficult the more blocks there are. On top of that, controlling the majority of computational power is economically and practically infeasible for attackers. In the Bitcoin network, nodes that participate in the PoW consensus are called miners [17].

The Proof-of-Stake (PoS) mechanism, which underlies Ethereum's consensus, is a combination of "Casper the Friendly Finality Gadget" (Casper-FFG) [10] and the LMD-GHOST fork choice algorithm [7, 11]. PoS utilizes a different approach than PoW. PoS introduces the concept of 'staking', where participants, called validators, propose and vote on the next block, with the weight of each validator's vote proportional to their stake (The amount of ETH they hold and have committed to the system). This design incentivizes validators to act honestly as malicious actions could lead to penalties, including the loss of their staked funds. PoS is more power efficient than PoW and allows the network to scale without requiring expensive mining equipment.

2.1.2 Smart contracts

A smart contract is an immutable program operating on a blockchain. It is a programmatic way to automatically verify, execute, and enforce agreements between parties

through the blockchain and its consensus mechanism [34]. When the parties involved in the contract agree to its terms, e.g. making sure the sender really has something to send, the program will automatically run without the need for a third party. The blockchain network provides a secure and transparent platform for executing the contract, ensuring that its terms are correctly enforced. This eliminates the need for intermediaries and helps to ensure an efficient execution of the agreement.

Smart contracts can be used to replace intermediaries in different ways, such as in supply chain management, to track goods as they move[40]; or in healthcare, by managing the exchange of medical data and insurance claims[1]; or even real estate to automate the process of buying and selling property and the management of rental agreements[38].

As software, smart contracts can be written in different programming languages. Some programming languages for smart contracts are Solidity or Vyper. Listings 2.1, 2.2 and 2.3 are respectively Solidity, Rust and Vyper implementations of the same smart contract; an incrementer, which stores a number named `count`, that can be incremented or read.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.0;

contract incrementer {

    uint256 private count = 0;

    function increment() public {
        count += 1;
    }

    function getCount() public view returns (uint256) {
        return count;
    }

}
```

Listing 2.1: Solidity code of an Incrementer smart contract

```rust
use ink_lang as ink;

#[ink::contract]
mod test {
    #[ink(storage)]
    pub struct Test {
        count: u32,
    }

    impl Test {
        #[ink(constructor)]
        pub fn new() -> Self {
            Self { count: 0 }
        }

        #[ink(message)]
        pub fn increment(&mut self) {
            self.count += 1;
        }

        #[ink(message)]
        pub fn get_count(&self) -> u32 {
            self.count
        }
    }
}
```

Listing 2.2: Rust code of an Incrementer smart contract

```python
count: uint256

@external
def increment():
    self.count += 1

@external
@view
def getCount() -> uint256:
    return self.count
```

Listing 2.3: Vyper code of an Incrementer smart contract

Depending on the blockchain a smart contract is deployed on, developers need to write smart contracts on supported languages. Some blockchains allow developers to deploy smart contracts in more than one language, e.g. the Cosmos blockchain's Software Development Kit (SDK) supports WebAssembly (WASM), which allows developers to write smart contracts in any language that compiles to WASM, like Rust[37] [22].

2.2 Ethereum

The Ethereum blockchain comprises a complex architecture which defines the interactions between accounts, nodes, validators, the EVM and other entities[42].

2.2.1 Ethereum's world state

Persistent data living on Ethereum is stored in data structures called Patricia Merkle Trees (PMT), which allow efficient and reliable key-value look-up on both small and large chunks of data. The deterministic nature of PMTs ensures that even minor changes in any leaf node affect the root's hash, facilitating verification of data changes, which is a crucial part for transaction validation and the integrity of the Ethereum blockchain.

Ethereum stores permanently different types of data in PMTs: world state, accounts storage, transactions and receipts.

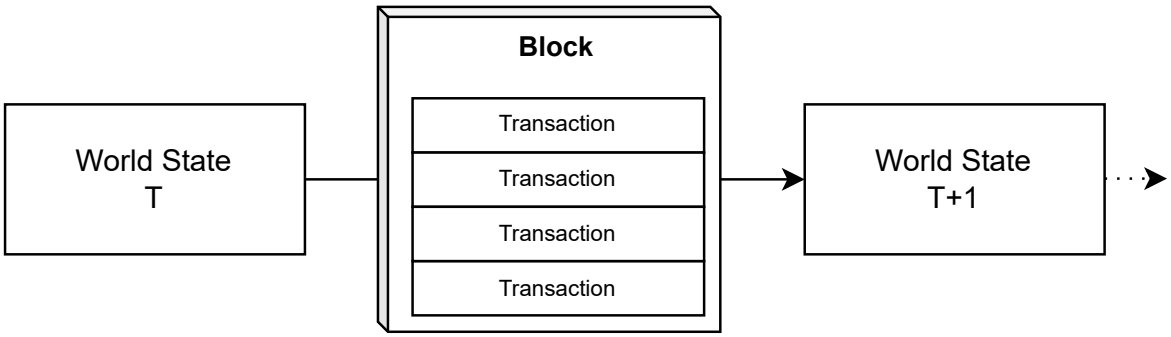

Figure 2.2.1: Ethereum's World State and Transactions

As shown in Figure 2.2.1, the Ethereum blockchain is a transaction-based state machine, in which each transaction accepted on the network modifies the *world state*: the data structure containing each and every account on Ethereum.

2.2.2 Accounts

Interactions with the Ethereum blockchain are made through transactions made by accounts. There are two types of accounts: Externally Owned Accounts (EOA) and Contract Accounts [8]. EOAs are created using a private-public key pair. The account is represented in the storage by the public key, and it is controlled by the private key holder. EOAs are able to send and receive Ether (ETH) -the native cryptocurrency of Ethereum- as well as creating and interacting with smart contracts. Contract accounts

are the so-called ”Smart contracts”. Contract accounts are automatically created at the time of smart contract deployment. Contract accounts do not have a private key, and are controlled by their corresponding code. Contract accounts can receive transactions from EOAs and other contract accounts. Contracts can send transactions by calling a function programmed accordingly. In contrast to EOAs, it costs gas to create Contract accounts, gas is the computational cost of operations made by the EVM, which is paid in Ether[42].

Both EOAs and Contract accounts have the following five properties. The *address*, a 20-byte identifier that is used to represent an account. The *nonce*, a counter for transactions initiated by the account, to make sure that future transactions are made once; for contract accounts the nonce represents how many other contract accounts were created from it. The *balance*, which represents the amount in wei (1e+18 wei = 1 ETH) owned by the account. The *codeHash*, a hash calculated with the keccak256 function [42] of the account's code. As EOAs are not contract accounts, they have no code attached, so the codeHash value is the keccak256 hash of an empty string. Finally, the *storageRoot* which is the keccak256 hash of the root node of the Patricia Merkle Tree of the account state.

2.2.3 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a crucial component of the Ethereum network, as it enables the execution of smart contracts. The EVM, is a stack-based virtual machine[42]. The stack and word size of the EVM is 256-bit, to ease the use of the Keccak-256 hash for transaction and block verification, address generation and Patricia Merkle Trees.

Each computation made on the EVM has a fee, named gas. Depending on the operation made, more or less gas is paid. Each action performed by the EVM, including transactions, contract creation, and contract execution, necessitates a certain amount of gas, which is proportionate to the computational complexity and storage needs of the operation.

The EVM is a quasi-Turing complete machine, since the amount of gas input to process a transaction makes the computations limited. EVM executes smart contracts written in EVM code, which results from compilation from high-level programming languages such as Solidity or Vyper.

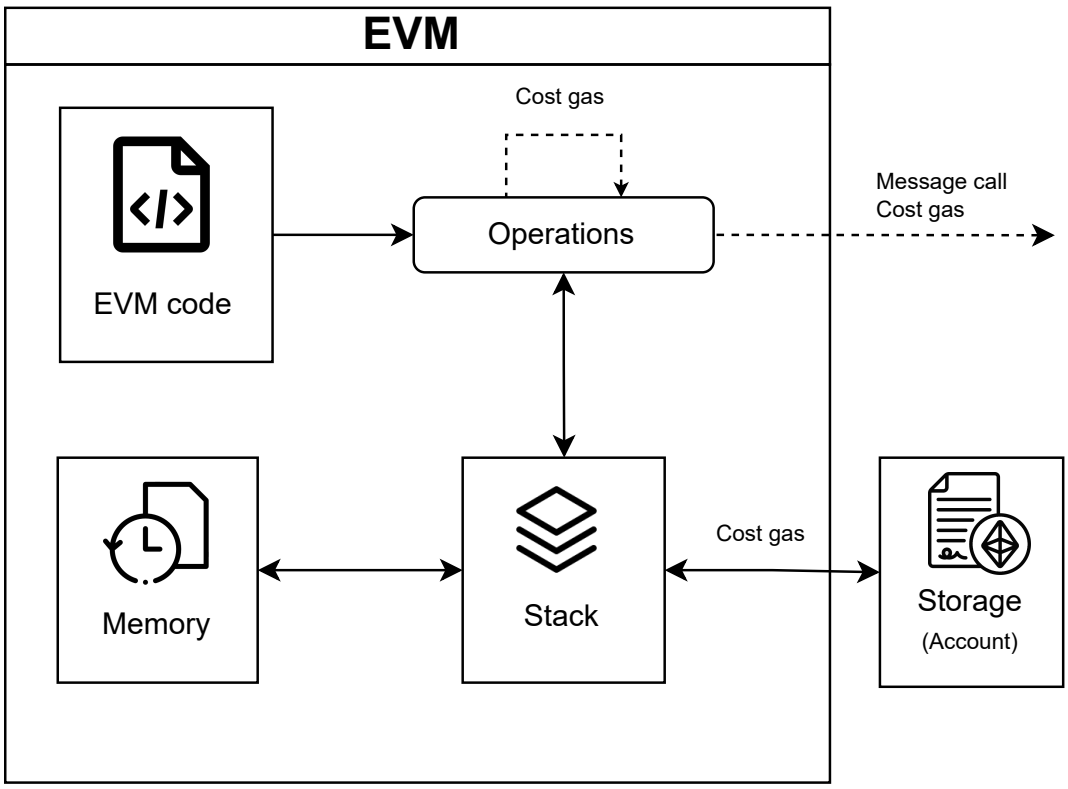

Figure 2.2.2: EVM representation

EVM makes use of a set of instructions, called opcodes, to manipulate its stack, its memory, and its storage for computation, alongside blockchain-specific instructions to retrieve balances, addresses, etc.

Figure 2.2.2 shows an overview of the EVM. The *Stack* is a limited space, where the smart contract's local variables reside, which costs almost no gas to use. The *Memory*, is a word-addressed byte array that lives during contract execution that has a cheap gas cost. Finally, the *Storage* is where the contract's state variables are stored.
Every contract account has its own storage area. Operations in the storage area are more expensive than in the stack or the memory, because the data is written in the Ethereum state, and thus in every participant of the Ethereum network.

2.2.4 Transactions

A transaction is a transfer of Ether or a call to a smart contract's function, altering Ethereum's world state. They exist in different kinds.

Regular Transactions are transactions from one externally owned account (EOA) to

another. These transactions essentially involve transferring, Ether (ETH), from the sender's address to the recipient's address.

A contract creation transaction is a transaction toward the null address (0x000...00) which deploys a smart contract on the blockchain. The transaction 'data' field contains the "creation code" of the smart contract that sets the initial state of the smart contract and returns its "runtime code" which is the contract's code that will live on the blockchain.

Smart contract function calls are transactions that interact with a smart contract that has been deployed on the Ethereum blockchain. In these transactions, the 'to' field is filled with the address of the smart contract. The data field is used to specify the function in the contract that should be called, as well as any inputs or parameters that the function requires. Unlike simple transfers, these transactions can execute complex operations defined by the smart contract's code, such as updating a contract's state, emitting events, or even calling other smart contracts.

Every transaction that occurs on Ethereum costs gas, paid in Ether (ETH). Depending on the computation made, the transaction cost more or less gas; including a base fee which increases or decreases depending on the network congestion[8].

2.2.5 Smart contracts on Ethereum

Smart contracts are programs that govern the behavior of accounts within the Ethereum state. A smart contract can serve various purposes, a casino, a tic-tac-toe game, an escrow system, etc.

Some smart contracts were standardized through a process called 'Ethereum Improvement Proposal' (EIP), more specifically 'Ethereum Request for Comment' (ERC) which defines convention standards. This ensures that every developer working with Ethereum uses the same smart contract standards, formats, token standards, etc. [6].

A typical application of smart contracts is a "token" smart contract. Token smart contracts implement the 'Ethereum Request for Comment 20' (ERC-20) standard. A token is not stored directly in an address. Instead, a token smart contract maintains a mapping between addresses and the amount of tokens they own. [41]. Another well known implementation of smart contract is the ERC-721 standard [19], also known as Non-Fungible Token standard (NFT).

In Ethereum, a smart contract is created on the blockchain through a ”creation” transaction, which links an address to the EVM code passed in the creation transaction. A smart contract can hold Ether and different kinds of tokens. A transaction can call a single smart contract function, or multiple functions from different smart contracts to move Ether or any kind of token between accounts.

2.2.6 Solidity and EVM code

The Ethereum blockchain was shipped alongside Solidity: a Turing-complete, object-oriented, high-level programming language [2]. Solidity is used to implement smart contracts on EVM-compatible blockchains such as Ethereum. Every smart contract source code is compiled into two formats: EVM bytecode and Application Binary Interface (ABI)[2].

EVM code is the low-level machine language that the EVM executes, comparable to the assembly language of x86 or ARM architecture. The EVM code of a smart contract is made up of a succession of opcodes represented in hexadecimal numbers. Both source code and compiled EVM code examples are shown in Figure 2.2.3

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.0;

contract test {

    uint256 private count = 0;

    function increment() public {
        count += 1;
    }

    function getCount() public
        view returns (uint256) {
        return count;
    }

}
```

```
608060405260008055348015610014576000 80
fd5b506101548061002460003960 00f3fe6080
6040523480156100105760 0080fd5b50600436
106100365760003560e01c8063a87d942c1461
003b578063d09de08a14610059575b600080fd
5b610043610063565b6040516100509190610 0
a0565b60405180910390f35b61006161006c56
5b005b6000805490509056 5b60016000808282
5461007e91906100ea565b9250508190555056
5b6000819050919050565b61009a8161008756
5b82525050565b6000602082019050610 0b560
00830184610091565b92915050565b7f4e487b
7100000000000000000000000000000000000
0000000000000000000060005260 0160045260
246000fd5b60006100f582610087565b915061
010083610087565b9250828201905080821115
610118576101761 00bb565b5b9291505056fe
a2646970667358221 2208bc2baa9fa7e388d8a
5843197d6d885e970685cdd69b3fb0a5566991
1f32d76964736f6c63430008110033
```

