BiAn: Smart Contract Source Code Obfuscation

Pengcheng Zhang, *Member, IEEE*, Qifan Yu, Yan Xiao, Hai Dong, *Senior Member, IEEE*, Xiapu Luo, Xiao Wang, and Meng Zhang

***Abstract*—With the rising prominence of smart contracts, security attacks targeting them have increased, posing severe threats to their security and intellectual property rights. Existing simplistic datasets hinder effective vulnerability detection, raising security concerns. To address these challenges, we propose *BiAn*, a source code level smart contract obfuscation method that generates complex vulnerability test datasets. *BiAn* protects contracts by obfuscating data flows, control flows, and code layouts, increasing complexity and making it harder for attackers to discover vulnerabilities. Our experiments with buggy contracts showed an average complexity enhancement of approximately 174% after obfuscation. Decompilers Vandal and Gigahorse had total failure rate increments of 38.8% and 40.5% respectively. Obfuscated contracts also decreased vulnerability detection rates in more than 50% of cases for ten widely-used static analysis detection tools.**

***Index Terms*—Blockchain, Ethereum, smart contract, source code, obfuscation.**

I. Introduction

SMART contracts are autonomous programs that run on the blockchain. They are developed in several high-level languages and then compiled into bytecode [1]. Contracts are deployed by packaging bytecode in the form of transactions into the blockchain. Both blockchain and smart contracts are in their infancy, and smart contracts usually handle transactions related to cryptocurrencies. Consequently, attacking smart contracts is feasible and profitable [2]. In the past few years, there have been several serious attacks caused by smart contract errors, which have resulted in huge financial loss. The most notorious of these is the DAO [3] attack, which caused users' financial loss of 60 million US dollars.

Manuscript received 23 November 2022; revised 18 July 2023; accepted 19 July 2023. Date of publication 27 July 2023; date of current version 19 September 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 62272145 and U21B2016, in part by the Fundamental Research Funds for the Central Universities of China under Grants B220202072 and B210202075, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20191297, and in part by the CloudTech RMIT Green Bitcoin Joint Research Program/Laboratory and the Cooperative Research Centres Projects (CRC-P) funding scheme "Fast and Secure Crypto Payments for E[1]Commerce Merchants" under Grant CRCPXIII000145 and Hong Kong ITF Project (No. GHP/052/19SZ). Recommended for acceptance by J. Sun. *(Corresponding authors: Pengcheng Zhang; Yan Xiao.)*

Pengcheng Zhang, Qifan Yu, Xiao Wang, and Meng Zhang are with the College of Computer and Information, Hohai University, Nanjing 210000, China (e-mail: pchzhang@hhu.edu.cn).

Yan Xiao is with the School of Cyber Science and Technology, Sun Yat-sen University, Shenzhen 518000, China (e-mail: xiaoy367@mail.sysu.edu.cn).

Hai Dong is with the School of Computing Technologies, RMIT University, 3000 Melbourne, Australia (e-mail: hai.dong@rmit.edu.au).

Xiapu Luo is with the Department of Computing, Hong Kong Polytechnic University, HongKong 999077, China (e-mail: csxluo@comp.polyu.edu.hk).

Digital Object Identifier 10.1109/TSE.2023.3298609

The reason behind this kind of attacks is that smart contracts cannot be modified after being released. If a serious vulnerability is found after release, the contract needs to be replaced with a new contract and redeployed. Meanwhile, users' whole records will be lost. To ensure the security of this code, smart contracts must be tested before release.

As a result, various smart contract vulnerability testing tools have flourished. These tools perform vulnerability detection on either the source code level or the bytecode level. For example, SmartCheck is a source-code-level static smart contract vulnerability detection tool. It has the highest running efficiency among all the available smart contract vulnerability detection tools [4]. The most representative bytecode-based static detection tool for smart contracts is Slither [5], which has the largest number of users and is still under maintenance. It performs detection by converting Solidity smart contracts into an intermediate representation called SlithIR.

Although many smart contract static detection tools have emerged, smart contract incidents are still persistent. For example, in 2021, Chainswap[1] lost $4 million dollars due to its smart contract security issues. These incidents indicate that smart contract security protection mainly based on vulnerability detection still has not reached its full potential. The limitations of the current smart contract vulnerability protection include:

- *Lack of smart contract obfuscation tools for reverse engineering prevention.*
  Lack of smart contract obfuscation tools for reverse engineering prevention. Smart contracts on Ethereum may require obfuscation to protect intellectual property, mask code defects, and prevent leakage of business logic [6]. However, reverse engineering the publicly accessible bytecode of these contracts undermines these efforts. Solidity, the language for smart contracts, lacks dedicated obfuscation tools to prevent reverse engineering.
- *Lack of sufficient evaluation on complex smart contracts.* Most detection tools have only been tested on smart contracts with simple structures. The complexity of those smart contracts is far lower than that of real-world smart contracts in terms of cyclomatic complexity, complexity of data flow, etc.

