Enhancing Smart Contract Security Through Obfuscation: Verification via Control Flow Graph Analysis

Karishma Yadav, Smita Naval

Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur, India

Corresponding author: Smita Naval, Email: smita.cse@mnit.ac.in

Ethereum smart contracts leverage blockchain technology to facilitate the transfer of values directly between participants on a network, eliminating the need for a central authority. These contracts are deployed on decentralized applications that operate on top of the blockchain. By doing so, they provide individuals with the ability to create agreements in a transparent and secure environment, minimizing conflicts and promoting trust. It has been observed that there are bugs in the smart contract's codes as these are provided by various programmers across the globe. The attackers exploit these security loopholes and pose a significant threat to applications, which subsequently result in financial losses to users. Discovering vulnerability in each contract is an important but time-consuming task. Therefore, we require to provide a security layer to each smart-contract such that it will make the exploitation a bit difficult task for attackers. The use of encryption and obfuscation techniques improves the security layer. The main focus of this research is source code obfuscation, which can increase security by up to 75%. The code obfuscation in security is mainly used by attackers to hide their malicious intent. We, in this approach suggest this method for increasing the complexity of smart contracts so that these cannot be exploited easily. We evaluate the impact of adding security layer to smart contract. The evaluation was done with various static and dynamic tools that identify the vulnerability in smart contracts. We achieved promising results which show that Obfuscation technique enhances the security and complexity of codes up to 75% which are stored on public blockchain.

Keywords: Obfuscation, Blockchain, Smart contract, vulnerability, Control Flow Graph.

2025. In Sandeep Kumar & Kavita Sharma (eds.), *Computational Intelligence and Machine Learning*, 57–84. Computing & Intelligent Systems, SCRS, India.
DOI: https://doi.org/10.56155/978-81-975670-5-6-6

1 Introduction

Blockchain is the technology, covered by Satoshi Nakamoto in 2008 [28]. Blockchain technology is a distributed database which supports Smart contracts [4, 14, 31] which is replacement of third parties in blockchain technology. Smart Contract contains all necessary condition of the involved parties for specific tasks are mentioned and all the necessary conditions are design by the programmer using specific programming languages like solidity [2], Vyper, Rust [6] etc. Smart Contract conditions are designed with the help of lawyers which will justify all legal terms and make easy for programmer so that these terms are designed and fitted in contract without any misunderstanding [26, 35, 37, 42]. Blockchain will check each condition and the current state of the smart contract before run the contract. Smart contracts are adopted due to its immutable property which means, it can't be updated by any involved parties and minimizes the third-party load which reduces the risks of information leaking from the third-party side. Smart contracts are irreversible if there are any changes in smart contract it will be again designed and deployed on Blockchain. Smart contract security is of utmost importance since they have the ability to store billions of dollars; a single error might result in the loss of the entire amount.

There are few crucial rules that must be followed when designing smart contracts. For instance, there must be no logical or syntactical errors of any kind in smart contracts. Designing of smart contracts includes following coding languages like Solidity, Go, Rust, Haskell etc [29]. But which language is better and safe for smart contracts is another ongoing research challenge. The Ethereum platform is most popular where solidity-based smart contracts are employed.

Bytecode [3,13] is the intermediate code which is understood by the Ethereum Virtual Machine(EVM). Bytecodes are long value hexadecimal representations of the final contract. Based on one byte, the bytecodes can be translated into EVM instructions or operation codes (opcodes). An attacker might easily misunderstand how a smart contract works, construct a malicious smart contract, and call the victim's smart contract to take advantage of its capabilities and utilize them maliciously because plain text source code is available.

Therefore, obfuscation is employed to preserve the security of source code; nevertheless, obfuscation increases code complexity, which affects code detection accuracy utilizing tools [23, 40]. And complexity of the code degrade the accuracy of the code. In this paper, we will focus on obfuscation techniques and apply them on source code and extract opcodes. After that we analysis the security and complexity of the code using existing tools.

The remainder of this article is structured as follows. We discussed about taxonomy comparison of vulnerability detection tools taxonomy comparison of vulnerability detection tools in Section 3 and We introduce the proposed works in Section 4. Section 5 contains a map of the dataset's details and Section 6 has the evaluation of work is discussed. The paper's results are assessed in section 7. Finally, in Section 8, we draw a conclusion and talk about potential future research.

Our goal is to enhances the security of smart contracts, and we suggest an obfuscation strategy for them. Our scheme is characterized by the following points

1. Analysis the obfuscation techniques for solidity code.
2. Apply obfuscation techniques on solidity codes and analysis the performance of the codes using available tools.
3. Generate CFG (Control Flow Graph) of obfuscated and non-obfuscated codes and analysis complexity factor in CFG edges and blocks.

2 Literature Review

2.1 Smart Contracts History

In past bitcoin [17, 28] is only used as medium of exchange between peer to peer nodes. Yet there are a lot more options offered like blockchain technology. As we know that blockchain supports smart contract's and smart contract's enhances security, anonymity and data integrity capabilities all operate independently of outside third parties. These capabilities will fundamentally alter many applications. Blockchain technology is used in numerous applications, including IoT, Smart Grid, Supply Chain Management, Healthcare, Smart Property, and Digital Content Distribution among others. Bitcoin also develops smart contracts, but they are inferior to Ethereum's. Scripts, the name given to Bitcoin's smart contracts, have a wide range of limitations and are not ideal for many use cases [39].

Now we will discuss about Ethereum which is used as a platform for smart contracts.

2.2 Ethereum Smart Contracts

Ethereum [22] is the open platform for smart contracts and smart contracts are immutable so if there is any change in smart contract it can't be edited. Along with this smart contracts are free from third parties there is no requirement of any mediator for smart contract. Since smart contracts can be easily combined

and reused, their openness makes them remain very appealing for many use cases. Decentralized finance (DeFi) [25], where contract-based systems are confronting financial institutions like banks and exchanges is the most notable use of this creation of interconnected smart contract systems. EVM (Etherum Virtual Machine) [20] must offer an abstraction between running code and running machine because the code and transactions are executed on miner nodes. To do this, EVM employs a series of opcode instructions. When a smart contract is launched, it's code must be converted into the EVM bytecode that represents these instructions like MSTORE is 0x52, SSTORE is 0x55, PUSH1 is 0x60. Solidity and Vyper are the two primary programming languages employed at the time of writing for Ethereum smart contracts. Solidity [7], an object-oriented, statically typed, curly-bracket language that allows inheritance, libraries, and user-defined types, is the more durable of the two. In contrast, Vyper [32] is a tightly typed, python-like language with less features than Solidity. Its objective is to use simplicity to make contracts more secure and auditable. But during programming smart contracts due to some logical error, syntactical errors and functions limitations vulnerabilities may occur in smart contracts.