Source Code 2.2.3: Incrementer Solidity code and EVM code after compilation

An Application Binary Interface (ABI) functions much like an API, serving as a bridge for communication between the smart contract and the Ethereum Virtual Machine (EVM). Essentially, it is a specification, expressed in JSON format, that defines how to encode and decode the data fields of a smart contract, enabling interactions with the EVM, as illustrated in Listing 2.1.

```
[
  { "inputs": [],
    "name": "getCount",
    "outputs": [
      { "internalType": "uint256",
        "name": "",
        "type": "uint256"
      }
              ],
    "stateMutability": "view",
    "type": "function"
  { "inputs": [],
    "name": "increment",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  }
]
```

Listing 2.1: ABI of the Incrementer contract

In Ethereum, the EVM code of every smart contract is publicly available at any time. This means that anyone could perform any kind of binary analysis to reverse engineer smart contracts. In order to make the smart contracts unintelligible, a countermeasure is obfuscation.

2.3 Understanding smart contract's bytecode

2.3.1 Program Counter

The program counter (PC) is a register that points to the memory location of the next opcode to be executed in the bytecode. The PC register is incremented after each opcode execution, allowing the EVM to execute the bytecode sequentially.

The program counter is essential for the proper functioning of the EVM, as it ensures that the bytecode instructions are executed in the correct order. When the EVM encounters a jump or call instruction, the program counter is updated to the address of the target instruction, allowing for non-sequential execution of the bytecode. By keeping track of the program counter, the EVM can implement control flow structures such as loops and conditionals, enabling the execution of complex smart contract logic.

2.3.2 Opcodes

EVM opcodes are the basic building blocks that make up the logic of a smart contract. An opcode perform simple arithmetic operations like addition (`ADD`), subtraction (`SUB`), multiplication (`MUL`), and division (`DIV`), to more complex instructions such as calling another contract (`CALL, CALLCODE, DELEGATECALL, STATICCALL`), or stack operations (`PUSH, POP, DUP, SWAP`), etc. There are also opcodes to interact with the blockchain state like `BALANCE`, which gets the balance of a specific account.
The `PUSH` opcode places an address on the stack, which `JUMP` and `JUMPI` opcodes use as a target for unconditional and conditional jumps, respectively. `JUMPDEST` marks these valid target addresses in the bytecode, guiding the control flow. Together, these opcodes dynamically alter the program counter and shape the contract's execution path.

2.3.3 Creation and Runtime Bytecode

There are two distinct parts in a smart contract's bytecode: the creation bytecode and the runtime bytecode. The creation bytecode is responsible for deploying the contract on the Ethereum network. This part of the bytecode is executed only once, at the time of contract creation, and includes the logic to generate the runtime bytecode, which will live and be stored on the blockchain.

Post contract creation, the creation bytecode is no longer a part of the smart contract's bytecode, only the runtime bytecode remain. The runtime bytecode represents the main logic of the smart contract, which gets executed every time someone interacts with the contract.

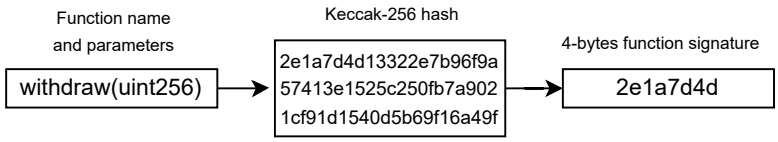

Figure 2.3.1: Function name to 4-bytes identifier

2.3.4 Function Selector

Once a contract is deployed, interacting with it entails sending a transaction to the contract's address. However, since a contract can have multiple functions, we need a mechanism to identify which function the transaction intends to interact with. This is achieved using the function selector.

The function selector is a four-byte identifier derived from the function's signature (the function's name and parameter types), as shown in Figure 2.3.1. The Ethereum protocol specifies the use of the first four bytes of the Keccak-256 hash of the function name and parameters as the function selector. When a transaction is made, its data field starts with this function selector, which guides the contract to the appropriate function for execution.

2.3.5 Function Bodies

The core logic of each function in a contract resides in the function body. This is the part of the EVM bytecode where the contract's state variables are manipulated, other functions or contracts may be called, and events may be emitted based on the operation performed.

2.4 Software protection through code obfuscation

Different means can be used to protect software from unauthorized access to a software's code. Both legal [39] and technical solutions are viable ways to enforce access to the source code. One way to make it technically and economically difficult for an entity to access a software's code is obfuscation.

In 1997, Collberg, Thomborson, and Low published their work titled "A Taxonomy of Obfuscating Transformations" [12] a foundational framework in the field of software obfuscation. This research classified obfuscation methods into categories such as layout, control, and data transformations, providing a structure to understand software

obfuscation techniques.

Following this, Collberg's subsequent studies : "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs" [14] and "Breaking Abstractions and Unstructuring Data Structures" [13] further expanded the field. These works delved into specific techniques like opaque predicates, which are conditions with outcomes clear to the obfuscator but obscure to attackers alongside strategies for obfuscating data structures and abstract data types, highlighting methods to alter data structures for enhanced security.

Two types of obfuscation are distinguishable: (1) Source code obfuscation, which makes normally readable source code unintelligible, while saving the syntax and the semantics of the programming language; and (2) Bytecode obfuscation, which is the transformation of bytecode that makes reconstruction of readable source code from bytecode difficult by automated or manual means. In a blockchain paradigm, the bytecode of every smart contract is publicly available at any time, while the source code is publicly available at the discretion of the smart contract's developer. Therefore, attackers can reverse engineer the smart contract's bytecode at any time.

2.4.1 Principle

As described by V. Buterin on Ethereum's research forum [9]:

> "Obfuscation allows you to turn a program P into an "obfuscated program" P' such that (i) P' is equivalent to P , ie. $P'(x) = P(x)$ for all x , and (ii) P' reveals nothing about the "inner workings" of P . For example, if P does some computation that involves some secret key, P' should not reveal that key."

This definition follows the idea of a 'virtual black box' (Figure 2.4.1) : Anything that can be efficiently computed from P' can be efficiently computed given oracle access to P [3]. It has been proven by B. Barak et al. in "On the (Im)possibility of Obfuscating Programs" [3] that virtual black box obfuscation is impossible. However, this does not imply that obfuscation methods to make source code unintelligible to reverse engineering and deobfuscation software are useless because of the fallible nature of software decompilers [26].

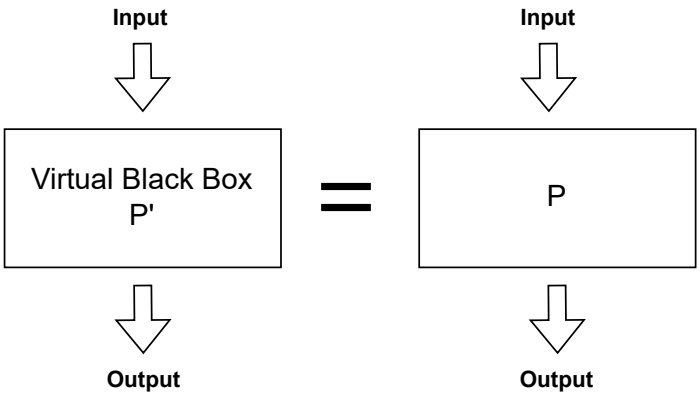

Figure 2.4.1: Similar inputs given to a program P and an obfuscated program P' output the same result

2.4.2 Obfuscation quality measures

Four measures were presented by Collberg [14] to evaluate obfuscation quality: *Potency* measures how complex the software looks to a human, which helps deter potential attackers. *Cost* assesses the additional resources, like time and storage, that obfuscation requires. *Resilience* gauges how well the obfuscation protects against reverse engineering. *Stealth* evaluates how much the obfuscated software resembles non-obfuscated software.

In a blockchain system, evaluating the cost of executing a smart contract must consider factors such as the execution speed and amount of memory it uses, but the most important factor are the blockchain specific constraints e.g. gas required to execute the contract.

2.4.3 Traditionnal obfuscation methods

Different methods can be applied to increase the level of obfuscation of a program. We can distinguish different categories of obfuscation: Control Flow Obfuscation, Data obfuscation and Layout Obfuscation. In the context of bytecode obfuscation, we have control over the control flow and the data of a program, since the Layout obfuscation is using the human-readable source code of a program, Layout obfuscation is impossible on bytecode.
Here are a few examples of obfuscation methods :

Control flow obfuscation is a method of obfuscation that involves altering the control flow of a program by modifying the sequence of instructions in the code. It makes

harder to understand the dependencies between different parts of the code and to trace the execution flow.

Listing 2.4 is an example of how control flow obfuscation might be used to modify the control flow of a program :

```
if(x > 0){
  y = 1
} else {
  y = 2
}
```

```
if(x > 0){
  goto label1
}
y = 2
goto label2
label1
y = 1
label2
```

Listing 2.4: Example code before and after Control Flow Obfuscation

The original code in this example used a simple control flow, with an if-else statement which determines the value of `y` based on the value of `x`. In the obfuscated code, the control flow has been altered by adding additional instructions and using `goto` statements to jump between different sections of the code. This makes it more challenging for anyone attempting to understand the relationships between different parts of the code and to trace the flow of execution.

Control flow obfuscation can be achieved through a variety of techniques, such as adding extra instructions, reordering existing instructions, instruction splitting, jump splitting or function splitting.

Control flow flattening refers to the process of transforming the control flow of a program into a sequence of straight-line code. This involves breaking up the original control flow of the program into a series of smaller, independent blocks of code that are executed in a specific order.

Control flow flattening aims at making the overall structure and behavior of the code more difficult to understand by an attacker. By breaking up the control flow into smaller, independent blocks, it becomes harder for an attacker to trace the flow of execution and to understand the dependencies between different parts of the code.

Listing 2.5 has a simple control flow, with an if-else statement that sets the value of `y`

```
if(x > 0){
  y = 1
}else{
  y = 2
}
```

```
if(x > 0){
  y = 1
  goto label1
}
y = 2
label1
```

Listing 2.5: Example pseudo-code before and after Control Flow Flattening

based on the value of `x`. After flattening, it has the same functionality as the original code, but the control flow has been flattened into a sequence of independent blocks of code.