Our goal is to design a smart contract source code obfuscation technique to address the above limitations. Our approach focuses on Solidity source code obfuscation by designing language-specific data flow obfuscation, control flow obfuscation and layout obfuscation techniques. The motivation

[1]https://www.163.com/dy/article/GEQC6RF30512D03F.html

behind this goal is that code obfuscation will significantly increase the complexity of the smart contract source code. After the obfuscation, the number of paths in the contract code will be greatly increased compared to the original code, while the main structure of the original code remains unchanged.

Motivation: First explain why code obfuscation technology is used in smart contracts:

Even though transparency is expected in smart contracts, and the motto "code is law" implies that the code should be clear and unambiguous. However, code obfuscation does not necessarily contradict this principle. Code obfuscation technology is used in smart contracts for several reasons.

- **Security:** Smart contract security is becoming an increasingly important issue, particularly with the development of Ethereum. Attackers are exploiting vulnerabilities in smart contracts, causing significant economic losses. Decompilation tools are also emerging, which can pose a number of threats to smart contract codes.
  First, the core code of the smart contract may be decompiled and leaked, resulting in being stolen by competitors. Second, the attacker may decompile and then insert malicious code, pretending to be the original program, and performing malicious acts. Third, the exposure of smart contract source code makes it easier for attackers to mine software vulnerabilities, making it more vulnerable to attacks.
- **Intellectual Property Protection:** Code obfuscation can be used to protect the intellectual property of the smart contract developer or to hide implementation details that are not relevant to the contract's functionality. In these cases, the obfuscation is not intended to obscure the logic of the contract or make it difficult to understand - it is simply a way to protect sensitive information.
  Let us assume that Jerry is the leader of an Ethereum smart contract development team. He is aware of safeguarding the original code of a contract 'getWageNumber.sol' (see Fig. 4a) developed by his team from public access. Such precautions are motivated by apprehensions surrounding the preservation of intellectual property rights. The original code of the smart contract contains informative comments that explain important aspects of the code. It uses meaningful variable names, such as 'DailyWage', which could easily be exposed to potential attackers, making the code readable, analyzable, and vulnerable to tampering. This poses significant risks to the security and intellectual property of the code.
- **Simple Data Set Issue:** The last issue is the problem of relatively simple data sets in the field of smart contract vulnerability detection. While the number and types of vulnerabilities that can be detected are increasing, the current data sets are generally too simple to accurately reflect the complexity of real smart contracts on Ethereum. This makes it difficult to train vulnerability detection tools that can accurately identify complex vulnerabilities. Code obfuscation can help to address this issue by increasing the complexity of the contract without changing its logical function [7]. We propose a convenient alternative: replace the required complex smart contract with an obfuscated vulnerability contract, thereby alleviating the problem of simple data sets.

The above points highlight the potential benefits of applying code obfuscation technology to smart contracts. The following will explain the feasibility, challenges and significance of applying code obfuscation technology to smart contracts.

First of all, code obfuscation technology is a well-established technique that has been widely applied in other programming languages, such as Java, C, Python, etc. Although direct application of this technology to the Solidity language is not possible, it is feasible to adapt code obfuscation techniques suitable for Solidity by analyzing the characteristics of the language.

Secondly, the challenge of increased gas consumption in smart contracts after code obfuscation is significant. Code obfuscation techniques inevitably lead to higher computational costs and increased gas consumption. To mitigate this negative impact, we have taken two approaches. First, we provide configuration files that allow users to select the desired obfuscation technology and intensity, which can reduce gas consumption and meet practical security needs. Second, we optimize gas consumption during the obfuscation process by using strategies that can lower gas consumption. As a result, the original average gas consumption after obfuscation increased from 100% to 82%, reducing the average gas consumption increment by 18%.

Finally, although developers are not required to upload source codes on Ethereum, many smart contract source codes are uploaded, and there will likely be more in the future. These uploaded source codes are exposed to the risks of analysis, exploitation, tampering, and vulnerability discovery, making it significant to apply code obfuscation technology to smart contract source codes.

Therefore, in this paper, we propose *BiAn*[2], a source code level code obfuscation tool, which is the first tool that is able to perform code obfuscation at the smart contract source code level. *BiAn* can modify the layout, data flow and control flow of the smart contract without affecting the original function of the smart contract. We conduct extensive experiments around *BiAn* to evaluate its performance.