2.3 Vulnerabilities in Smart Contracts

Vulnerabilities are loopholes which attract attackers and which cause million of dollar loss [22]. Due to denial of service with failed calls vulnerability in 2016, 110 ether loss, re-entrancy vulnerability causes the reason of loss of 60 million us dollars, and in February 2022, 320 dollar drained. We found that vulnerabilities are the big loopholes in smart contracts and due to immutability features of smart contract we can't modified them. If we must fix the bug, then a new smart contract should be designed. Additionally, as the deployer details of smart contracts are anonymous, we are unable to notify the smart contract owner if a bug is discovered. Smart contracts are deployed on ethereum in the form of source code and bytecode attacker can easily track the functions and working of smart contracts. If they found any function weakness, then attacker can easily drain the ethers from smart contract without the knowledge of smart contract owner. In order to solve these issues and rescue smart contracts from these losses, a number of automated tools that analyze vulnerabilities. Some of the tools are based on source code, while others are based on bytecode and opcodes, and few tools support both. Around 150+ smart contract vulnerabilities detection tools are available. But there are some limitations still faced by smart contract tools like they didn't support all version of smart contracts and tools are based on specific vulnerabilities no tools detect all vulnerabilities. So we are not sure about the

security of smart contract source code. After go through all the papers we found that obfuscation add-On a security layer. Obfuscation of code malicious actors didn't see the functions and logic of the code. Furthermore, it is very difficult for malevolent attackers to create a different contract that disrupts the original contract utilizing contract calls without being able to view the code and functions. For obfuscation we have used the BiAn tool.

BiAn [1] is a source-code level obfuscation tool built for Solidity smart contracts and it is responsible for generating the masked samples. That is the only method we are aware of for hiding the nature of an Ethereum smart contract at the time of this research. There are three types of contract obfuscation [15] that BiAn is capable of producing: Layout Obfuscation (LAO), Data Flow Obfuscation (DFO), and Control Flow Obfuscation (CFO for short). LAO simply modifies the lexical analysis, such as changing variable names and removing comments and not the logical structure of the source code. Such as DFO and CFO modify the flow of data and control respectively. Maintainers of BiAn have admitted that the CFO module is broken at the time of our research. If the source code for the obfuscated contract can be compiled into a valid contract, then we may safely believe that the obfuscation was successful. As a result, we only find LAO, DFO, and LDO (coupled with LAO and DFO) to produce evasive contracts. We have used this tool for obfuscation and generated some more obfuscation smart contracts with other techniques like dead code insertion, opaque code insertion, array transformation etc.

2.4 Obfuscation

A key method for shielding software intellectual property is program obfuscation [5,38]. It modifies computer programs into new iterations that are less intuitive but semantically equal to the original. The idea was first presented during the 1984 International Obfuscated C Code Contest, which gave prizes for innovative C source codes with "smelly aesthetics". It has now evolved into a technology that is essential for software security. Since obfuscation has been researched for about 30 years, there are numerous surveys accessible.

Obfuscation however,is undoubtedly a viable option for enhancing privacy and add a security layer. Our findings provide verifiable evidence that obfuscation has practical uses for the entire public blockchain ecosystem, and we advocate for increased security and safety while creating blockchain applications. Obfuscation variations are like secure and usable obfuscation, code-oriented obfuscation explained in next section.

2.4.1 Secure and Usable Obfuscation

An obfuscation technique must be highly resistant and efficient in order to guarantee safety. Obfuscation needs to safeguard the semantic privacy of the program in order to be effective. To be more precise, a secure obfuscation technique should mask the fundamental meaning of a program.

Resistance refers to how challenging it is to decipher the concealed meaning. Attacks shouldn't be able to break a secure obfuscation method. When discussing code-based obfuscation, obfuscation capability is associated with power, whereas resistance encompasses both robustness and stealth.

2.4.2 Code-Oriented Obfuscation

A work on code obfuscation was first published in 1993 by [10]. A seminal study on the classification of obfuscation changes was published by [12] in the latter half of 1997. Afterwards, a plethora of obfuscation strategies were put out in the published works. we can classify code-oriented obfuscation techniques as preventive obfuscation, transformation obfuscation, or polymorphic obfuscation according to their intended level of security. The purpose of preventive obfuscation is to make it more difficult for attackers to access the original source code. It is more difficult to understand the original codes once they have been transformed, which reduces their security. As such, the goal of polymorphic obfuscation is to make it difficult, if not impossible, for attackers to identify the semantics or features they are looking for in each disguised version. We can further categorise them by how they obscure information, such as layout transformation, control transformation, or data transformation. These transformations increase the complexity of the code as we have shown in definition 1.

Definition 1 When a program Q is c times more complex than a program P—that is, when the added complexity is c times the original one and is hence harder to understand—it is said to be c-unintelligible with respect to P. Mathematically:

$$K(Q) >= (c + 1)K(P) \tag{6.1}$$

and,

Definition 2 A C-Obfuscator

$$O \equiv (P * L \rightarrowtail)Q \tag{6.2}$$

is a mapping from programs with security parameters L to their obfuscated ver-

sions such that for all

$$P \epsilon P, \lambda \epsilon L. O\,(P, \lambda) \neq P \tag{6.3}$$

where (P,λ is the input to O, and satisfies the following properties:

- **Secrecy** : Security parameters are kept private and contain all the information necessary to acquire P given O (P,λ).
- **Functionality**: The computation of the same function using O(P,λ) and P results in [[P]] = [[O(P,λ)]].
- **Polynomial Slowdown**: The size and running time of O(P,λ) are at most polynomially larger than the size and running time of P, i.e. for some polynomial function p.

  $|(P, \lambda)| \leq p\,|(P)|$,
  and if P halts in k steps on an input i, then O(P, λ) halts within p(k) steps on i.
- **Unintelligibility**: O(P, λ) is c-unintelligible with respect to P.

2.5 Types of Obfuscations

1. Preventive Obfuscation

   Preventative obfuscation makes it harder for attackers to access code in a readable manner. It was mainly designed to be used with non-scripting languages, such as Java and C/C++. In order to impede the disassembly phase, preventive obfuscation introduces mistakes to general dissemblers. The goal is to inject incomplete instructions as trash codes after unconditional jumps, which linear sweep algorithms will see as a red flag. If a dis-assembler is unable to process such incomplete instructions, the mechanism will be activated. Further, they swap out standard procedure calls for branch functions and jump tables to foil recursive algorithms. Since the return addresses are determined dynamically at runtime, static disassemblers have less chance of discovering them. The same idea was offered by [30], who want to change unconditional jumps into traps that trigger signals.