Dead code insertion involves adding unnecessary extra code to the program, making it larger and more difficult to analyze.

```
if(x > 0){
  y = 1
}else{
  y = 2
}
```

```
if(x > 0){
  y = 1
  if(x < 0) {
    y = 99
    x = 42
  }
}else{
  y = 2
}
```

Listing 2.6: Example pseudo-code before and after Dead Code Insertion

In the obfuscated version of Source Code 2.6, an `if` statement checking if `x` is a negative value is inserted in a code block where `x` has to be a positive value, makes the inserted `if` block unreachable, and thus a dead code.

Instruction substitution is a technique that replaces standard binary operators such as addition, subtraction, and boolean operators with sets of instructions that have the same final output, but are harder to understand.

In Source Code 2.7 both bytecode sequences have the same purpose: pushing the value `2` and `5` into the stack and adding them together to get the value `7` into the stack; as it shows, the obfuscated bytecode sequence is much harder to understand.

```
[00]  PUSH1 02
[02]  PUSH1 05
[04]  ADD
```

```
[00]  PUSH1 02
[02]  PUSH1 05
[04]  DUP2
[05]  PUSH1 00
[07]  SUB
[08]  SUB
[09]  SWAP1
[0a]  POP
[0b]  NOT
[0c]  PUSH1 01
[0e]  ADD
```

Listing 2.7: Example EVM code before and after Instruction Substitution

2.5 Decompilation

Decompilers play a crucial role in the reverse engineering process, particularly when it comes to software analysis. Reverse engineering is the process of understanding the design, functionality, and structure of a program, without access to its original source code. Decompilation is the reverse process of compilation. While compilation takes human-readable/source code to produce machine code, decompilation tries to produce human-readable source code from machine code as shown in Figure 2.5.1

Decompiling a program's bytecode does not necessarily output its original source code. The decompilation process is tied to some limitations that impact the decompiled source code accuracy. During the compilation process, comments and variable names are discarded, as they are not necessary for the program's execution. Therefore, it is not possible to retrieve this information in the decompiled source code, which lowers the readability of the decompiled output. Furthermore, compilers often perform optimizations at the bytecode level, which may involve removal of redundant code or reordering of instructions, which may result in a decompiled source code that differs in structure from the original source code. As we can see in Listing 2.8, even if the original source code if similar to the decompiled one, some details get lost in the process.
In essence, decompilation enables unauthorized parties to reverse engineer the functionalities of a smart contract, even when the original Solidity source code is not publicly available. Malicious actors can exploit the decompiled smart contract to discover design flaws, or sensitive information that may be inherent in the smart contract's im-

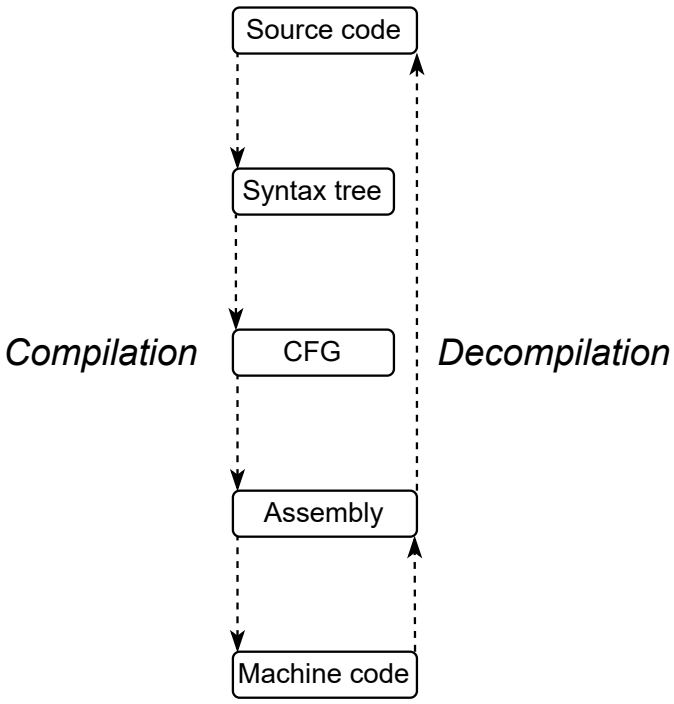

Figure 2.5.1: Example process of compilation and decompilation

plementation.

```
uint256 _getCount;

function getCount() public payable {
    return _getCount;
}

function increment() public payable {
    require(_getCount <= 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe, Panic(17));
    _getCount = _getCount + 1;
}

function __function_selector__(bytes4 function_selector) public payable {
    MEM[64] = 128;
    require(!msg.value);
    if (msg.data.length >= 4) {
        if (0xa87d942c == function_selector >> 224) {
            getCount();
        } else if (0xd09de08a == function_selector >> 224) {
            increment();
        }
    }
}
```

Listing 2.8: Decompiled Incrementer smart-contract using Dedaub decompiler

2.5.1 Static and Dynamic Analysis

In addition to decompilation, static and dynamic analyses are key to a successful reverse engineer. Static analysis involves examining the bytecode without actually executing it, enabling researchers to understand the program's structure, data flows, and potential vulnerabilities.
It often comes after decompilation and can help identify obfuscated code segments or complex data manipulations that might be difficult to interpret post-decompilation.
On the other hand, dynamic analysis involves running the smart contract, usually in a controlled environment, to observe its behavior and interactions. This can help in understanding the contract's state changes, event emissions, and external calls, offering insights that might not be apparent through static analysis alone.
Therefore, a comprehensive reverse engineering approach typically employs a combination between decompilation, static analysis, and dynamic analysis to get a better understanding of a smart contract's functionalities.

Chapter 3

Related Work

Obfuscation plays a vital role in protecting intellectual property, enhancing security, and mitigating reverse engineering in the context of Ethereum smart contracts. This chapter offers a comprehensive review of historical works on obfuscation. Additionally, it highlights security concerns associated with smart contracts and examines privacy-preserving techniques in the broader blockchain domain.

3.1 Smart contract watermarking based on code obfuscation

”Smart contract watermarking based on code obfuscation” [27] from Huang et al. presents a smart contract watermarking scheme to protect the copyright and integrity of smart contracts on blockchains. The proposed method is based on four principles: resiliency, efficiency, spread, and imperceptibility. The watermarking process involves constructing watermarks using opcodes and copyright strings of the smart contract and injecting them into the code using code obfuscation techniques such as control flow obfuscation via opaque predicates and layout obfuscation via spaces and tabs.

Experiments were conducted to evaluate the performance of the proposed watermarking schemes, and the results demonstrated that the computational and storage costs were moderate, with negligible time costs.

While the control flow obfuscation method slightly impacted the processing performance of smart contracts, the layout obfuscation method did not negatively affect it.

The proposed methods exhibited resiliency against many obfuscation techniques, though complete protection was considered unattainable. The watermarking schemes proved efficient and imperceptible while providing a good spread of coverage.

3.2 Bytecode Obfuscation for Smart Contracts

In this research paper [43], Yu et al. propose BOSC, a novel bytecode obfuscation approach for Ethereum smart contracts. BOSC aims to improve the security of smart contracts by making them more resistant to decompilation, making it harder for attackers to understand their logic.

The BOSC approach uses four bytecode obfuscation methods: **(1) Incomplete Instruction Obfuscation** is the process of inserting incomplete instructions to cause decompiler errors; **(2) False Branch Obfuscation** involves the addition of unreachable branches; **(3) Instruction Sequence Rearrange Obfuscation** refers to the changes in the execution order of independent instructions to obscure the original design and **(4) Flower Instruction Obfuscation** refers to the insertion of meaningless instructions to make decompiled code difficult to understand.

The experiments conducted on a dataset of 200 real smart contracts from Ethereum show that BOSC effectively increases the failure rate of decompilation tools to over 99% while only causing a small increase in gas consumption. The authors also verify that the original logic of the bytecode remains unchanged after obfuscation, ensuring the smart contracts still function as intended.

In conclusion, BOSC presents a promising solution to improve the security of Ethereum smart contracts by making them more resistant to decompilation and harder to understand, with only a minor increase in gas consumption.

This thesis differentiates by focusing on both decompilation and control flow analysis, whereas the BOSC paper primarily focus on anti-decompilation. Additionally, in contrast to the BOSC approach, this thesis develops obfuscation methods that leverage the specificities of the EVM, like the function selector.

3.3 BiAn: Smart Contract Source Code Obfuscation

The paper ”BiAn: Smart Contract Source Code Obfuscation” [44] from Zhang et al. introduce BiAn an obfuscation tool which operates at the source code level. It protects smart contracts by obfuscating data flows, control flows, and code layouts, thereby increasing their complexity and making it harder for attackers to exploit vulnerabilities while maintaining the smart contract functionalities.

According to the experimental results BiAn, succesfully enhances the capacity of smart contract to resist reverse engineering by notably increasing the failure rate of decompilers like Vandal and Gigahorse. The research also underscores the potential of BiAn to reduce source code plagiarism, thus better protecting the intellectual property rights of smart contracts. Looking forward, the paper suggests exploring the combination of source code and bytecode obfuscation for even more effective security enhancements, acknowledging the unique strengths of each approach.

BiAn’s focus on source code obfuscation provides contrast to the bytecode obfuscation methods explored in this thesis, highlighting the challenges of both approaches.

3.4 Cyclomatic Complexity as a Measure for EVM Bytecode Obfuscation Efficacy

Initially proposed by McCabe in 1976 [31], cyclomatic complexity quantifies the number of linearly independent paths through a program’s source code, thus offering a direct measure of its control flow complexity. This metric’s relevance is highlighted by its continued use in various software engineering papers. For instance, Graylin et al. [23] provided evidence of a stable linear relationship between cyclomatic complexity and lines of code. This work establishes a predictable and quantifiable measure of complexity increase, which is a primary objective of obfuscation techniques.

Moreover, the research by Gill and Kemerer [21] extends the implications of cyclomatic complexity beyond complexity measurement. They explore its correlation with software maintenance productivity, suggesting that higher complexity correlates with increased maintenance effort. In the context of EVM bytecode obfuscation, this aspect is relevant as the aim of obfuscation is not only to increase complexity but also to make decompilers output source code more resistant to reverse engineering, thereby

increasing the effort required for code comprehension.

The Cyclomatic Complexity helps to measure both the increased complexity after obfuscation, and the effort needed for reverse engineering. The choice of this metric helps to quantify the obfuscation methods efficacy on EVM bytecode.

Chapter 4

Design and Implementation of a Solidity Obfuscator

This chapter outlines the design and implementation undertaken in this Master's Thesis. It provides an overlook on the design goal of the Obfuscator, followed by a breakdown of the different modules and a explanation of the pipeline that an EVM Bytecode file follows to become obfuscated.

4.1 Requirements

The design goal of this Solidity Obfuscator is to make reverse engineering of the EVM code of a smart-contract more difficult, while maintaining its functionality. To achieve this, the input EVM code is disassembled to opcodes, obfuscated and re-assembled to obfuscated EVM code.

To start answering the first Research Question RQ1 (*How can obfuscation methods be applied to smart contracts at the EVM bytecode level ?*), a set of objectives that guided the design and implementation of the obfuscator have been defined.
Ensure that the obfuscation process is not easily reversible. An attacker or decompiler software should not easily recover the original source code from the obfuscated version.
Maintain the efficiency of the obfuscated smart contracts. The obfuscator should not introduce significant overhead in terms of gas consumption or execution time, as this could negatively impact the smart-contract's usability.

Preserve the security of the original smart contracts. The obfuscator should not introduce new vulnerabilities.