In summary, we make the following contributions:

- We propose an improved chaotic mapping function, Chebyshev-PWLCM Map (CPM), by fusing two existing one-dimensional chaotic mapping functions, Chebyshev and PWLCM [8], in terms of trigonometric functions. CPM inherits the compelling features of Chebyshev and PWLCM, such as high sensitivity and randomness, and avoids their defects including smaller value ranges and existence of unchaotic points. CPM is demonstrated to be able to greatly enhance the generation quality of opaque predicates in control flow obfuscation.
- *BiAn* contains a set of dedicated obfuscation methods for the unique language features of Solidity. The proposed control flow, data flow and layout obfuscation methods are based on the special features of Solidity, including keywords, type names, third-party package dependencies, rows and columns of two-dimensional arrays,

[2]https://github.com/Nonreq/TSE2022

special structures (e.g., contracts, transfer, etc.), and special modifiers (e.g., view, pure, payable, etc.). In this way, we explore a new way of dataset generation to assess the performance of smart contract vulnerability detection tools. The obfuscated smart contract vulnerability datasets are more complex and closer to real smart contracts, which can address inadequacy and lack of complexity of existing testing datasets.

- *BiAn* is approved to enhance the anti-decompilation capacity of smart contracts. A decompiler can analyze and recover the assembly or source code format of the smart contract to make unauthorized use, analysis or vulnerability discovery. The anti-decompilation capacity of *BiAn* can improve the security of smart contracts.
- We conduct an extensive experimental evaluation for *BiAn*. The results of our experiments on complexity and gas consumption demonstrate a significant increase in the complexity of obfuscated smart contracts, with the number of paths being amplified by approximately 174%. In contrast, the gas consumption of the smart contracts after obfuscation rises by around 82%, which is considerably lower than the extent of complexity enhancement. Furthermore, we have assessed the performance of ten state-of-the-art static smart contract bug detection tools using the obfuscated smart contracts. Our findings reveal a substantial decrease in the detection accuracy of almost 100% of these tools. Additionally, the results obtained from evaluating decompilation resistance indicate that the overall failure rate of the Vandal and Gigahorse decompilers rises by approximately 40% after obfuscation.

The rest of this paper is organized as follows: Section II introduces background information in relation to this research, including Ethereum and smart contracts, program slicing, and code obfuscation. In Section III, the technical details of *BiAn* are provided. In Section IV, we use an open source buggy smart contract dataset to validate our method. Existing works are discussed in Section V. Finally, Section VI concludes the whole paper and plans future works.

II. Background

A. Ethereum and Smart Contracts

The Ethereum platform has encapsulated the blockchain technology. It allows blockchain applications to be directly developed on the Ethereum platform [9] and developers to fully focus on the application development without concerning about the underlying infrastructure, thus significantly reducing application development complexity [10]. At present, a more complete development ecosystem has been formed around Ethereum, including many available development frameworks and tools as well as community support[3].

Smart contracts are programs run on Ethereum, comprising a collection of code and data (state) [11]. Essentially, these automated contracts work like the if-then statements of other computer programs. Smart contracts simply interact with real-world assets in this way. When a pre-programmed condition is triggered, the smart contract executes the corresponding contract terms [12]. Smart contracts can be understood as contracts (special transactions) that are automatically executable (event-driven) and written in code on the blockchain [13]. In Bitcoin scripts, Bitcoin transactions are programmable, but Bitcoin scripts have many limitations and are limited in the number of programs that can be written [14]. In contrast, Ethereum is more complete (called “Turing-complete” in computer science), allowing users to write programs that can solve any reasonable computational problem [15]. Smart contracts are ideal for applications that require high levels of trust, security, and persistence, such as digital currencies, digital assets, voting, insurance, financial applications, prediction markets, property ownership management, the Internet of Things, peer-to-peer transactions, etc. [16].

[3]https://ethereum.org/en/whitepaper/ethereum

Algorithm 1 Split function

```
Input: $solidity\_source\_code$, $solidity\_json.ast$
Output: $program\ snippet$
1: Inital $solidity\_source\_code$, $solidity\_json.ast$;
2: while $solidity\_source\_code$, $solidity\_json.ast$ $!=$ Null do
3:     Find each $function$;
4:     FunctionList.append($function$);
5:     if $FunctionName\ != Null$ then
6:         Split to multiple $functions$;
7:     end if
8:     Reset $solidity\_source\_code$, $solidity\_json.ast$;
9: end while
```