2. Layout Obfuscation

   The layout of a program is jumbled up while the syntax remains unchanged through layout obfuscation. It might, for instance, swap around the names

of variables and classes or rearrange the execution sequence of instructions. Commonly used in layout obfuscation, lexical obfuscation replaces meaningful identifiers with gibberish. Adopting consistent and clear naming standards (like the Hungarian Notation [33] is a best practise for most programming languages. Due to the irreversibility of some alterations, layout obfuscation offers excellent resistance. Unfortunately, the obfuscation is only effective at the layout level. Methods used in layout obfuscations are:

- Remove Comments:

  Obfuscation at this level is the most basic type of documentation for software and frequently includes developer comments. As far as I can tell, comments and debugging information are routinely removed from a program when it is compiled, despite the fact that doing so apparently reduces the program's readability.

- Source-code Formatting:

  Removing all white-space and indentations from the program source code [16] changes the format of the code, making it less readable by a reverse engineer. Obfuscation C competition [16] contains examples of such formatting obfuscation.

3. Control Obfuscation

   When control flows are obfuscated, they become more difficult to follow. In computer programming, a bogus control flow is one that was added on purpose but will never be used. It will make a program more difficult to understand and maintain, as measured by metrics like McCabe complexity [24]and Harrison metrics [19]

   The McCabe complexity of a linked component can be raised by adding either new edges or new nodes or by doing both [12] proposed the concept of opaque predicates to ensure the unreachability of fake control flows shown in figure 1. They came up with the term "opaque predict" to describe a predicate whose result is known at obfuscation time but is obscured by static code analysis.

   Example of control flow is shown in Figure 2.

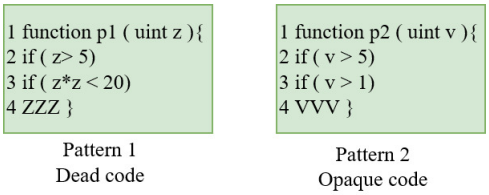


Figure 1: Dead code and Opaque code used for obfuscation

a) Aggregation:

Aggregation transformations include inlining (which replaces a method call with the body of the method) outlining (which replaces a sequence of statements with a method call), interleaving (which combines multiple methods into one), cloning (which generates multiple copies of the same method), and loop transformation (which rewrites a loop) [12] and [36].

b) Ordering:

The order in which the calculations are performed can be shuffled around. Swap the placement of a term in an expression, a statement inside a basic block, a method inside a class, or a class inside a file. For example, loop reversal [36] and control flow flattening [12] allow for the reordering of loops. By flattening the relevant control flow graph. Control flow flattening eliminates the control flow structure that functions have included the nested loops and conditional statements. **Figure 2 is an example of code obfuscation using control flow flattening [27].**

c) Computation:

It alters the source code algorithm by introducing dead code (as seen in figure 1) or redundant operands. One of the most common forms of computational obfuscation is the practice of parallelizing programs, which conceals the underlying control flow by generating inactive processes. It also separates a block of code that runs sequentially into multiple pieces that can be executed concurrently.

4. Data Transformation:

Data transformation makes the program functionality more difficult to understand. Data transformation is broken down into three categories by [12]:

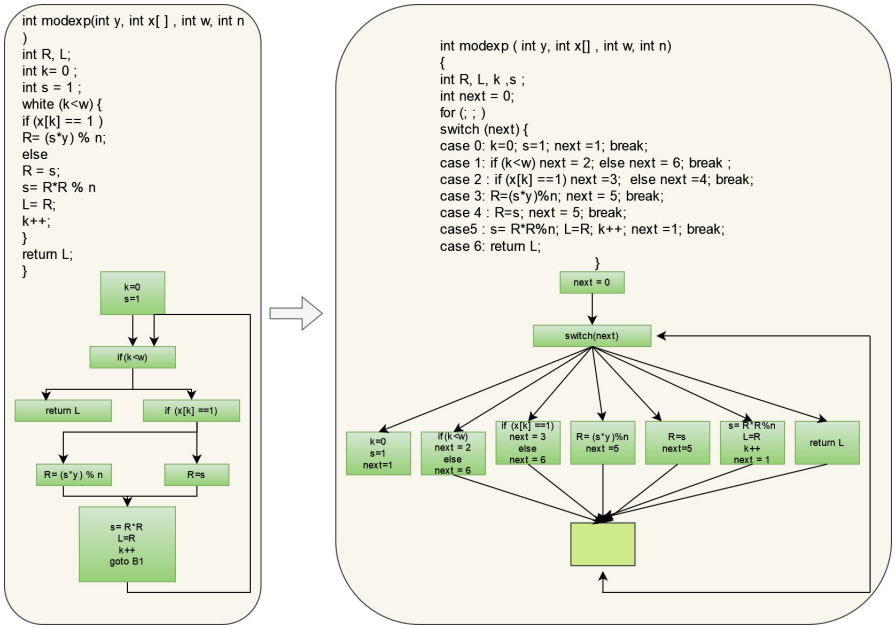


Figure 2: Control flow obfuscation

storage and encoding, data aggregation, and data ordering. Drape [16] proposed the abstract data-type for concealment and gave a taxonomy based on the work of [12]. Methods for Data transformation

a) Variable Encoding:

   To encode a variable, an expression is substituted for it; for example, if I is a variable, then it would be replaced by the expression I = di + e, where d and e are constants. Encoding variables so that their correct values can be retrieved on demand is called "invertibility" and it becomes especially important when one variable's output is needed to determine the value of another variable.

b) Merging and Splitting:

   Two or more scalar variables can be combined into a single variable for obfuscation purposes, provided that the combined range is contained within the accuracy of the resulting obfuscated variable.

c) Array Transformations:

   Reorganizing operations, such as modifying the array's indexes, can be used to conceal information contained within an array.

2.5.1 Overview of Findings

After literature survey we will highlights the findings of this area in this section. Till date we do not have secure and usable obfuscation techniques which increases the security with less overhead. First, no previous studies of obfuscation have focused on assessing the impact on the program's residual semantics. Evaluation metrics proposed by [12] are widely used; these metrics evaluate the increasing obscurity of the code rather than the semantics of the unprotected code. When it comes to obfuscation efficiency and other security-related parameters, none of them can measure up. Therefore, it appears that defining suitable measurements is the first order of business for future research into secure obfuscation techniques. It is possible that the stringent security requirements for obfuscating circuits make it impossible to create even a theoretically viable obfuscation solution. For this reason, graded encoding may be the sole kind of obfuscation that may now fulfil a security need.