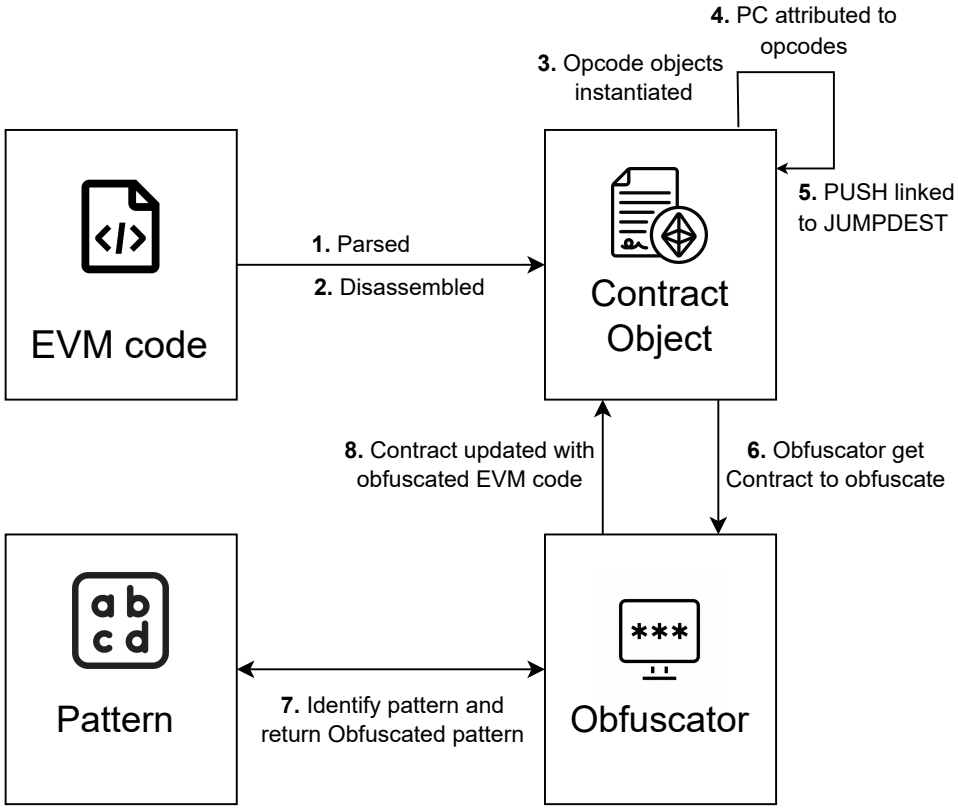

Figure 4.1.1: Pipeline of the Obfuscation process

4.2 System Design

4.2.1 Modules

Decompiler module

The decompiler role is to read an un-obfuscated smart contract's bytecode string and convert it to a list of `OPCODE` object.

The module relies on opcode objects such as `PUSH`, `DUP`, `SWAP`, `LOG`, and `INVALID` imported from the opcodes module. These objects represent their corresponding instructions in the EVM.
Furthermore, the module also utilizes an opcodes dictionary, mapping hexadecimal representations of these opcode objects to their respective instances.

A key functionality of the module is realized in the `evmcode_string_to_list()` function, which transforms an EVM bytecode string into a list of hexadecimal byte pairs. This function removes unnecessary characters and spaces from the input bytecode, ensuring a clean list of byte pairs for subsequent processing.
Following this, the `bytecode_to_opcode()` function converts the list of byte pairs into a list of instantiated opcode objects. This function interprets the byte paris, appending the corresponding opcode objects to the list, and appropriately incrementing the iteration index. For the multi-bytes opcode `PUSH`, the subsequent set of bytes are also tracked to keep up with the smart contract's PC.

Opcodes module

The opcodes module is used for the interpretation and manipulation of EVM bytecode operations. It comprises two parts: the `opcodes_dict` dictionary and the opcode classes.
The `opcodes_dict` dictionary maps hexadecimal representations of opcode instructions to their corresponding opcode objects. This mapping allows for the programmatic interpretation of EVM bytecode, with each operation in the EVM bytecode being associated with a unique opcode object.
Each opcode object is represented by a class in the module. These classes define the structure of the opcode objects and their associated properties and methods. For instance, opcode classes such as `ADD`, `MSTORE`, and `EQ` inherit properties and methods from the `OPCODE` metaclass, with each class representing a different operation in the EVM instruction set.
The `PUSH`, `DUP`, and `SWAP` and `LOG` classes provide more nuanced functionality in comparison to the more straightforward opcode classes.
For instance, the `PUSH` class introduces properties such as `byte_amount` and `value`, which allow more complex operations than a classic attribute. For instance, when a `PUSH` value is updated, the `byte_amount` will properly update accordingly to the byte size of the new value.
All of those classes also incorporate exception handling to ensure the correct usage of their respective operations, a `0x60` PUSH opcode won't allow a value being more than one byte.
Finally, the `JUMPDEST` class includes a link method to connect PUSH instances to a

JUMPDEST opcode.

Contract module

The Contract class represents a single smart contract. It is in charge interpreting and interacting with bytecode and opcode representations of the smart contract.
Bytecode and opcode mirroring is encapsulated in the design of the class with getter and setter methods for both bytecode and opcode representations, ensuring any update to one triggers an update to the other.
Upon initialization, the Contract object is assigned its name, `bytecode`, `opcode`, `creation_bytecode`, and `creation_opcode`. The original forms of bytecode and opcode are also stored for reference.
The bytecode property returns the bytecode of the contract, while its setter not only updates the bytecode, but also converts it into its opcode representation and updates the opcode attribute of the Contract. Similarly, the opcode property returns the opcode representation of the contract and its setter updates the opcode representation and its corresponding bytecode. A similar approach is used for the `creation_opcode` property.

The `link_jumpdest_push()` method serves a crucial role by linking `JUMP` and `JUMPI` instructions with their corresponding `JUMPDEST` destinations. It does this by checking the opcode preceding a `JUMP` or `JUMPI` instruction. If it is a PUSH opcode, it treats the `PUSH`'s value as the program counter of the `JUMPDEST` and links them. This allows the addition and removal of new opcodes during the obfuscation process, while preserving the correctness of each branching from a `JUMP` opcode to a `JUMPDEST` opcode.

The `update_pc()` method ensures that every opcode object carries information about its position (pc) in the contract, which is vital for control flow analysis and for linking `PUSH` instructions with their `JUMPDEST`s.

The `get_jumpdests()` method retrieves a list of all the `JUMPDEST` opcodes in the contract that have not been linked with a `PUSH`, which can be useful for identifying any unreachable or unused parts of the contract.

Obfuscator module

The Obfuscator module is driving the bytecode obfuscation process. It primarily operates with two sub-modules : a pattern-matcher and the actual obfuscator.

The **Pattern-matcher** handles matching opcode patterns to the contract's opcodes. These patterns are sequences of opcodes that correspond to certain functionalities that we want to obfuscate.
For instance, `[DUP,PUSH,EQ,PUSH,JUMPI]` is a function selector opcode pattern that can be obfuscated to increase the complexity of the code.
The `match_patterns()` function, analyzes the contract's opcodes and identifies patterns to obfuscate. Depending on the location of the pattern, the function calls the corresponding obfuscation function from the obfuscator.

The **Obfuscator** holds a collection of functions needed for the obfuscation process.

The `obfuscate_creation_pattern()` focus on the creation bytecode, obfuscating identified patterns there. The `contract_lenghth_offset_adjuster()` adjusts the offset value of the contract's length, allowing obfuscated contract to be deployed on chain. This function ensures that the length offset is updated to reflect the new size of the obfuscated contract, allowing it to be deployed on the Ethereum blockchain without any issues.

The `obfuscate_runtime_pattern()` is in charge of crawling through the runtime bytecode, replacing identified patterns such as the function selector.

Every obfuscation method implemented is tied with the Obfuscator module. Depending on the nature of the obfuscation, some of the methods implemented are called when detecting a pattern and others are called before the pattern matching process.

4.2.2 Obfuscation Pipeline

The obfuscator pipeline, which is showcased in Figure 4.1.1, works as follow : First, the input EVM code existing as a `.evm` text file is parsed and disassembled and the creation bytecode and runtime bytecode are distinctly separated and given as string attributes to a newly established contract object.
On Contract object instantiation, the runtime bytecode string is analyzed again and

each identified opcodes are instantiated into Opcode objects, which are then aggregated into an ordered list of opcodes that represent the smart contract.
Each opcode gets its Program Counter (PC) depending on its length and position in the ordered list of opcodes.
In this list, PUSH opcodes used by the smart contract to branch are identified and linked to their respective JUMPDEST. Once the PUSH and JUMPDEST are linked, we can obfuscate our code. To do so, the contract object is passed to a newly instantiated Obfuscator object and opcodes patterns present in the contract's opcode are identified and obfuscated. Those procedures are shown in Figure 4.2.1.
On obfuscation, some new opcodes are added, thus the PC of the existing opcodes are off, and are updated to adjust to the amount of bytes added or removed by the obfuscation.

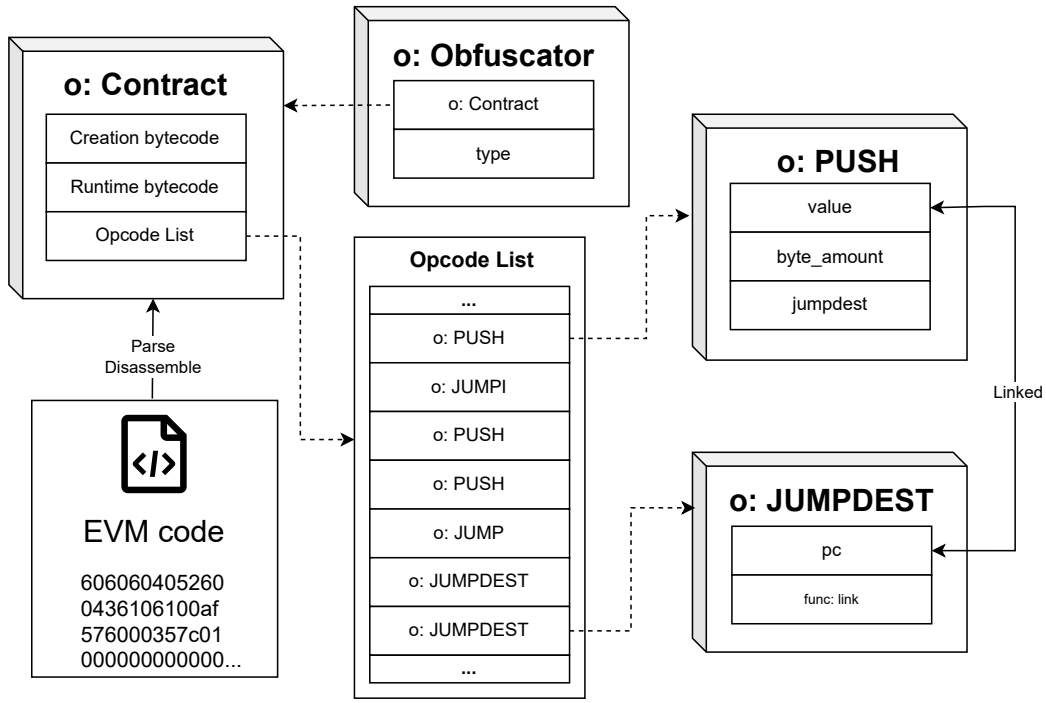

Figure 4.2.1: Data Workflow of a raw EVM Bytecode text file to a Contract object

4.3 Implementation

4.3.1 EVM Obfuscation Methods

Inspired by obfuscation methods presented in the Background chapter, here are five EVM specific obfuscation methods. Jumpdest Spammer, Control Flow Graph Spam-

mer, Jump Address Transformer and Function Signature Transformer aims at disturbing the Control Flow of the smart contract. While the ADD Opcode Stack Manipulation method's goal is to affect the layout of the smart contract at decompilation.

Jumpdest Spammer

The `spam_jumpdest()` technique falls under the category of Control Flow Obfuscation. It consists of inserting a defined amount of `JUMPDEST` opcodes in the smart contract's bytecode. This straightforward obfuscation method makes decompilers create false paths and nodes in the reconstruction of the smart contract's CFG.
The function has been designed to avoid inserting `JUMPDEST` opcodes close to the metadata section of the bytecode, thereby minimizing the chances of corrupting the contract's metadata.

Control Flow Graph Spammer

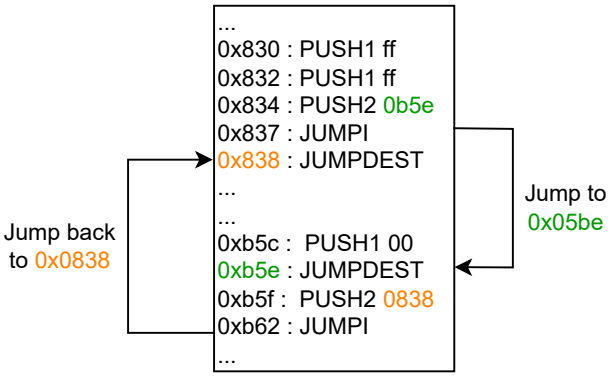

Figure 4.3.1: CFG Spammer principle

The objective of this method is to insert 'pseudo-paths' within the control flow graph (CFG) of a contract. These pseudo-paths exist alongside genuine paths but are not executed in normal program runs. This makes it challenging for an attacker to distinguish between real and fake paths, thereby increasing the complexity of both static and dynamic analysis.
In EVM bytecode, `PUSH` places an address on the stack, which `JUMP` and `JUMPI` use as a target for unconditional and conditional jumps, respectively. `JUMPDEST` marks these valid target addresses in the bytecode, guiding the control flow. Together, these opcodes dynamically alter the program counter and shape the contract's execution path.