B. Program Slicing

Program slicing usually consists of three parts: 1) *program dependency extraction* that mainly extracts various kinds of information from the program [17], including control flow and data flow information, to form a program dependency graph, 2) *slicing rule formulation* where slicing criteria are designed according to specific program analysis requirements [18], and 3) *slicing generation* in which the corresponding program is selected according to the aforementioned slicing criteria. A program slicing method should be determined by the preceding slicing guidelines based on the aforementioned slicing guidelines, and then analyzes and processes the dependency relations extracted from the first step to generate program slices [19]. We employ a split function for Solidity-based smart contract program slicing, which is shown in Algorithm 1. Its workflow is as follows. First, the source code of the smart contract and its corresponding json_ast file are fed into the algorithm. If the input is not null, each function in the source code is located and added to the function list. Finally, the split function generates the function snippets of the program.

C. Code Obfuscation

Code obfuscation has been proposed as a method to resist software reverse analysis [20]. Code obfuscation refers to the semantic transformation of a proposed application so that the transformed program is functionally identical or similar to the original program, but more difficult to be understood and decompiled. Collberg [21] classifies code obfuscation into three categories: *layout obfuscation*, *data flow obfuscation*, and *control flow obfuscation*.

- The principle of layout obfuscation is to remove irrelevant information from a program or to replace the class names and method names in the program so that it violates the software engineering principle of "knowing the meaning by name" [22].
- Data flow obfuscation is the obfuscation of the data domain and data structure of a program, including variable storage and coding obfuscation, variable aggregation and splitting obfuscation, and order adjustment obfuscation [23]. Data flow obfuscation does not intentionally modify the code of a program, but only transforms it to different data structures in the program.
- Control flow obfuscation is a widely used method of code obfuscation. Information about the control transformation process of a program is an important clue to track and locate the state of the program [24]. How to protect such information is an important part of software protection. The purpose of control obfuscation is to alter or complicate the control flow of a program to make it more difficult to decipher.

Control obfuscation has enhanced security protection than the other two obfuscation types, which has been the main research hotspot in the field of code obfuscation nowadays [25].

D. Definitions of Terminologies

The key terminologies used in this paper are defined as follows.

Obfuscating Transformation. Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{\tau} P'$ is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, for $P \xrightarrow{\tau} P'$ to be a legal obfuscating transformation, the following conditions must meet: if P fails to terminate or terminates with an error condition, then P' may or may not terminate; otherwise, P' must terminate and produce the same output as P [26].

Transformation Potency. Let T be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P, $T_{pot}(p)$, the potency of T with respect to a program P, is a measure of the extent to which T changes the complexity of P. It is defined as

$$T_{pot}(p) \overset{def}{=} E(P')/E(P) - 1 \tag{1}$$

T is a potent obfuscating transformation if $(p) > 0$ [26].

Chaos. Let J be a metric space with a continuous map $f : J \to J$ and let X be a set. The map f is viewed to be chaotic over J, given that the following three conditions are satisfied [27]:

- $f : J \to J$ is viewed to be topologically transitive if for any pair of open sets $U, V \subseteq J$ there exits $k > 0$, such that,
$$f^k(U) \cap V \neq 0 \tag{2}$$
- The periodic points of f are dense in X
- $f : J \to J$ has sensitive dependence on initial conditions if there exists $\delta > 0$, such that, for any $x \in J$ and any neighborhood N of x, there exists $y \in N$ and $n \geq 0$, such that
$$|f^n(x) - f^n(y)| > \delta \tag{3}$$

Opaque Predicate. A predicate P is opaque at P if its outcome is known before it is applied to a location in the program. We write $P_p^F (P_p^T)$ if P is always assessed to be False (or True) at a program point P, and $P_p^?$ if P may sometimes be assessed to be True (or False) [28].

III. Our Method

Our method targets Ethereum smart contracts encoded in Solidity, i.e., the most widely used smart contract programming language. The general architecture of our method is shown in Fig. 1. As can be seen from the figure, the operation of the obfuscation tool focuses on the source code of a smart contract.

The obfuscation operation is performed on the smart contract source program by constructing a control flow graph (CFG) to extract code blocks and program slices. The detailed design is divided into three parts, namely *control flow obfuscation*, *data flow obfuscation*, and *layout obfuscation*.

A. Control Flow Obfuscation

Control flow refers to the order in which instructions are executed during code execution. With various control logics, programs are executed along a specific logical sequence [29]. The general control logic includes branches, loops, function calls, etc. In normal circumstances, the logic of the program should be clear, and there might involve various human interventions in the development process to make the code logic clear and easy to maintain and extend [26]. However, in the case of decompilation prevention, clear code logic makes the code easy to be captured and accelerate the cracking process. Control flow obfuscation complicates the control flow of a smart contract by relying on the control flow graph during compilation, thus hiding the control flow of the original program. The module is composed of two parts: *inserting opaque predicates* and *flattening control flow graph*.