Most of the obfuscation techniques applicable for simple mathematics expressions. It is difficult to handle non-mathematical syntax and other coding concepts like data flow and control flow using obfuscation.

3 Vulnerabilities Detection Tools

Vulnerabilities are the loopholes which occur due to weakness of code and coders, and they can resolve but some time vulnerabilities resolved after paying a large amount of loss. In Ethereum which is a public blockchain and there are millions of smart contracts deployed but due to some error attacker take the advantages of that error and extract the balance of the smart contracts like in re-entrancy attack this attack occur due to lack of ordering of instructions. Re-entrancy attack occur due to irregular patterns of logic's.

So along with re-entrancy attack there are many more vulnerabilities possible in solidity. Solidity is the language used for writing smart contracts for Ethereum blockchain and polygon blockchain and both blockchain are base of billion of ethers.Analysts have realized that prior to smart contracts being implemented, vulnerability analysis is crucial. For this examination, there are three primary approaches: formal, dynamic, and static. Every approach has advantages and disadvantages of its own [11]. For example, while dynamic analysis tools evaluate contracts as they are being executed, they may result in slower testing than static analysis tools, which can be effective but may also yield false positives and negatives. Tools that make use of these methods are being created in order to find

vulnerabilities before they are implemented on the blockchain. There is a list of available tools for analysis of smart contract. But we found that smart contract analysis tools are focusing on maximum 5-6 vulnerabilities at a time and every tool is depends on version of solidity so version dependency of tools is the weak point of tools. Inferences of tools:-

- Tools are failed to detect obfuscated patterns.
- Tools are version dependent.
- Tools fail to analysis the enhanced complexity of code.

Table 1: Literature survey of Solidity obfuscation tools

Year	Objective	Approach	Dataset	Findings
2023 [41]	BiAn: smart contract source code obfuscation	Code Obfuscation using BiAn tool	1000 buggy smart contracts	Experiments with buggy smart contracts.
2022 [21]	Smart Contract Obfuscation Technique to Enhance Code Security and Prevent Code Reusability	By substituting characters with random counterparts, or statistical substitution, procedures function.	NA	Five techniques for obscuring a smart contract.
2022 [34]	Smart Contracts Obfuscation from Blockchain-based One-time Program	Cryptographic obfuscation scheme for smart contracts	NA	No secure setup is designed yet.
2022 [9]	WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts	Concolic fuzzer	3,340 obfuscated samples	Data flow and control flow obfuscation using popcount algorithm
2022 [23]	SmartFast: an accurate and robust formal analysis tool for Ethereum smart contracts	Taint tracking technology	13,687 contracts	Feature extraction for 3 types of vulnerabilities
2021 [8]	SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts	Symbolic execution technique	1,395 contracts and 133 ponzi contracts)	Opcodes feature extraction work for Layout obfuscation and data flow obfuscation
2020 [41]	Source Code Obfuscation for Smart Contracts	Code Obfuscation using BiAn tool	10 vulnerabilities and 300 obfuscated smart contracts	When a contract's complexity rises, it is evident that most smart contract static analysis tools perform worse when the original contracts are obscured.

SADPonzi mechanism is tool which detect four type of obfuscation techniques DFO (Data flow obfuscation) CFO (control flow obfuscation) LAO (layout Obfus-

cation) and combination of LAO and DFO and Smart Contracts. Obfuscation of Blockchain-Based One-time Program piece also explores obfuscation. A blockchain uses a corrupted circuit and witness encryption. All information, including the algorithm and any hard-coded secrets, are kept private by the proposed system. Its security relies on proven blockchains and doesn't necessitate a trusted hardware majority or even a majority of truthful participants in order to function.

This tool only detect few obfuscation transformations only. But if we manipulate data using more than one technique than this method fail to detect obfuscation.

Blockchain-enabled Cryptographically-secure Hardware Obfuscation [18]. This study offers a revolutionary Blockchain-enabled, cryptographically-secure hardware obfuscation strategy that is backwards-compatible with existing circuit synthesis and fabrication methods. Here, Proof-of-Stack Blockchain protocols and witness encryption algorithms ensure the safety of the obfuscation rather than just monitoring the supply chain via the Blockchain. WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts [9]. According to this paper we found that there is no tool for bytecode obfuscation tool available so they can obfuscated the code using two techniques. The first thing is to hide its data trail to encode function arguments using the pop-count algorithm [9], which tally's the number of bits set to '1' in the provided value. Second, it injects recursion invocations into the bytecode at locations where the entry condition can never be satisfied, which obscures the control flow.

4 Proposed Work

In this section we discussed about the proposed work in figure 6 which is based on CFG of the source codes. The core of most static software analysis is the representation of a program structure in the form of a control flow graph (CFG) or call graph. we recover CFG and call graph information from both original and obfuscated source codes. We determine the total number of basic blocks, call graph edges, and control graph edges by traversing the respective graphs. These details are what we use to evaluate a (obfuscated) program complexity. The data analysis results are displayed in Table II. Obfuscated programs are more complicated than their unmodified counterparts across the board. We further analysis the Cyclomatic metric is defined as

Cyclomatic = E - N + 2 where E and N stand for the total number of edges and nodes in a directed graph. The number of knots in a CFG represents the total number of edge crossings in that graph. To some extent, the number of log-

ical branches in a program can be inferred from these two measurements. Table II demonstrates that following Turing machine obfuscation, both the knot and cyclomatic number grow significantly for both scenarios. We conclude that the promising results displayed in Tables II and figure 4 and 5 graphs demonstrate that obfuscation adds security layer which enhances complexity to the original algorithms. Algorithm also shown in section VI (Proposed Work). After executing algorithm, we can see the results in table II, illustrates the number of call graph edges with the growth of obfuscation levels. According to empirical evidence, the number of call graph edges grows in tandem with the obfuscation level. Naturally, as obfuscation levels rise, obfuscated programs get more complex. In the next section, the algorithm used in the proposed work is explained.