The `cfg_spammer()` leverages the Jumpdest Spammer method to achieve control-flow graph obfuscation. This technique aims to strengthen the contract's resilience against both static and dynamic analysis while maintaining its original behavior.
The function achieves this obfuscation by injecting two sets of opcodes referred to as the "Top Sequence" and "Bottom Sequence" into two random `JUMPDEST` locations created by the Jumpdest Spammer method within the contract's bytecode.
The Top Sequence comprises two PUSH opcodes to prevent stack underflow, followed by a PUSH, the value of which is the PC destination of the following `JUMPI` opcode. The final `JUMPDEST` opcode of the Top Sequence is the final destination of the "loop back" : where the Bot Sequence's `JUMPI` opcode jumps to.
Similarly, the Bottom Sequence includes a PUSH opcode initialized with the value "00" before the `JUMPDEST` opcode, to prevent jumping if the execution is not coming from the Top Sequence.
Another `PUSH` opcode pushes the Top Sequence's `JUMPDEST` PC to the stack, which the final `JUMPI` opcode will jump to.
The purpose of this arrangement is to create a seemingly complex but non-altering loop within the CFG, thereby adding a layer of obfuscation.

This method challenges traditional static analysis techniques, which may be confounded by the additional paths in the CFG. Dynamic analysis is also complicated as the inserted sequences are executed at runtime but do not alter the intended functionality of the contract.

Jump Address Transformer

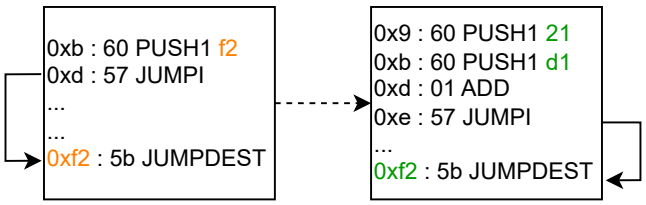

Figure 4.3.2: `JUMPI` address obfuscation `with jump_address_transformer()`

The `jump_address_transformer()` function represents a form of Address Obfuscation aimed at disrupting the natural control flow graph (CFG) of an Ethereum smart contract. This technique specifically targets the `PUSH` and `JUMPI` opcodes, which are essential for conditional jumps in EVM bytecode.
The PUSH opcode places an address onto the stack, which `JUMPI` consumes to perform

a conditional jump. Altering these particular opcodes can have an impact on the CFG without changing the contract's behavior.

The `jump_address_transformer()` function identifies sequences in the contract's opcode list where a PUSH opcode is immediately followed by a `JUMPI`. Upon detection, the function generates a random value that is less than the original jump address. This random value is then used to create a new PUSH opcode, which is inserted before the original PUSH.
The original PUSH is linked to the new one, thus, the lower value of the new PUSH is subtracted to the value of the original PUSH. Finally, an ADD opcode is inserted after the original PUSH, effectively making the new and original PUSH opcodes sum up to the original jump address.

As shown in Figure 4.3.2 the sum of the hexadecimal value `0x21` and `0xD1` is `0xF2`, which will make the `JUMPI` instruction jump to the `0xF2` address.

Function Signature Transformer

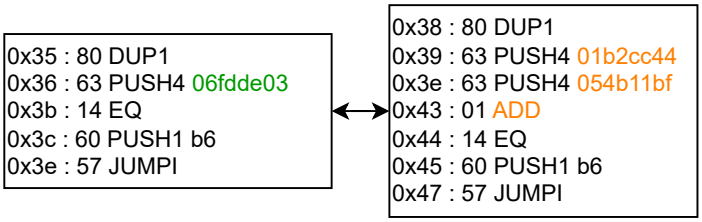

Figure 4.3.3: Obfuscation of function selector's 4-bytes identifier

The `func_sig_transformer()` function is obfuscating the smart contract's function selectors. These selectors, represented as 4-byte values, serve as the entry points for each contract function. The goal of this obfuscation method is to confuse the decompilers when they try to identify the function name using some online services like `4bytes.directory`[4] or `openchain.xyz`[35] which link 4-bytes function identifier to the name and parameters of a function, thus helping to understand what the function is used for.
The Function Signature Transformer method works by first extracting the original function selector 4-bytes value. Then, a random function signature (`rd_func_sig`) smaller than the original is then generated. Finally, the new random lower 4-bytes value is subtracted from the original 4-bytes signature.
Following this, the original `PUSH` opcode is updated with the subtracted 4-bytes value, a new `PUSH` which value is the random lower 4-bytes and an `ADD` opcode are inserted into the sequence. These newly inserted opcodes ensure that the original function selector

is reconstructed at runtime, thus maintaining the contract's functionality.
As illustrated in Figure 4.3.3, "`0x06FDDE03`" 4-bytes function signature is equal to the sum of "`0x01B2CC44`" and "`0x054B11BF`"

ADD Opcode Stack Manipulation

Rather than directly performing the addition, this obfuscation method replaces every few `ADD` opcodes with a sequence: `[DUP2, PUSH1 00, SUB, SUB, SWAP1, POP, NOT, PUSH1 01, ADD]`. This sequence is designed to manipulate the stack, positioning the two values intended for summation appropriately for the final ADD operation. The introduction of operations like duplication, subtraction, swapping, and bitwise NOT serves to shuffle and modify the stack's contents before the actual addition. Thus, at decompilation, every obfuscated ADD will output more operation than `a+b`

Chapter 5

Experimental Methodology

5.1 Smart Contract Dataset

Contract	Bytecode Length	Read	Write	Usage	ERC Standard
BAYCNFT	36768	21	16	NFT	ERC-721
CentralizedStablecoin	25838	4	6	Stablecoin	ERC-20
CloneXMintvial	25252	10	19	NFT	ERC-1155
CloneXNFT	18746	15	12	NFT	ERC-721
CloneXRandomizer	12822	1	7	NFT	None
Lending	21300	13	6	DeFi	None
RewardToken	11822	6	3	Token	ERC-20
Sablier	12406	4	3	DeFi	None
Seaport	52248	6	11	Marketplace	None
Staking	10110	11	3	DeFi	None
UniswapFactory	27980	5	3	DeFi	None
UniswapRouter	44796	7	17	DeFi	None
USDC	5778	24	26	Stablecoin	ERC-20
USDT	23800	19	13	Stablecoin	ERC-20
WETH	7008	6	5	Token	ERC-20

Table 5.1.1: Smart Contracts Dataset showcasing bytecode length by character count, number of Read and Write functions, their usage and the ERC standard they implement if there is one

Table 5.1.1 showcases a diverse set of 15 smart contracts that were gathered to analyze the efficiency and efficacy of the obfuscator written in the context of the thesis. Most of these contracts are well known in the Ethereum ecosystem, and have been used in thousands to millions of individual transactions.

Uniswap Factory: The Uniswap Factory is a smart contract on the Ethereum blockchain responsible for deploying and managing individual Uniswap child-contract. It serves as a centralized registry for all Uniswap child-contracts, allowing users and other smart contracts to easily discover and interact with them.

Uniswap Router: The Uniswap Router is a smart contract that acts as an interface for users and other smart contracts to interact with Uniswap child-contracts. It provides a set of functions for executing a set of functions through one or multiple child-contracts.

USDT: Tether (USDT) is an ERC-20 token called, a stablecoin, issued by Tether Limited, pegged to the value of the U.S. dollar. USDT is commonly used as a trading pair.

USDC: USDC is an ERC-20 token, a stablecoin, issued by Circle, also pegged to the value of the U.S. dollar.

BAYC NFT: The Bored Ape Yacht Club token is an ERC-721 Non-Fungible Token (NFT) Ownership of a Bored Ape grants access to various virtual and real-world perks, such as exclusive events and digital assets. The project has gained significant attention for its blend of digital art, community engagement, and utility-driven features.

CloneX NFT: CloneX is an NFT collection consisting of unique, algorithmically generated avatars minted as ERC-721 tokens on the Ethereum blockchain. Created by the digital art studio RTFKT, CloneX avatars serve not just as collectible digital art but also as access keys to various virtual and real-world experiences.

CloneX Mintvial: It is a specialized NFT that serves as a gateway to mint a CloneX avatar. It is an ERC-721 token that, when activated, allows the holder to generate and claim a unique CloneX NFT. The Mintvial separates the act of acquiring the minting rights from the actual minting of the CloneX avatar.

CloneX Randomizer: It is a specialized NFT that serves as attibute generator for CloneX minting process. It leverages Chainlink on Ethereum Mainnet, but has been modified in the local test environment which cannot use Chainlink.

Seaport: Seaport holds the core logic of the Opensea marketplace. It manages the listing, bidding, and executing sales of NFTs.

Sablier: Sablier enables continuous, real-time payments. Sablier allows users to lock

up funds and distribute them over a specified time period, providing a novel way to handle payroll, subscriptions, or any time-based financial agreements.

WETH: WETH stands for ”Wrapped Ether,” a smart contract that represents the native (ETH) in an ERC-20 compliant format. Ethereum’s native currency, Ether, does not conform to its own ERC-20 standard, which makes it incompatible with smart contracts leveraging ERC-20 standard.

Lending: This smart contract is an ERC-20 token lending platform smart contract written in Solidity based on the Aave protocol, taken from defi-minimal[15] public repository.

Staking: This contract is a minimal implementation of a staking protocol inspired by Synthetix, taken from defi-minimal[15] public repository.

CentralizedStablecoin: It a centralized stablecoin contract inspired by USDC, taken from defi-minimal[15] public repository.

RewardToken: RewardToken is a smart contract based on the OpenZeppelin ERC-20 contract, which is a well tested and community reviewed implementation of the ERC-20 standard, used alongside defi-minimal contracts.

5.2 Experiments and Metrics

To create a coherent dataset, the obfuscator presented in this thesis was used to produce 6 versions of obfuscated EVM bytecode of the 15 smart contracts comprising the contracts dataset, totaling 90 files to analyze. These versions can be categorized as follow:

The obfuscation process involved creating four Single-Method obfuscation versions, namely:

1. CFG Spammer (CFGSPAM)
2. Jumpdest Spammer (JUMPDESTSPAM)
3. Function Signature Transformer (FUNCSELEC)
4. ADD Opcode Stack Manipulation (ADD)

In addition to these, a FULL obfuscation version was created, which is a combination

of all the obfuscation methods applied to the EVM bytecode. One ORIGINAL version, which is the original contract as it is deployed on Ethereum Mainnet.

To evaluate the **Correctness** of an obfuscated smart contract, a comparison between the state of the original and the obfuscated smart contract is performed. See Section 6.1.

To measure the **Efficacy** of the obfuscator, we computed two metrics for both the original and fully obfuscated EVM bytecode of each smart contract that comprise the dataset. The first is the Cyclomatic Complexity derived from the Control Flow Graph (CFG). The second is the Halstead's Effort complexity measures which offer a different angle by focusing on the understandability and complexity of the decompiled EVM bytecode. See Section 6.2.

To evaluate the **Efficiency** of the obfuscator we perform a Gas Cost changes evaluation. It evaluates the computational (and economical) costs associated with the obfuscation. See Section 6.3.

5.2.1 Correctness: On-Chain Execution Methodology

To evaluate the correct execution of an obfuscated smart contract, a comparison between the state of the original smart contract and the obfuscated one is performed.

If the set of transactions of the test scenario includes the minting of 20 ERC-20 "X" Token by Address "A" and the transfer of them to Address "B", we check for both original and obfuscated if the `balanceOf()` function returns 20 after minting for Address "A" and 20 after `transfer()` for Address "B".

5.2.2 Efficacy: Control Flow Graph Evaluation

In order to analyze the Control Flow Graph of the obfuscated EVM bytecode, we use a set of metrics that provide an understanding of its structure and its complexity : Nodes, Edges and Cyclomatic Complexity,

The **Node** (N) Count metric involves counting the total number of nodes present in the CFG. Each node corresponds to a basic block in the contract's execution. A surge in the node count suggests an increase of the contract's complexity. Whereas, a reduction in

the node count, especially when compared to the original CFG, might indicate that the obfuscation techniques have introduced elements that confuse the CFG analysis tools. Such change would mean that the obfuscation is effectively thwarting the tools from generating a complete and accurate representation of the contract's control flow.