- Inserting opaque predicates refers to using opaque predicates to construct branching conditions to increase the number of directions for a control flow. In code obfuscation, if the value of a certain expression $P = F(I)$ (where P represents a certain expression value, F means a certain function where we can get an expression value and I represents the input space has been determined before embedding the program, the expression is considered opaque. It can prevent attackers from inferring values from expressions and/or understanding the programmer's intentions.

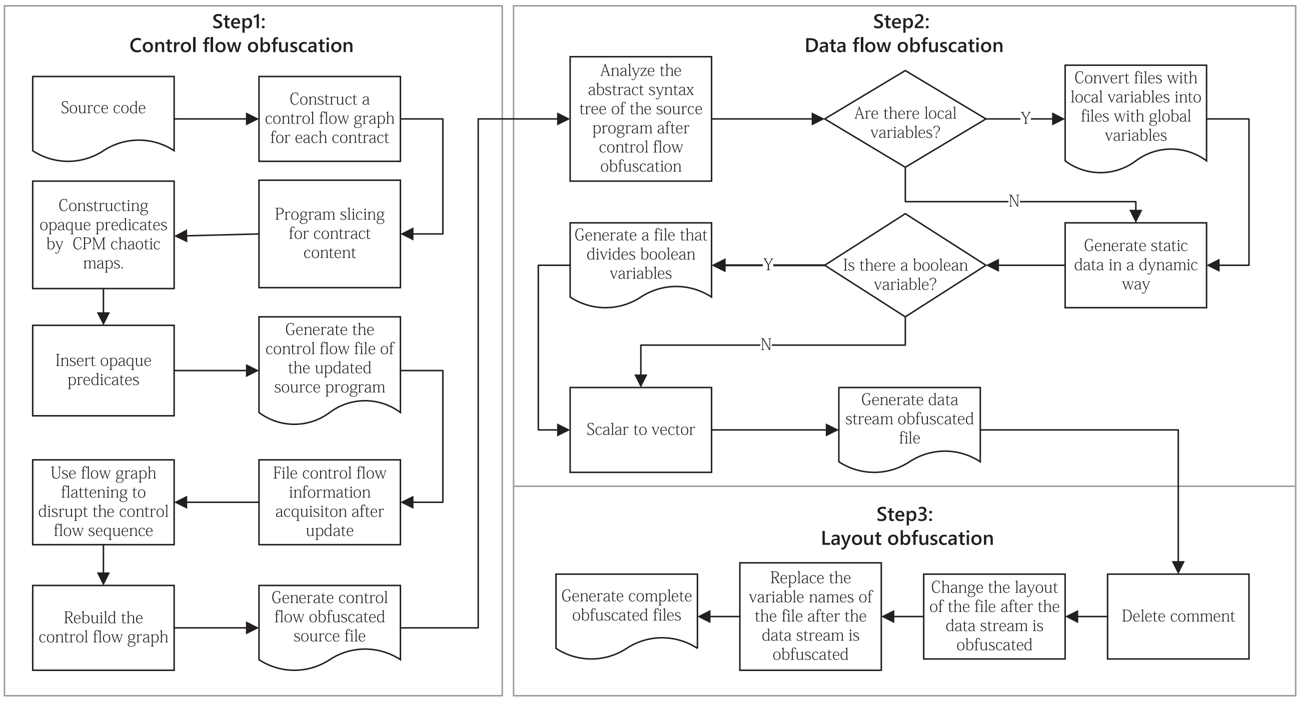

Step1:
Control flow obfuscation
Source code
Construct a control flow graph for each contract
Constructing opaque predicates by CPM chaotic maps.
Program slicing for contract content
Insert opaque predicates
Generate the control flow file of the updated source program
Use flow graph flattening to disrupt the control flow sequence
File control flow information acquisiton after update
Rebuild the control flow graph
Generate control flow obfuscated source file
Step2:
Data flow obfuscation
Analyze the abstract syntax tree of the source program after control flow obfuscation
Are there local variables?
Y
Convert files with local variables into files with global variables
N
Generate a file that divides boolean variables
Y
Is there a boolean variable?
Generate static data in a dynamic way
N
Scalar to vector
Generate data stream obfuscated file
Step3:
Layout obfuscation
Generate complete obfuscated files
Replace the variable names of the file after the data stream is obfuscated
Change the layout of the file after the data stream is obfuscated
Delete comment

Fig. 1. The overview of *BiAn*.