4.1 Algorithm

Procedure EXTRACT SMART CONTRACTS (Etherscan.io)

Step1: Analysis of Smart Contracts ← Available tools

Step2: Check vulnerabilities ← and label the dataset

Step3: Extract bytecodes of vulnerable codes ← Solidity compiler

Step4: Extract opcodes of vulnerable codes ← Solidity compiler

Step5: Generate CFG of Opcodes

Step6: Analysis of CFG ← Ether solve

Step7: Obfuscate Smart Contract

Step8: If Obfuscate (CFG) = Non-Obfuscate (CFG)

Then

Step9: Repeat step 3 to step 7

Step10: Analysis CFG using tools

Step11: if Obfuscate bytecode = Non_obfuscate_bytecode

Then

Step12: Same complexity

Else

Step13: if obfuscation_bytecode != Non_obfuscationcode

Then

Complexity enhanced

End if

We deduce the CFG from the bytecode of Ethereum smart contracts. Bytecode Parsing use raw bytecode where the analysis begins. Finding the appropriate header mentioned in the official literature will help us to locate the metadata section [2]. When metadata has undocumented experimental features, the header is different. By manually looking at the bytecode of several contracts, we deduced the header structure of experimental instances that were not explicitly disclosed. The metadata is needed to determine which version of the Solidity compiler was used to create the bytecode. The Control Flow Graph (CFG) is a directed graph reflecting the flow of execution, nodes are the basic blocks of program (patterns of opcodes with no jumps, except in the last opcode). A basic block is a set of instructions that are executed sequentially between a jump target and a jump instruction without the presence of any other instructions that change the direction of control. JUMPDEST denotes the start of a new basic block, whereas JUMP, JUMPI, STOP, REVERT, RETURN, INVALID, and SELFDESTRUCT opcodes denote the end of an existing basic block. Every basic block has a unique identifier known as an offset, which indicates where its beginning opcode is situated in the bytecode. Although the jump destination is not an opcode parameter and is instead provided at execution time, this process is not always straightforward. We classified jumps into two categories: pushed jumps and orphan jumps. A pushed jump is a JUMP that has a PUSH opcode right before it so that its final point quickly determined by looking at the value of the PUSH opcode. Edges are the possible connections between succeeding basic blocks.

The edges of pushed loops are computed by the ethersolve tool [1]. To achieve this, each basic block is examined in accordance with its most recent opcode:

- When a JUMP is followed by a PUSH, the appropriate edge is added to the CFG and the push argument is the jump destination offset.
- When a PUSH comes before a JUMPI, the true branch is the push argument, which is regarded as the JUMPI destination offset. The false branch leads to the block after it (in offset order). In this instance, the CFG is expanded to include the two related edges.
- JUMP not immediately preceded by a PUSH: The jump must be resolved by symbolic stack execution because it is not an easy problem to handle.
- There are no successors for REVERT, SELFDESTRUCT, RETURN, INVALID, and STOP since the control flow is broken.

[1] https://github.com/SeUniVr/EtherSolve

- If not, the basic block is located at the following point in the bytecode and the control flow continues.

The approach uses a DFS (Depth-First Search) to iterate through the CFG while maintaining a stack state for each fundamental block.

- Entry blocks : Entry block is the start point of CFG.
- CFG edges: Edges used to connect the basic blocks of the CFG.
- Basic blocks : A basic block is a block of code that is run in order between a jump target and a jump instruction, without any extra instructions that change the flow of control. The code is broken down into smaller modules by opcodes that modify the program's control flow. A basic block's execution is terminated by the opcodes JUMP, JUMPI, STOP, REVERT, RETURN, INVALID, and SELFDESTRUCT, while a new block's execution is initiated by the opcode JUMPDEST. An offset, or the location of an instruction's first opcode in the bytecode, is a distinctive marker for each basic block. There is a JUMPDEST at the beginning of each basic block and an opcode that changes the flow of control at the end of each one. After the code has been broken down into its constituent parts, we may move on to computing the edges.
- Dispatch blocks : The fallback function's entry point can be found via a dispatcher block. A smart contract's execution can be kicked off with a transaction that transfers more than just money. To ensure that the correct function is executed, the compiler inserts a dispatcher into the contract code from the very beginning.
- Invalid blocks : Invalid blocks are denoted by red boarder and blocks are invalid if they end with REVERT, STOP, RETURN.

These are the type of blocks which are the part of CFG and opcodes patterns are used for identification of vulnerabilities. But due to obfuscation many unused and bug blocks are added in CFG which make CFG analysis more complicated. In table II we have analysis the CFG changes after and before obfuscation. And, in table III analysis results using available tools.

4.2 Tables

In this section we discussed about the Comparison of CFG Graph complexity before and after obfuscation of code. And we have calculated the graph complexity in terms of complexity of graph in table II.

Table 2: Comparison of CFG Graphs Before and After Obfuscation

Vulnerabilities	# CFG Edges		# Basic Blocks		# Dispatch Blocks		# Entry Blocks		Complexity	
	Without_obfu	With_obfu	Without_obfu	With_obfu	Without_obfu	With_obfu	Without_obfu	With_obfu	Without_obfu	With_obfu
Reentrancy	56	74	50	63	15	18	1	3	8%	13%
Access_Control	25	61	18	48	6	9	1	1	9%	15%
Bad_Randomness	36	86	23	62	9	14	1	2	15%	26%
Denial of Service	43	89	15	54	10	16	1	1	30%	37%
Arithmetic	34	96	18	79	5	9	1	1	18%	25%
Front_running	45	107	20	94	8	14	1	3	15%	27%
Short_addresse	23	74	15	56	6	9	1	1	10%	20%
Unchecked_low_call	34	56	19	23	7	10	1	3	17%	35%
time_manipulation	25	77	20	65	8	15	1	1	7%	14%
Other	23	48	17	28	5	8	1	2	8%	22%

5 Dataset

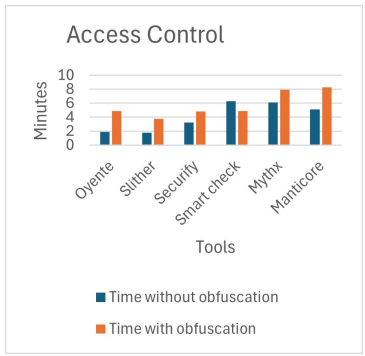
SWC registry [2] and Messi-Q smart [3] contract dataset were mined for smart contracts, then we obfuscated each vulnerability by hand using a variety of methods like dead code insertion, control flow obfuscation, renaming of variables and insert opaque code and then create a dataset and dataset contains 3320 smart contracts. Obfuscated smart contracts data-set is analysed and examined the complexity of the graph. We examine how the intricacy of smart contracts has grown together with their gas use and the failure rate of existing methods in identifying vulnerabilities in smart contracts has dropped.