The **Edge** (E) Count focuses on the total number of edges that link the nodes within the CFG. These edges are representing the transitions between basic blocks in the smart contract execution flow. An increased edge count shows a greater number of possible execution paths. Inversely, a decrease in edge count compared to the original smart contract's CFG suggests the ability for the obfuscator to mislead the CFG analysis tools.

The McCabe's **Cyclomatic Complexity** [31] gives an understanding of a program's complexity. It is computed using the formula `E-N+2P`, where `E` stands for the number of edges, `N` the number of nodes, and P the number of connected components (usually `1` for a single program). A higher cyclomatic complexity suggests a more complex control flow, demanding more scrutiny during reverse engineering. The higher the Cyclomatic Complexity, the better the obfuscation. Inversely, since the CFG generating tools can output CFGs with less Nodes and Edges in an original than an obfuscated contract, the Cyclomatic Complexity could be lower too. In this case, a lower Cyclomatic Complexity translate a loss of information about the smart contract's flow, thus a better obfuscation. Summarized, a significantly higher or lower Cyclomatic Complexity highlights a good obfuscation, however, a minor change in the Cyclomatic Complexity shows a poor obfuscation.

5.2.3 Efficacy: Decompiled EVM Bytecode Evaluation

To evaluate the decompiled obfuscated EVM bytecode, we leverage the Halstead complexity measures [25], which offer an insight into the understandability of a program's source code. These metrics are particularly useful in measuring the effectiveness of obfuscation techniques.

Operators are symbols that represent specific operations. In Solidity, just like in many other programming languages, there are arithmetic operators (`+`, `-`, `*`, `/`), comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`), logical operators (`&&`, `||`, `!`), and assign-

ment operators (`=`, `+=`, `-=`), among others. For instance, in the expression `(a + b)`, the `+` symbol is the operator.

Operators acts on **Operands** : They can be constants, variables, or more complex expressions. In Solidity, operands could be variables like `uint`, `address`, `bytes`, or more complex data structures like `arrays` and `structs`.

The count of unique **Operators** and **Operands** are respectively represented by `n1` and `n2`. A change in these values from the original bytecode indicate the introduction of new operations or data types due to obfuscation.

`N1` and `N2` denote the **Total Occurrences** of operators and operands. An increase in these values suggests an increase in the program's operations.

Vocabulary (`n`) is calculated as `n1 + n2`, it represents the total number of unique symbols (both operators and operands) in the program. A larger vocabulary can be indicative of a more complex program.

Program Length (`N`) is given by `N = N1 + N2`, this metric measures the total size of the program in terms of its operators and operands. A significant increase in program length after obfuscation suggest the addition of redundant or misleading operations.

Volume (`V`) is defined as `V = N * log2(n)`, the volume represents the size of the implementation of an algorithm. A higher volume post-obfuscation can suggest that the obfuscation has added complexity, making the bytecode more intricate.

Difficulty (`D`) is calculated with `D = (n1/2) * (N2/n2)`, this metric shows the challenge associated with writing or understanding the program. A higher difficulty score post-obfuscation indicates that the obfuscated bytecode is more complex and harder to understand.

Effort (`E`) given by `E = D * V`, measures the mental effort required to develop or understand the program. A spike in this metric post-obfuscation can suggest that more work is needed to understand or reverse-engineer the obfuscated bytecode, indicating the efficiency of the obfuscation techniques employed. Inversely, a significant decrease of the Effort value between the original and the obfuscated version translates a loss of informations in the decompiled source code, so, an effective obfuscation.

Because the Effort value aggregates every Halstead's metrics, it will be used as the primary metric for obfuscation efficacy evaluation.

5.2.4 Efficiency: Gas Cost Changes Evaluation

Through the test scenario, for each version of the same smart contract, each transaction performed, the gas usage of the transactions is recorded. A script reads the gas usage data recorded, computes the percentage change for each function, and finally calculates the average percentage change across all functions.

Evaluation		*Metrics*
Efficacy	**Decompiled Source Code**	Halstead's Effort
Efficacy	**CFG**	McCabe's Cyclomatic Complexity
Efficiency	**Gas Cost Changes**	Transaction's Gas
Correctness	**On-Chain Execution**	Contract State

Figure 5.2.1: Table summarizing the Metrics recorded for each Experiments performed

5.3 Implementation Details

The CFG and decompiled source code analyzed were generated by Heimdall[5], Dedaub[18], and Ethersolve[16]. Some other tools, while powerful, did not accept raw EVM bytecode as input. Additionally, it is mandatory to have tools that implement the latest EVM specifications.

5.3.1 On-Chain Execution & Gas Cost Changes Analysis

To evaluate the correct execution and the gas cost changes of the Smart Contracts present in the dataset, a framework was set up.

This framework, a testing environment, leverages Hardhat[20], Ethers.js[32] and Foundry's Cast[36].

Hardhat was used to set up a development environment and a local Ethereum node to deploy original and obfuscated smart contracts from the dataset. Ethers.js is a javascript library for blockchain development and was used to interact with the deployed smart contracts on the local node.

Foundry's Cast command-line tool was used to perform Ethereum RPC calls on the local Hardhat's node. Due to a bug in the gas estimation of raw transactions performed with Ethers.js on Hardhat's local node, it was necessary to use Cast to send the raw transactions that deploy smart contracts with the creation code only.

The six versions of each Smart Contracts that comprises the dataset were deployed on the local node.

A test scenario is written as a Hardhat script for every version of each smart contract that comprises the dataset. Each test scenario comprises a set of arbitrary transactions designed to replicate real-world usage while covering the largest possible amount of functions of the smart contract tested. The test scenario purpose is to ensure that the obfuscated smart contract behave the same way as the original smart contract. The Hardhat script is sending a set of transactions replicating real-world usage. e.g. Mint, Approve and Transfer of ERC-721 or ERC-20, Staking and Un-Staking or Swap and Transfer of ERC-20, etc.

Once every scenario is run, the states of the obfuscated contracts are compared to the state of the original one, to check if the transactions were executed correctly: the same way as the original smart contract. Every transaction sent through the test scenario gets their gas usage recorded in order to compare the gas fees between the original and obfuscated versions of a smart contract.

5.3.2 Control Flow Graph Generation

The generation of Control Flow Graphs (CFG) was made using EtherSolve and Heimdall. These tools parse the EVM bytecode, identify basic blocks, establish the potential transitions between them and create a visual representation of the contract's control flow.

Heimdall requires the analyzed contracts to be deployed and take its address as input while EtherSolve uses the raw bytecode file. Custom Python scripts were written to adapt to the specific requirements of both tools, to have more control and automate the CFG generation process.

5.3.3 Decompilation of Obfuscated EVM bytecode

The decompilation of obfuscated EVM bytecode was done using Dedaub and Heimdall. These tools are designed to decompile EVM bytecode and translate it back into a higher-level representation. The same way as for CFG Generation, multiple Python scripts were written to automate the decompilation of the 90 obfuscated EVM bytecode from the 15 dataset's contracts.

Chapter 6

Experimental Results

This chapter describes the experimental methodology, dataset, and evaluation metrics used in the thesis. The dataset consists of 15 diverse smart contracts. Six versions of obfuscated EVM bytecode were generated for each contract, resulting in 90 files. The experiments evaluates Correctness, Efficacy, and Efficiency of the obfuscation methods proposed.

6.1 On-Chain Execution Correctness

EXPERIMENTAL RESULTS	On Chain Execution					
	ORIGINAL	FULL	CFGSPAM	JUMPDEST SPAM	FUNCSELEC	ADD
BAYCNFT	OK	OK	OK	OK	OK	OK
CentralizedStablecoin	OK	OK	OK	OK	OK	OK
CloneXMintvial	OK	OK	OK	OK	OK	OK
CloneXNFT	OK	OK	OK	OK	OK	OK
CloneXRandomizer	OK	OK	OK	OK	OK	OK
Lending	OK	OK	OK	OK	OK	OK
RewardToken	OK	OK	OK	OK	OK	OK
Sablier	OK	OK	OK	OK	OK	OK
Seaport	OK	NOT OK	NOT OK	NOT OK	NOT OK	NOT OK
Staking	OK	OK	OK	OK	OK	OK
UniswapFactory	OK	NOT OK	NOT OK	NOT OK	NOT OK	NOT OK
UniswapRouter	OK	NOT OK	NOT OK	NOT OK	NOT OK	NOT OK
USDC	OK	OK	OK	OK	OK	OK
USDT	OK	OK	OK	OK	OK	OK
WETH	OK	OK	OK	OK	OK	OK

Figure 6.1.1: On-Chain execution outcomes of dataset's Smart Contract under different Obfuscation methods

Table 6.1.1 provides an overview of the integrity of the obfuscated smart contracts when executed on-chain. The ”OK” status suggests that the obfuscation techniques preserve the functionality of the original contracts. It is a positive indicator on the correctness aspect of the RQ1 research question, as it shows that the obfuscation methods do not compromise the integrity of the smart contracts.

However, it’s worth noting that for the ”Seaport”, ”UniswapFactory” and ”UniswapRouter” contracts, the obfuscation methods seem to break the contract, indicated by the ”NOT OK” status. The issue with UniswapFactory seems to be related with the deployments of new ”trading pairs” contracts triggered by CREATE2 opcode (The offset of CREATE2 opcode is not adjusted to the amount of bytecode added through obfuscation).

6.2 Obfuscation Efficacy

The objective of measuring the Cyclomatic Complexity and the Halstead’s Effort is to assess the efficacy of the obfuscation methods implemented in this thesis. An effective obfuscation would translate in a high variation of the cylomatic complexity, thus a harder time to understand a smart contract’s CFG. It would also mean a higher Effort value, which hint a harder effort to understand the decompiled source code of an obfuscated smart contract.

Cyclomatic Complexity

Table 6.2.1 shows the Cyclomatic Complexity computed from the CFG generated by EtherSolve[16], and Heimdall[5] of each Smart Contract of the dataset before (ORIGINAL) and after obfuscation (FULL OBF). Uniswap related contracts and Seaport contract are not working after obfuscation, and thus have been omitted from evaluation.

Several takeaways can be drawn from Table 6.2.1. For most contract’s CFGs analyzed by Heimdall, the Cyclomatic Complexity drastically is reduced after obfuscation, as seen in the FULL OBF columns compared to the ORIGINAL columns. It indicates that the obfuscation is highly effective in complicating the control flow.

EtherSolve isn’t able to analyze 8 out of the 12 fully obfuscated smart contract, which leads to a generated CFG missing almost the entire bytecode, those evaluation are marked as ”FAIL”

EXPERIMENTAL RESULTS	CFG Cyclomatic Complexity					
	ETHERSOLVE			HEIMDALL		
	ORIGINAL	FULL OBF	% Changes	ORIGINAL	FULL OBF	% Changes
BAYCNFT	284	84	-70%	101	162	60%
CentralizedStablecoin	177	FAIL	FAIL	95	307	223%
CloneXMintvial	328	63	-81%	123	273	122%
CloneXNFT	233	FAIL	FAIL	86	CRASHED	CRASHED
CloneXRandomizer	79	FAIL	FAIL	31	120	287%
Lending	210	FAIL	FAIL	175	133	-24%
RewardToken	81	FAIL	FAIL	35	85	143%
Sablier	112	FAIL	FAIL	84	133	58%
Staking	94	FAIL	FAIL	67	108	61%
USDC	253	FAIL	FAIL	124	183	48%
USDT	146	68	-53%	27	52	93%
WETH	41	25	-39%	11	123	1018%

Figure 6.2.1: CFG's Cyclomatic Complexity comparative table between a Smart Contract and its obfuscated version

HEIMDALL	ORIGINAL				FULL				CFGSPAM				JUMPDESTSPAM				FUNCSELEC				ADD			
	N	E	P	CC	N	E	P	CC	N	E	P	CC	N	E	P	CC	N	E	P	CC	N	E	P	CC
RewardToken	68	100	1	**35**	69	152	1	**85**	82	194	1	**114**	68	100	1	**34**	68	100	1	**34**	62	82	1	**22**
WETH	76	84	1	**11**	91	212	1	**123**	88	199	1	**113**	76	84	1	**10**	76	84	1	**10**	76	84	1	**10**

ETHERSOLVE	ORIGINAL				FULL				CFGSPAM				JUMPDESTSPAM				FUNCSELEC				ADD			
	N	E	P	CC	N	E	P	CC	N	E	P	CC	N	E	P	CC	N	E	P	CC	N	E	P	CC
RewardToken	261	341	1	**81**	FAIL				FAIL				279	359	1	**82**	261	341	1	**82**	261	341	1	**82**
WETH	103	142	1	**41**	82	84	12	**25**	84	74	12	**14**	120	159	1	**41**	103	142	1	**41**	103	142	1	**41**

Figure 6.2.2: WETH and RewardToken individual CFG analysis

The failed Heimdall's analysis suggest that some obfuscation techniques are effective to the point of breaking the analysis tools, this claim is supported by the fact that the On-Chain Execution clearly shows that the contract still works as intended after obfuscation.