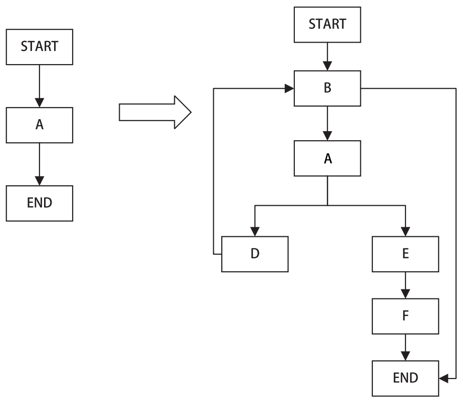


Fig. 2. Insertion of opaque predicates.

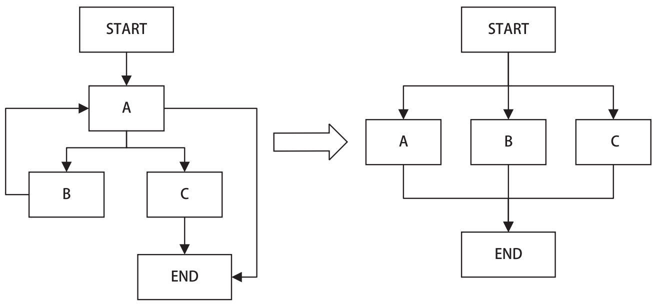


Fig. 3. Control flow graph flattening.

A sample insertion of opaque predicate is shown in Fig. 2, where predicates B, D, E, and F whose values are already pre-determined are inserted into the program.

- Flattening the control flow graph means re-breaking each basic block in the source program, jumping according to the identifier, and completely reconstructing the corresponding control flow graph to weaken the relationship between code blocks. Flattening control flow is the opposite of changing the structure of the source code. It destroys the internal logic of the program code block, which makes it difficult to perform static analysis, thereby increasing the difficulty of reverse engineering. The flattening of the control flow graph is shown in Fig. 3. As can be seen from the figure, A is a branch structure connecting code blocks B and C. After the control flow is flattened, the branch structure disappears and is replaced by a sequential structure, whereby the internal logical structure of the code block is destroyed.

Our control flow obfuscation method is optimized according to the following unique features of Solidity.

- Special keywords, e.g., modifier. The use of modifier can easily change the behavior of the function. For example, they can automatically check a certain condition before executing the function. The modifier is an inheritable property of the contract and may be covered by a derived contract. Therefore, in the case of obfuscation, it is necessary to find the execution position of the condition according to one or more conditions checked by the modifier. If we want to eliminate the influence of the modifier keyword, we need to first extract the prerequisites and adjust the position of the prerequisites.
- Internal function calls. These function calls are translated into simple jump statements in Ethereum Virtual Machine (EMV). The current memory cannot be cleared because the functions referenced by the memory are very efficient.

Algorithm 2 CPM chaotic mapping algorithm

```
Input: $X_n$, $\mu$, $p$, $M$
Output: $Opaque\ Predicates$
1:  if ($X_n \leq 1 \& X_n \geq 0$ ) and ($\mu > 2$) then
2:      for  i in $X_n$ do
3:          Substitute four parameters into the chaos equation;
4:          $Res$ = ChaosEquation($Xi$, $u$, $p$, $M$);
5:          List.append($Res$);
6:      end for
7:      Get List = $a1$, $a2$, , $an$;
8:      Remove the duplicate elements in List;
9:      $VauleList$ = BooleanValueFunction($List$);
10:     return $ValueList$;
11: else
12:     Print wrong parameters;
13:     Re-execute parameter input;
14: end if
```

Only functions in the same contract can be called internally. We choose to copy the function content in the internal calling function and paste it directly into the location of the called function. This makes the contract more complicated and interferes with and obfuscates the decompilation operation.

Opaque predicates are a very important part of the control flow obfuscation described above. In order to construct opaque predicates with higher quality and maximize the cost of cracking opaque predicates for attackers, we will use a new chaotic mapping-based method to construct opaque predicates below.

B. CPM Chaotic Mapping for Opaque Predicate Construction

This section presents a new algorithm for constructing opaque predicates and squashing control flow obfuscation, which bases on a CPM chaotic map. The main procedure of the CPM chaotic mapping is shown in Algorithm 2. The inputs of CPM chaotic mapping algorithm are X_n, μ, p, M, the description of which can be found from formula (7). First, it is required to judge the ranges of X_n and μ. If they are not within the specified ranges, the parameter range error will be prompted and the parameter input will be re-executed by requiring re-entry of parameters that fit the ranges. If they are within the specified range, each value of X_n will be traversed. For each iteration, the four parameters are fed into the CPM formula to obtain the corresponding result, where the result is appended to a list. Next, the corresponding Boolean value is obtained through the function of 'BooleanValueFunction'. The rules of this function can be varied according to actual application situations, so as to prevent attackers from analyzing the function rules and making corresponding cracks. The currently adopted rule is that the result is true when the value is closer to 1, and false when the value is closer to 0. Finally the algorithm returns a list of Boolean values.