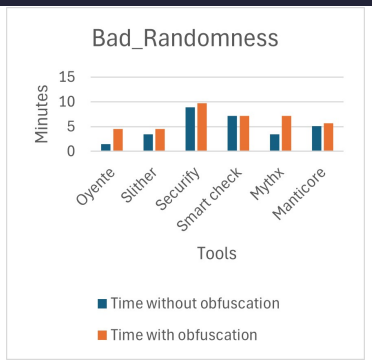
6 Evaluation

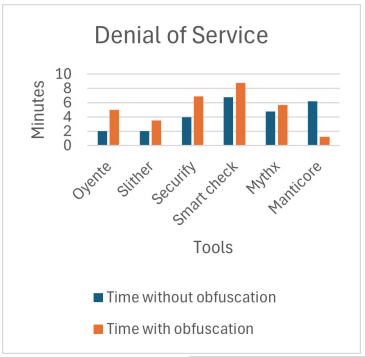
Randomly selecting 500 contracts containing 10 types of bugs. We use BiAn tool to generate obfuscated contracts and compile them with solc to produce bytecode and manually label the bug locations of the obfuscated contracts and create a public buggy contract dataset. BiAn tool make the contracts more complex, then we employ Ethersolve, a smart contract CFG (Control flow Diagram) generator which convert the source to bytecode and generate blocks for opcodes corresponding to byte-code. Calculate the number of Edges and blocks inside the CFG and compare the blocks and edges between the original and obfuscated contracts. The result, as shown in figure 7 and figure 8 demonstrates that BiAn can multiply the complexity of the contracts along with the gas consumption. Gas consumption in obfuscated code might increase for a variety of reasons. First is complexity of Obfuscation techniques sometimes require adding new layers of complexity to the code. Second, expanded bytecode size due to obfuscation. Third, Obfuscation may result in duplicate or superfluous processes in the code. Fourth, deploying obfuscated smart contracts can result in higher initial gas prices. Last, but not least Obfuscated code may take longer to execute due to its increased complexity or poor design. When we evaluate different Solidity static analysis tools, we use obfuscated contracts and observe their performance and complex contracts were analyzed the losses and efficiency. We have selected 5 state-of-the-art tools. These tools are free to use; 2) These tools use Solidity contract and compiled bytecode as input. These 5 tools used to detect vulnerabilities in original and obfuscated contracts. However, tools detect only 6 types of bugs out of 10. Among the five tools. Oyente and Orisis are unable to evaluate obfuscated contracts, even

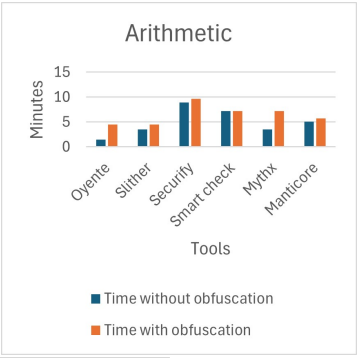
[2] https://swcregistry.io/
[3] https://github.com/Messi-Q/Smart-Contract-Dataset

```
18
19      // withdraw all funds from the user's account
20      function withdraw() external nonReentrant {
21          require(balances[msg.sender] > 0, "Withdrawl amount exceeds available balance.");
22
23          // console.log("");
24          // console.log("EtherBank balance: ", address(this).balance);
25          // console.log("Attacker balance: ", balances[msg.sender]);
26          // console.log("");
27          //balances[msg.sender] = 0;
28          payable(msg.sender).sendValue(balances[msg.sender]);
29          balances[msg.sender] = 0;
30      }
```









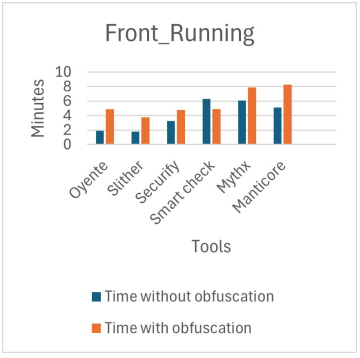


Figure 3: Graphs

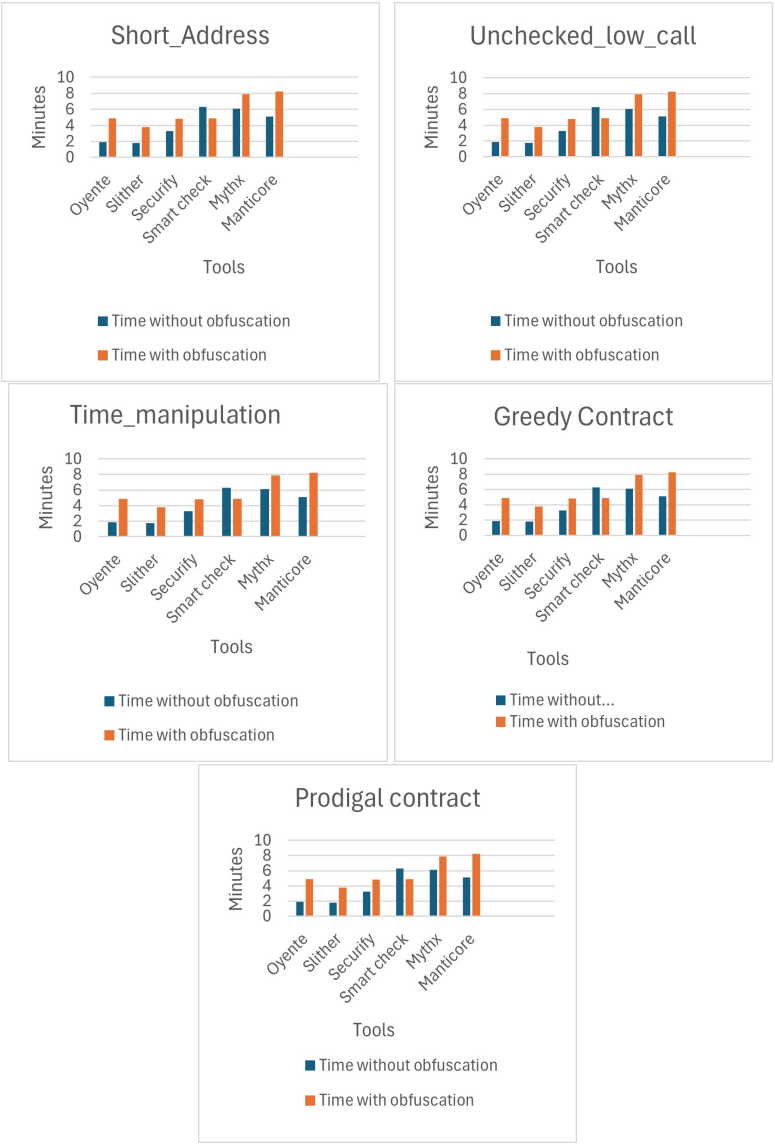


Figure 4: Graphs

if they can be successfully worked for simple smart contracts.