There are inconsistencies between the results from EtherSolve and Heimdall. For instance, the WETH original contract has a Cyclomatic Complexity of 41 in EtherSolve compared to 11 with Heimdall. The choice of tool significantly impact the CFG analysis.

Table 6.2.1 alone does not confirm the effectiveness of the obfuscation techniques. However, it suggests that the obfuscation is effective in complicating the control flow of the smart contracts modulo the fact that we can doubt the reliability of Heimdall and EtherSolve on generating a correct CFG, due to the disparity in the results.

Table 6.2.2 shows the McCabe's measures for the RewardToken and WETH smart con-

tracts across the different obfuscation methods implemented. In red, the results highlights the reduced impact of the following *Jumpdest Spammer (JUMPDESTSPAM)*, *Function Signature Transformer (FUNCSELEC)* and *ADD Opcode Stack Manipulation (ADD)* techniques in increasing the complexity of the CFG. In green, inversely, the Cyclomatic Complxity increase when the CFG Spammer (CFGSPAM) method is used in the obfuscation process. Therefore, the FULL obfuscation which combines every technique, also increases the CFG complexity of the obfuscated contract.

Halstead's Effort measure

EXPERIMENTAL RESULTS	DECOMPILER Effort					
	HEIMDALL			DEDAUB		
	ORIGINAL	FULL OBF	% Changes	ORIGINAL	FULL OBF	% Changes
BAYCNFT	2909122	FAIL	FAIL	4625176	10049645	117%
CentralizedStablecoin	389224	1088562	180%	180030	2840270	1478%
CloneXMintvial	1088972	287566	-74%	3579365	14113801	294%
CloneXNFT	636462	FAIL	FAIL	946678	3579903	278%
CloneXRandomizer	246489	FAIL	FAIL	320552	2909549	808%
Lending	425966	FAIL	FAIL	370909	3585075	867%
RewardToken	89084	FAIL	FAIL	67152	4958072	7283%
Sablier	761891	FAIL	FAIL	2799354	5304301	89%
Staking	103046	1205912	1070%	98143	681206	594%
USDC	1335314	FAIL	FAIL	1072067	10825028	910%
USDT	169942	FAIL	FAIL	192770	942190	389%
WETH	43796	FAIL	FAIL	40536	829109	1945%

Figure 6.2.3: Halstead's Effort complexity measure comparative table between a Smart Contract and its obfuscated version

Table 6.2.3 shows the Halstead's Effort measure computed from the decompiled source code generated by Dedaub[18], and Heimdall[5] of each Smart Contract of the dataset before (ORIGINAL) and after obfuscation (FULL OBF).

Only the Effort is showcased in Table 6.2.1 as it aggregate the data gathered for the analysis.

According to Table 6.2.1 result, 9 out of the 12 contracts analyzed by Heimdall, the decompilation fails. CloneXMintvial decompiled source code shows a major reduction in the Effort value between the original and the obfuscated version, a significant reduction of the Effort value translates a loss of informations, a hardher time to understand the decompiled source code, thus : An effective obfuscation. For the two remaining contract, the analysis shows a great increase in the Effort value.

Every contracts decompiled by Dedaub, shows a significant rise in the Effort value, from 89% to 7283% increase with an average 1254% increase. It indicates that the obfuscation is highly effective in complicating the decompiled source code.

Difference in Effort value between Heimdall and Dedaub is due to the way the decompiled source code is produced. Dedaub produce way more operators and operands than Heimdall.

Table 6.2.3 does not shows the impact of the Function Signature Transformer obfuscation methods. But when applied, every decompiled source code produced by either Dedaub or Heimdall showcase wrong or no human-readable representation of the functions.

WETH DEDAUB	*n1*	*n2*	*N1*	*N2*	*Vocab*	*Program Length*	*Volume*	*Difficulty*	*Effort*	*% Change*
ORIGINAL	**19**	**20**	**165**	**69**	**39**	**234**	**1237**	**33**	**40536**	**+0%**
FULL	21	36	551	475	57	1026	5985	139	**829109**	**+1945%**
ADD	19	20	223	159	39	382	2019	76	**152487**	**+276%**
FUNCSELEC	19	23	185	90	42	275	1483	37	55125	+36%
CFGSPAM	17	29	115	111	46	226	1248	32	40613	+0%
JUMPDESTSPAM	19	20	165	69	39	234	1237	33	40536	+0%

Figure 6.2.4: WETH contract Dedaub's CFG Analysis Results on each Obfuscation techniques

REWARDTOKEN DEDAUB	*n1*	*n2*	*N1*	*N2*	*Vocab*	*Program Length*	*Volume*	*Difficulty*	*Effort*	*% Change*
ORIGINAL	**20**	**19**	**199**	**85**	**39**	**284**	**1501**	**45**	**67152**	**+0%**
FULL	21	76	1430	1724	97	3154	20816	238	**4958072**	**+7283%**
ADD	21	48	496	561	69	1057	6457	123	**792359**	**+1080%**
FUNCSELEC	20	19	199	87	39	286	1512	46	69217	+3%
CFGSPAM	17	25	221	158	42	379	2044	54	109787	+63%
JUMPDESTSPAM	20	19	202	86	39	288	1522	45	68899	+3%

Figure 6.2.5: RewardToken contract Dedaub's CFG Analysis Results on each Obfuscation techniques

The Tables 6.2.5 6.2.4 shows a great effectiveness of the *ADD Opcode Stack Manipulation (ADD)* obfuscation techniques on Dedaub's decompiler. Inversely, as Table 6.2.6 and 6.2.7 show : *CFG Spammer (CFGSPAM)* and *Function Signature Transformer (FUNCSELEC)* methods have great impact on Heimdall's decompiler while *ADD Opcode Stack Manipulation* and *Jumpdest Spammer (JUMPDESTSPAM)* not.

WETH HEIMDALL	n1	n2	N1	N2	Vocab	Program Length	Volume	Difficulty	Effort	% Change
ORIGINAL	**11**	**23**	**106**	**144**	**34**	**250**	**1272**	**34**	**43796**	**+0%**
FULL	1	4	1	4	5	5	12	1	**6**	**-100%**
ADD	12	26	109	150	38	259	1359	35	47050	+7%
FUNCSELEC	1	4	1	4	5	5	12	1	**6**	**-100%**
CFGSPAM	13	37	8920	12356	50	21276	120079	2171	**260648634**	**+595043%**
JUMPDESTSPAM	11	24	106	145	35	251	1287	33	42781	-2%

Figure 6.2.6: WETH contract Heimdall's CFG Analysis Results on each Obfuscation techniques

REWARDTOKEN HEIMDALL	n1	n2	N1	N2	Vocab	Program Length	Volume	Difficulty	Effort	% Change
ORIGINAL	**12**	**31**	**191**	**211**	**43**	**402**	**2181**	**41**	**89084**	**+0%**
FULL	12	15	39	51	27	90	428	20	**8730**	**-90%**
ADD	13	31	216	216	44	432	2358	45	106816	+20%
FUNCSELEC	10	17	22	42	27	64	304	12	**3759**	**-96%**
CFGSPAM	13	30	546	1022	43	1568	8508	221	**1884040**	**+2015%**
JUMPDESTSPAM	12	31	191	210	43	401	2176	41	88441	-1%

Figure 6.2.7: RewardToken contract Heimdall 's CFG Analysis Results on each Obfuscation techniques

Observation

The results from tables 6.2.1 and 6.2.3 highlight the effectiveness of the obfuscation methods due to the difficulty encountered by Heimdall and Dedaub to produce clear decompiled source code; Alongisde Heimdall and EtherSolve struggling to produce coherent CFG. On top of that, Tables 6.2.5, 6.2.4, 6.2.6 and 6.2.7 highlights that depending on the decompiler used, certain obfuscation techniques are more effective than others.

Even though the obfuscation methods shows great effectiveness, the results do not assess the efficiency of the obfuscation techniques developed. While the On-Chain Execution Correctness confirm that an obfuscated smart contracts works as well as its original version, the obfuscation techniques applied might increase the gas cost associated with executing the contract's functions economically expensive.

EXPERIMENTAL RESULTS	Gas Cost Changes
BAYCNFT	0.13 %
CentralizedStablecoin	0.23 %
CloneXMintvial	0.20 %
CloneXNFT	0.15 %
CloneXRandomizer	0.20 %
Lending	0.20 %
RewardToken	0.30 %
Sablier	0.24 %
Staking	**5%**
USDC	0.24 %
USDT	**5%**
WETH	0.20 %

Figure 6.3.1: Percentage change of Gas Cost of main functions between Smart Contracts and their obfuscated version

6.3 Obfuscation Efficiency

Gas Cost Changes

Table 6.3.1 focuses on the efficiency by presenting the percentage change in gas costs between the original and obfuscated versions of the smart contracts. Most contracts show a minimal increase in gas costs (ranging from 0.13% to 0.30%). This minimal increase suggests that the obfuscation techniques can be applied without significantly affecting the economic viability of interacting with the contract, highlighting an efficient obfuscation.

UniswapFactory, UniswapRouter and Seaport contracts have not been analyzed since the contracts are not working after On-Chain Execution Correctness evaluation.

Observation

After reviewing the Gas Cost Changes results, the spike of percentage change for USDT and Staking contracts seem to be linked to few functions which appear to cost a way larger amount of gas (15% to 30%) while the other function's gas cost range between 0.1% and 0.30% increase.

6.4 Summary

The experimental results provide evidence to answer both RQ2 : ”How effective are obfuscation methods in throttling smart contract analysis tools ?” and RQ3 : ”How efficient are obfuscation methods in throttling smart contract analysis tools?” research questions.

For RQ2, the results indicates a great level of efficacy. The Cyclomatic Complexity and Halstead’s Effort metrics show a significant increase in complexity for most of the obfuscated contracts, making it difficult for analysis tools like EtherSolve and Heimdall to produce coherent Control Flow Graphs or Decompiled Source code. This suggests that the obfuscation methods are highly effective in complicating the tasks for smart contract analysis tools.

As for RQ3, which focuses on the efficiency of the obfuscation methods introduced in this thesis, the results are nuanced. While the ”OK” statuses in the On-Chain Execution analysis indicate that the obfuscation techniques preserve the functionalities of the smart contracts, the gas cost changes are variable. For most contracts, the increase in gas costs is marginal, suggesting that the obfuscation methods are economically feasible. However, contracts like ”Seaport, ”UniswapFactory,” and ”UniswapRouter” indicate that the obfuscation methods may compromise the functionality in specific cases, requiring further fine-tuning. Therefore, while the obfuscation methods are generally efficient, their applicability may be contract-specific and could entail additional computational and economic costs.

Chapter 7

Discussion

7.1 Reflections

The main contribution of this Thesis is the design and development of EVeilM, an EVM Bytecode Obfuscator aiming at protecting smart contracts against reverse engineering. This has been possible by an in-depth exploration of Ethereum's architecture, focusing on the complexity of smart contract deployment and execution in the Ethereum Virtual Machine.

A series of obfuscation techniques were developed, some directly inspired from existing methods, while others were tailored specifically for EVM smart contracts. From basic opcode substitutions to more complex transformations of control flow and data representation. The whole challenge was to develop effective obfuscation while preserving functional integrity of the smart contract and keep the gas costs under control. Working on this subject was challenging, as I started on this journey with limited knowledge on the technical aspect of Ethereum and Solidity. One of the main challenges was to develop from the ground up a solution to a problem I knew nothing about. It took a long time to design an architecture for the software, finding the right tools to understand, iterate, develop or adapt and finally test the existing and new obfuscation techniques. As blockchain is a relatively new domain, there is still a lack of documentation about some key aspects of EVM that I needed to develop the obfuscator. Even though I thought that I found the right architecture to develop the obfuscator, I discovered new subtleties on the go about EVM bytecode that would break the software.

7.2 Limitations

One of the main limitations of the obfuscator is the implementation itself. Even though working at the bytecode level is the actual subject of the thesis, I am not sure if it is the best way to apply obfuscation. The reason is that the different versions of the Solidity compiler are inconsistent in giving reliable patterns to delimit creation bytecode with runtime bytecode; or runtime bytecode and contract's metadata at the end of the bytecode. This variability makes it challenging to develop a one-size-fits-all obfuscation approach that is consistently effective across different compiler outputs.