1) Constructing Opaque Predicates: For the construction of opaque predicates, it is common to use mathematical tools to construct *True* and *False* values. For example, for any x in Z, the expression

$$(x^2 \geq 0)|x(x+1)(x+2) \tag{4}$$

is always real. However, this method can easily be cracked by decompilation tools. If an attacker can easily crack the opaque predicate generation method, it indicates that this method has very low protection ability for the code. Compared to its running and implementation cost, the benefit of the opaque predicate generation is low.

The chaotic mapping (depicted in Section II-D) can solve the problem above. Since the chaotic map has a high sensitivity to the initial value and can generate highly random results through iterations, the opaque predicate constructed on the chaotic map can effectively prevent the encrypted information from being cracked or increase the cracking difficulty. The more complex the opaque predicate, the harder it is to attack. The reference can be found in [25].

Based on the Chebyshev [30] and PWLCM [8] chaotic maps, we propose an improved chaotic map, named CPM. CPM not only retains the advantageous characteristics of the sensitivity and randomness of the two existing chaotic maps, but also weakens the insufficiency of the existence of breakpoints, leading to an improved performance. The technical details of CPM are discussed below.

The selection of a suitable chaotic map relies on the sensitivity, randomness and timeliness of a chaotic system. The sensitivity refers to small changes in initial values or related determined control parameters of a system leading to significantly different results [31]. In this way, it is difficult for an attacker to reversely deduce the initial parameter key of a chaotic map based on the opaque predicate generation result. The randomness refers to the random distribution of chaotic mapping in the metric space [32]. The timeliness refers to the speed of obfuscation. While multi-dimensional chaotic system-based encryption can generate high security, it is time-consuming [33]. Therefore, this paper mainly focuses on adopting low-dimensional chaotic maps for code obfuscation.

We select two existing one-dimensional chaotic mapping methods, i.e., Chebyshev and PWLCM, according to the aforementioned criteria.

The Chebyshev chaotic map has a high sensitivity to its initial values and long-term unpredictability for its chaotic sequences [30]. In particular, with the continuous increase in the number of chaotic map iterations, the Chebyshev chaotic map can have uniformly distributed chaotic trajectories. The Chebyshev chaotic map is defined as:

$$x_{n+1} = \cos\left(\delta \arccos\left(x_n\right)\right), \quad x_n \in [-1, 1] \tag{5}$$

where δ is the control parameter of the chaotic map. When $\delta >= 2, \quad x_n \in [-1, 1]$, the Chebyshev chaotic map is in a chaotic state. Its sensitivity to initial values and unpredictability are high, but its chaotic range is slightly small [30].

PWLCM (Piece Wise Linear Chaotic Map) is an intuitive and clear piecewise chaotic map [8]. The chaotic map also has high initial value sensitivity and randomness. The chaotic trajectories of PLWCM map are uniformly distributed. It is expressed as:

$$x_{n+1} = \begin{cases} x_n/\gamma & 0 \le x_n < \gamma \\ (x_n - \gamma)/(0.5 - \gamma) & \gamma \le x_n < 0.5 \\ 0 & x_n = 0.5 \\ F(1 - x_{n-1}, \gamma) & 0.5 < x_n < 1.0 \end{cases} \quad (6)$$

where γ is the control parameter of the chaotic map. When $\gamma \in (0, 0.5),\ x_n \in [0, 1)$, PWLCM is in a chaotic state. Although the chaotic range of PWLCM is wider in comparison to the Chebyshev chaotic map, there is a point $(x = 0.5)$ where the function value is zero. When it is at this point, PWLCM will lose its chaotic effect, and the security and continuity of the chaotic system are discontinued.