7 Results

In this section, we will analysis the results of obfuscated code using a variety of various obfuscation methods. Results are divided in 2 phases in first phase we collect the smart contracts and apply the CFG technique. In second, phase we obfuscate the smart contract and create a new dataset which contain only obfuscated smart contract and then apply CFG.

In first phase all smart contracts are classified according to different vulnerabilities and then analysis the smart contracts using available tools and execution time of smart contract using different tools mentioned in table III. In table III we have mentioned the time of execution of smart contracts using tools in minutes. Some of the tools mentioned in table III are failed to find few vulnerabilities and some are partially failed. After calculating time of execution of each smart contract we build the CFG of each smart contract using Ethersolve tool [13]. In table II we have mentioned the analysis of CFG details like number of edges, number of basic blocks, number of entry blocks and complexity of CFG. Along, with this in Figure 4 using graph also we have shown the complexity of the CFG's. Next, in phase second, we obfuscate the smart contract dataset using BiAn tool [41]. After obfuscation we analysis the obfuscated codes using CFG's and we found that complexity of CFG's enhances and increase in number of edges along with basic blocks and dispatch blocks as shown in Table III. Then in Table II we analysis the complexity of CFG's before obfuscation and after obfuscation and mentioned the complexity variation in Figure 4. Figure 5 shows that time variation for each vulnerabilities using different available tools with and without obfuscation.

Along with these two phases after obfuscation process when we analyses the byte code and opcodes, we find that numerous unneeded opcodes are added into the code. These opcodes didn't have any effect on the code itself, but they did make the code more complicated. Because we have demonstrated in the CFG diagram that the CFG is rapidly changing, the degree of difficulty of the graphs is listed in table II. We have examined the changes occurred before and after obfuscation in CFG's.

When we analyse these obfuscated smart contracts using static tools such as Mythril, Oyente, Slither, Securify, Silther and Manticore, we miss the accuracy of tools because of Obfuscation vs Static tool, obfuscation protect against reverse engineering,Obfuscation vs Dynamic tool, complicated the code and obfsuaction vs formal verification concealing the code logic. So we finally conclude that ob-

fuscation add a security layer for smart contracts. And increase the complexity of smart contracts which makes difficult to alter the motive of smart contracts by anonymous node.

8 Conclusion

Obfuscation has been demonstrated to be necessary and once it has been brought to light in our study, it ought to be implemented into publicly available smart contracts. Because of its adaptability, ease of use, low cost, significant improvement in security, and ability to prevent attackers from easily exploiting smart contracts, obfuscation should be used in blockchain explorers. This is because obfuscation can improve security and prevent attackers from easily taking advantage of smart contracts. In addition, code may execute more slowly as a result of performance overhead caused by obfuscation. It makes debugging and maintenance more difficult, making it more difficult for engineers to find faults and comprehend the code. Compatibility problems could also arise, and resource use might be impacted by the larger code. Lastly, dealing with obfuscated code may present a challenging learning curve for novice coders.obfuscation contributes significantly to the enhancement of the level of security. We presented a variety of various case studies on code reuse and blockchain security throughout the course of this paper.

In addition, to the best of our knowledge, there is no tool that can analyse the vulnerabilities in obfuscated smart contracts. Available tools failure rate is increased in case of obfuscated smart contracts. We have reason to expect that the results of our research will make a significant contribution to the safety of blockchains and assist in reducing the likelihood of code being reused.

In our future work, we plan to examine other obfuscation techniques and analysis utilising additional tools in order to make it more difficult for attackers to comprehend the code. And we will further try to develop a tool for vulnerabilities analysis of obfuscated smart contracts.

Table 3: Execution time of Smart contracts with and without obfuscation using tools

S.no	Vulnerabilities	Oyente		Slither		Securify		SmartCheck		Mythx		Manticore	
		Without obfuscation	Obfuscation	Without obfuscation	Obfuscation	Without obfuscation	Obfuscation	Without obfuscation	Obfuscation	Without obfuscation	Obfuscation	Without obfuscation	Obfuscation
1	Re-entrancy problem	yes (2)	partial (5)	yes (3)	No (5)	yes (5.2)	Partial (7.3)	No (3.3)	No (5.9)	yes (2.7)	No (4.9)	yes (2.3)	Partial (5.7)
2	Access control	yes (1)	No (5)	yes (3)	No (6)	yes (6.08)	yes (8.9)	yes (5.3)	No (6.7)	yes (4.4)	Partial (5.7)	yes (1.5)	No (2.2)
3	Bad$_r$*andomness*	yes (3)	Yes (3.87)	Yes (3.7)	Partial (4)	yes (3.98)	No (6.9)	yes (6.8)	No (8.8)	yes (4.8)	No (5.7)	yes (6.2)	No (5.7)
4	Denial of service	Yes (2)	No (5)	yes (2)	No (3.5)	yes (7.23)	No (9)	yes (4.4)	No (6.8)	yes (4.6)	Partial (5.7)	yes (6.2)	No (1.2)
5	Arithmetic	yes (1.5)	No (4.5)	yes (3.5)	No (4.5)	yes (8.91)	No (9.7)	yes (7.2)	No (7.2)	yes (3.5)	No (7.2)	yes (5.1)	yes (5.7)
6	Front$_r$*unning*	yes (2.53)	No (3.67)	Yes (1.78)	yes (2)	yes (4.90)	No (6.9)	No (2.3)	No (6.74)	yes (6.5)	No (6.1)	yes (4.8)	No (4.9)
8	Short address	No (2)	Partial (2.5)	No (2.5)	Yes (2.5)	yes (2.2)	yes (5.1)	yes (5.6)	No (8.1)	yes (1.3)	No (5.5)	yes (7.5)	No (9.24)
9	unchecked call	No (3.4)	Partial (2.67)	yes (1.8)	No (2.8)	yes (2.1)	No (3.8)	yes (8.3)	No (4.9)	yes (5.2)	No (6.2)	yes (4.4)	No (7.45)
10	Time manipulation	yes (2.1)	Partial (2.89)	yes (3.4)	No (3.90)	yes (6.27)	No (9.7)	yes (1.89)	Yes (5.1)	yes (6.8)	No (8.1)	yes (3.5)	Partial (5.5)
11	Greedy contract	yes (1.8)	No (2.5)	yes (1.78)	No (4.6)	yes (1.29)	No (3.6)	yes (3.4)	No (4.7)	yes (4.2)	No (5.1)	yes (4.9)	No (6.54)
12	Prodigal contract	yes (1.89)	No (4.89)	yes (1.78)	No (3.78)	yes (3.26)	No (4.8)	yes (6.3)	No (4.9)	yes (6.1)	Yes (7.9)	yes (5.1)	Partial (8.25)