Another limitation concerns smart contracts which create new smart contract themselves, like UniswapFactory which create UniswapPair for swapping ERC-20 tokens. Those ”parent” smart contract use the ”CREATE2” opcode to create/deploy new contract on the blockchain. Unfortunately, the current architecture of the obfuscator made tricky the obfuscation of those ”parent” smart contract. Thus, it is not supported.

There is no formal proof to guarantee that the obfuscated smart contracts maintain exact functional equivalence with their original version, or that the obfuscator does not introduce new vulnerabilities. The scope of this thesis does not extend to formally verifying these aspects, leaving open the possibility of alterations in contract functionalities or new security vulnerabilities.

About the efficiency evaluation of the obfuscation methods, one notable limitation in assessing it comes from the reliability of the evaluation tools like Heimdall, EtherSolve, and Dedaub. Each of these tools has its own set of constraints and potential inaccuracies, which can influence the outcome of the evaluation process.

It is important to note that this thesis primarily focuses on Solidity-based smart contracts, and as such, smart contracts written in Vyper have not been directly tested with the developed obfuscation methods. Although the obfuscation techniques are designed to be applicable to EVM compatible smart contracts and should theoretically work with Vyper, the absence of testing on Vyper contracts is a notable limitation.

7.3 Ethics and Sustainability

The ethics considerations of using bytecode obfuscation are complex. On one hand, these techniques are meant to protect smart contracts from attacks. On the other hand,

they could also hide harmful code, leading to fraud or dangerous contracts. This raises a moral responsibility for developers and researchers to think about the wider effects of their work.

Also, for the Blockchain community, the synergy between transparency and security is important. While obfuscation increases smart contracts security from potential reverse engineering, it goes against the principle of transparency inherent to blockchain. This trade-off raises ethical considerations. On one hand, enhancing security is a great lever for protecting contracts against unintentional use. On the other hand, the decreased readability and comprehensibility of obfuscated code are not aligned with the core values of open and transparent blockchain ecosystems, potentially diminishing trust between users and developers. This situation highlights the ethical challenge of developers to carefully balance the need for improved security with preserving transparency, making sure to preserve the principles of blockchain.

Chapter 8

Conclusions

8.1 Answering Research Questions

In answering the RQ1, the thesis demonstrated the application of obfuscation methods at the EVM bytecode level through a rigorous system design. It involves adapting traditional obfuscation techniques, such as control flow obfuscation and dead code insertion alongside EVM tailored obfuscation methods like Function Signature Transformer, to suit the EVM's unique characteristics like gas. The challenge was not only in the implementation of these techniques, but also in ensuring that they do not impact the functionalities of the smart contracts. The successful obfuscation revealed by the on-chain execution tests, shows that the obfuscated smart contracts continue to work as intended. The results of the On-Chain Execution Correctness tests highlight the achievement in applying obfuscation methods while maintaining the operational integrity of the smart contracts.

About RQ2, the thesis demonstrated the effectiveness of the obfuscation methods in increasing the difficulty for smart contract analysis tools. By employing techniques that significantly raised the Cyclomatic Complexity and Halstead's Effort metrics, the obfuscated contracts presented a much higher analytical challenge. Tools like Ether-Solve, Heimdall and Dedaub struggled to generate coherent Control Flow Graphs or decompiled source code from these obfuscated contracts. This demonstrated that the obfuscation methods were highly effective in adding complexity, thus complicating the task of analyzing and understanding the smart contracts' logic.

RQ3 focused on the efficiency of the obfuscation methods: the gas cost associated with

smart contract execution. While obfuscation could increase the computational complexity, and consequently, the gas cost for contract deployment and execution, the thesis aimed to mitigate this impact. The results showed that every obfuscation methods implemented did not significantly increase the gas cost. This efficiency is important in the Ethereum ecosystem, where gas cost is a key consideration. This work found right balance between security increase through bytecode obfuscation and the need to maintain reasonable gas cost during smart contract deployment and execution.

8.2 Future Work

Some limitations of this work could be addressed in future work. An obvious continuation would be to adapt other traditional obfuscation techniques, create new ones and enhance the existing obfuscation techniques. Addressing the variability in Solidity compiler outputs (e.g. with or without optimizations) is also something to look at. Future research could also explore the application and effectiveness of the obfuscation techniques on Vyper-based contracts to fully validate their utility across different Ethereum programming languages. Another task would be the formal verification of the different obfuscation methods to ensure that they do not alter smart contract functionality or introduce new vulnerabilities. Developing frameworks to better assess obfuscation effectiveness, its impact on gas costs, and maintaining the functional integrity of complex contracts is another aspect to focus on. Also, as Ethereum evolves, obfuscation methods would need to keep up with updates of the EVM specifications.

Bibliography

[1] Cornelius C. Agbo, Qusay H. Mahmoud, and J. Mikael Eklund. "Blockchain Technology in Healthcare: A Systematic Review". In: *Healthcare* 7.2 (2019). ISSN: 2227-9032. DOI: 10.3390/healthcare7020056. URL: https://www.mdpi.com/2227-9032/7/2/56.

[2] The Solidity Authors. *Solidity*. 2023. URL: https://docs.soliditylang.org/.

[3] Boaz Barak et al. "On the (im) possibility of obfuscating programs". In: *Annual international cryptology conference*. Springer. 2001, pp. 1–18.

[4] beaconcha.in. *Ethereum Signature Database*. 2023. URL: https://www.4byte.directory/.

[5] Jon Becker. "GitHub - Jon-Becker/heimdall-rs: Heimdall is an advanced EVM smart contract toolkit specializing in bytecode analysis. — github.com". In: [Accessed 26-09-2023].

[6] Martin Becze and et al. Hudson Jameson. *Ethereum Improvement Proposals*. 2015. URL: https://eips.ethereum.org/EIPS/eip-1.

[7] Vitalik Buterin. *A CBC Casper Tutorial*. 2018. URL: https://vitalik.ca/general/2018/12/05/cbc_casper.html.

[8] Vitalik Buterin. "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform". In: *Ethereum project white paper* (2013). URL: https://ethereum.org/en/whitepaper.

[9] Vitalik Buterin. *How obfuscation can help Ethereum*. 2020. URL: https://ethresear.ch/t/how-obfuscation-can-help-ethereum/7380.

[10] Vitalik Buterin and Virgil Griffith. "Casper the Friendly Finality Gadget". In: *CoRR* abs/1710.09437 (2017). arXiv: 1710.09437. URL: http://arxiv.org/abs/1710.09437.

[11] Vitalik Buterin et al. *Combining GHOST and Casper*. 2020. DOI: 10.48550/ARXIV.2003.03052. URL: https://arxiv.org/abs/2003.03052.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations*. Tech. rep. 148. Department of Computer Sciences, The University of Auckland, 1997. URL: http://www.cs.auckland.ac.nz/$%5Csim$collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. "Breaking abstractions and unstructuring data structures". In: *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*. IEEE. 1998, pp. 28–38.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing cheap, resilient, and stealthy opaque constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998, pp. 184–196.

[15] Patrick Collins. *DEFI Minimal*. 2023. URL: https://github.com/smartcontractkit/defi-minimal/.

[16] Filippo Contro et al. "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode". In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE. 2021, pp. 127–137.

[17] Christian Decker and Roger Wattenhofer. "Information propagation in the Bitcoin network". In: *IEEE P2P 2013 Proceedings*. 2013, pp. 1–10. DOI: 10.1109/P2P.2013.6688704.

[18] Dedaub. "The Dedaub decompiler takes Ethereum Virtual Machine (EVM) bytecode and produces more readable Solidity-like code, allowing for better understanding of unverified smart contracts." In: [Accessed 26-09-2023]. 2023.

[19] William Entriken et al. *EIP-721: Non-Fungible Token Standard*. 2018. URL: https://eips.ethereum.org/EIPS/eip-721.

[20] Nomic Foundation. *Hardhat : Ethereum development environment for professionals*. 2023. URL: https://hardhat.org/.

[21] G.K. Gill and C.F. Kemerer. "Cyclomatic complexity density and software maintenance productivity". In: *IEEE Transactions on Software Engineering* 17.12 (1991), pp. 1284–1288. DOI: 10.1109/32.106988.

[22] Confio GmbH. *What is CosmWasm?* 2023. URL: https://docs.cosmwasm.com/fr/docs/.

[23] Jay Graylin et al. “Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship”. In: *Journal of Software Engineering and Applications* 2.03 (2009), p. 137.

[24] Stuart Haber and W Scott Stornetta. “How to time-stamp a digital document”. In: *Conference on the Theory and Application of Cryptography*. Springer. 1990, pp. 437–455.

[25] Pieter Van Remoortere Halstead. “Elements of software science : M.H. Halstead: Published by North Holland Amsterdam and N.Y., 1977, 128 pages, US $ 18.95. ISBN 0-444-00205-7”. In: 1979. URL: https://api.semanticscholar.org/CorpusID:118534371.

[26] Shohreh Hosseinzadeh et al. “Diversification and obfuscation techniques for software security: A systematic literature review”. In: *Information and Software Technology* 104 (2018), pp. 72–93. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.07.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584918301484.

[27] Teng Huang et al. “Smart contract watermarking based on code obfuscation”. In: *Information Sciences* 628 (2023), pp. 439–448. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2023.01.126. URL: https://www.sciencedirect.com/science/article/pii/S002002552300138X.

[28] Markus Jakobsson et al. “Proofs of Work and Bread Pudding Protocols”. In: *Communications and Multimedia Security* (1999). DOI: null.

[29] Sunny King and Scott Nadal. “PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake”. In: *null* (2012). DOI: null.

[30] David Mazières and Dennis Shasha. “Building secure file systems out of Byzantine storage”. English (US). In: *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. Proceedings of the Twenty - First Annual ACM Symposium on Principles of Distributed Computing PODC 2002 ; Conference date: 21-07-2002 Through 24-07-2002. 2002, pp. 108–117.

[31] T.J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[32] Richard Moore. *ethers : a simple, compact and complete JavaScript library for all your Ethereum needs*. 2023. URL: https://ethers.org/.

[33] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: *Decentralized Business Review* (May 2009). URL: http://www.bitcoin.org/bitcoin.pdf.

[34] Nick Szabo. “Formalizing and Securing Relationships on Public Networks”. In: *First Monday* 2.9 (Sept. 1997). DOI: 10.5210/fm.v2i9.548. URL: https://firstmonday.org/ojs/index.php/fm/article/view/548.

[35] openchain.xyz. *Openchain Signature Database*. 2023. URL: https://openchain.xyz/signatures.

[36] Paradigm. *Foundry is a blazing fast, portable and modular toolkit for Ethereum application development written in Rust*. 2023. URL: https://getfoundry.sh/.

[37] Parity. *Why WebAssembly for Smart Contracts ?* 2023. URL: https://use.ink/why-webassembly-for-smart-contracts.

[38] Anniina Saari, Jussi Vimpari, and Seppo Junnila. “Blockchain in real estate: Recent developments and empirical applications”. In: *Land Use Policy* 121 (2022), p. 106334. ISSN: 0264-8377. DOI: https://doi.org/10.1016/j.landusepol.2022.106334. URL: https://www.sciencedirect.com/science/article/pii/S0264837722003611.

[39] Pamela Samuelson. “Reverse-engineering someone else’s software: is it legal?” In: *IEEE Software* 7.1 (1990), p. 90.

[40] Ju Myung Song, Jongwook Sung, and Taeho Park. “Applications of Blockchain to Improve Supply Chain Traceability”. In: *Procedia Computer Science* 162 (2019). 7th International Conference on Information Technology and Quantitative Management (ITQM 2019): Information technology and quantitative management based on Artificial Intelligence, pp. 119–122. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.11.266. URL: https://www.sciencedirect.com/science/article/pii/S1877050919319787.

[41] Fabian Vogelsteller and Vitalik Buterin. *EIP-20: Token Standard*. 2015. URL: https://eips.ethereum.org/EIPS/eip-20.

[42] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[43] Qifan Yu et al. “Bytecode Obfuscation for Smart Contracts”. In: *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*. 2022, pp. 566–567. DOI: 10.1109/APSEC57359.2022.00083.

[44] Pengcheng Zhang et al. “BiAn: Smart Contract Source Code Obfuscation”. In: *IEEE Transactions on Software Engineering* 49.9 (2023), pp. 4456–4476. DOI: 10.1109/TSE.2023.3298609.