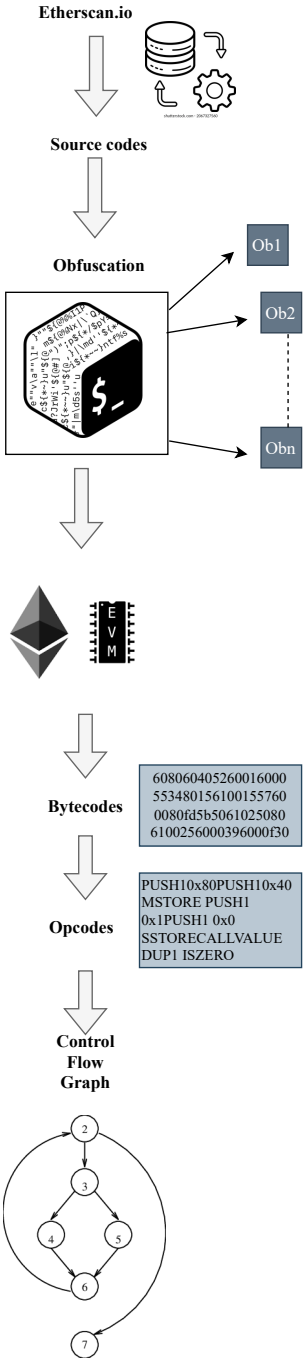


Figure 5: Flow chart

References

[1] GitHub - Our4514/BiAn — github.com. https://github.com/Our4514/BiAn. [Accessed 05-Dec-2022].

[2] Solidity x2014; Solidity 0.8.19 documentation — solidity.readthedocs.io. https://solidity.readthedocs.io/. [Accessed 06-Apr-2023].

[3] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *International symposium on automated technology for verification and analysis*, pages 513–520. Springer, 2018.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.

[5] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19, 2005.

[6] Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, et al. Smart contract languages: a comparative analysis. *arXiv preprint arXiv:2404.04129*, 2024.

[7] Davi Pedro Bauer. Solidity. In *Getting Started with Ethereum: A Step-by-Step Guide to Becoming a Blockchain Developer*, pages 13–16. Springer, 2022.

[8] Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, and Xiapu Luo. Sadponzi: Detecting and characterizing ponzi schemes in ethereum smart contracts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(2):1–30, 2021.

[9] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. Wasai: uncovering vulnerabilities in wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 703–715, 2022.

2025. In Sandeep Kumar & Kavita Sharma (eds.), *Computational Intelligence and Machine Learning*, 57–84. Computing & Intelligent Systems, SCRS, India.
DOI: https://doi.org/10.56155/978-81-975670-5-6-6

[10] Frederick B Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.

[11] Lee Song Haw Colin, Purnima Murali Mohan, Jonathan Pan, and Peter Loh Kok Keong. An integrated smart contract vulnerability detection tool using multi-layer perceptron on real-time solidity smart contracts. *IEEE Access*, 2024.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[13] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 127–137. IEEE, 2021.

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

[15] Zhiqiang Ding, Yiting Ma, Jiaqi Zhang, Jian Liu, Yue Xiong, Bo Li, and Jing Zhao. Security and privacy enhancements of software obfuscation techniques. *IEEE Access*, 9:1–38, 06 2020.

[16] Stephen Drape. Intellectual property protection using obfuscation. 2010.

[17] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[18] Fatemeh Ganji, Shahin Tajik, Jean-Pierre Seifert, and Domenic Forte. Blockchain-enabled cryptographically-secure hardware obfuscation. *Cryptology ePrint Archive*, 2019.

[19] Warren A Harrison and Kenneth I Magel. A complexity measure based on nesting level. *ACM Sigplan Notices*, 16(3):63–74, 1981.

[20] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum

virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[21] Kakelli Anil Kumar, Aena Verma, and Hritish Kumar. Smart contract obfuscation technique to enhance code security and prevent code reusability. 2022.

[22] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 2022.

[23] Zhaoxuan Li, Siqi Lu, Rui Zhang, Rui Xue, Wenqiu Ma, Rujin Liang, Ziming Zhao, and Sheng Gao. Smartfast: an accurate and robust formal analysis tool for ethereum smart contracts. *Empirical Software Engineering*, 27(7):1–52, 2022.

[24] TJA McCabe. complexity measure ieee trans. on software engineering, vol, 1976.

[25] Eva Meyer, Isabell M Welpe, and Philipp G Sandner. Decentralized finance—a systematic literature review and research directions. ECIS, 2022.

[26] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*, pages 1–4. IEEE, 2018.

[27] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Pearson Education, 2009.

[28] Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)*, 2008.

[29] Reza M Parizi, Ali Dehghantanha, et al. Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *International Conference on Blockchain*, pages 75–91. Springer, 2018.

[30] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[31] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*, 2019.

[32] Matthias Schenk. Practical inlining in viper. 2022.

[33] Charles Simonyi. Hungarian notation. *MSDN Library, November*, 1999.

[34] Sora Suegami. Smart contracts obfuscation from blockchain-based one-time program. *Cryptology ePrint Archive*, 2022.

[35] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113. IEEE, 2018.

[36] Michael Joseph Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[37] Carl R Worley and Anthony Skjellum. Opportunities, challenges, and future extensions for smart-contract design patterns. In *International Conference on Business Information Systems*, pages 264–276. Springer, 2018.

[38] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. On secure and usable program obfuscation: A survey. *arXiv preprint arXiv:1710.01139*, 2017.

[39] Xun Yi, Xuechao Yang, Andrei Kelarev, Kwok Yan Lam, and Zahir Tari. Bitcoin, ethereum, smart contracts and blockchain types. In *Blockchain Foundations and Applications*, pages 25–65. Springer, 2022.

[40] Meng Zhang, Pengcheng Zhang, Xiapu Luo, and Feng Xiao. Source code obfuscation for smart contracts. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 513–514. IEEE, 2020.

[41] Pengcheng Zhang, Qifan Yu, Yan Xiao, Hai Dong, Xiapu Luo, Xiao Wang, and Meng Zhang. Bian: smart contract source code obfuscation. *IEEE Transactions on Software Engineering*, 2023.

[42] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.