In view of the aforementioned advantages and disadvantages of the Chebyshev and PWLCM chaotic maps, we fuse the two chaotic maps by trigonometric functions and propose an improved one-dimensional chaotic map, Chebyshev-PWLCM Map (CPM). The definition of the CPM chaotic map is:

$$x_{n+1} = \begin{cases} \bmod\left(\cos\left(M\cos\left(\mu\arccos\left(x_n\right)\right) + x_n/p\right), 1\right), \\ 0 \le x_n < p \\ \bmod\left(\sin\left(\cos\left(\mu\arccos\left(x_n\right)\right) + \right.\right. \\ \left.\left.\left(x_n - p\right)/\left(0.5 - p\right) + M\right), 1\right), \\ p \le x_n < 0.5 \\ F\left(1 - x_n, \mu, p\right), 0.5 \le x_n < 1 \end{cases} \quad (7)$$

When $\mu \ge 2, p \in (0, 0.5)$, the CPM chaotic map is in a chaotic state, M belongs to the part of the key in the chaotic map and it is the disturbance parameter of the chaotic map. The CPM chaotic map retains the advantages of high initial value sensitivity and randomness of Chebyshev and PWLCM chaotic maps, expands the chaotic range, and increases the security and continuity of chaotic maps by eliminating the limitation of zero function value of PWLCM. In addition, CPM goes a step further to distribute the chaotic trajectories more evenly, which improves the performance of chaotic mapping. In this way, the opaque predicates can be constructed by chaotic mapping.

2) Insertion of Opaque Predicates: Three forms of opaque predicates are introduced in Section II-D: never-true opaque predicates, never-false opaque predicates, and true or false opaque predicates. After constructing the opaque predicates, we can design these three opaque predicates as needed, and insert them into the Solidity programs where the truth needs to be judged, such as if-else or while statements. Opaque predicates can also be used in ordinary statements that are executed sequentially.

When inserting a never-true opaque predicate into an if-else statement, it is necessary to perform a logical AND operation on the original conditional judgment expression in the if-else statement and the never-true opaque predicate, according to the grammar rules. In this way, if the result of the original conditional judgment expression is true, the output result is still true after the logical AND operation; if the result generated by the original conditional judgment expression is false, after the logical AND operation, the output result is still false. Since the logic AND operation itself has short-circuit characteristics, when the original condition is true, the true or false of the always true opaque predicate will be detected; when the original condition is false, the true or false of the never true opaque predicate will not be checked. The time overhead of computing never-true opaque predicates is reduced to a certain extent.

3) Algorithm Execution Description: The explanation of the squashing control flow algorithm can be referenced in Section III-A. The source code, chaotic map, obfuscation method and save path are the input of the algorithm. The integrity of the input parameters is checked. If the parameter format is correct, the opaque predicates will be generated and inserted into smart contract source code and the control flow of the code will be flattened; otherwise these functions cannot be executed. Finally, the obfuscated code will be generated.

C. Data Flow Obfuscation

Data flow obfuscation refers to modifying the data fields in a program without processing the internal logic structure of the program [34]. It modifies data fields by analyzing the abstract syntax tree (AST), which is a tree-like representation of the abstract syntax structure of the source code. Each node on the tree represents a structural component in the source code. It is abstract because the abstract syntax tree does not represent the real syntax. The modification of data fields together with detailed information in a smart contract would make an attacker more difficult to obtain valid information from the smart contract The process of realizing this feature is divided into five tasks: 1) *converting local variables into global variables*, 2) *converting static data to dynamic data*, 3) *transforming integer constants into arithmetic expressions*, 4) *splitting Boolean variables* and 5) *converting scalar variables into vectors*. The technical details of these tasks are depicted below:

- First, we obtain all the declared local variables by parsing the json ast file. According to all the local variables declared, we search for local variables with the same names. Then, we overwrite all the variables with the same variable names to make them as global variables, which constructs the foundation for the following redeclaration and removal of the state of the local variables. Second, the state variables corresponding to the local variables are redeclared and the declaration statements of the local variables are removed. Finally, where each local variable used is located according to the variable id, and the original local instruction is replaced with the newly declared variable. Some unique operations are made based on the unique features of Solidity. There are two types of functions in Solidity, namely pure function and view function. It is not allowed to read or modify the state variable in a pure function, while it is not allowed to modify the state variable in a view function. Therefore, *BiAn* does not handle local variables in pure functions. The detailed steps for converting local variables to global variables can be found in Algorithm 3. To illustrate this algorithm, let's consider a smart contract where a global variable is declared outside any function, such as *'uint256 public globalVariable;'*. For

Algorithm 3 Converting local variables to global variables

```
Input: $solidity\_source\_code$, $local\_variables$
Output: $solidity\_source\_code$, $global\_variables$
1: Initial Smart Contract Source Code;
2: Find $localVar\_Post$;
3: while $Find\_ASTNode$ $!= Null$ do
4:     for $i = 1$ $to$ find $localVar.length$ do
5:         if Process same name $!= Null$ then
6:             Find $same\ name\ state$;
7:             Over write $declare\_\ state$;
8:         end if
9:         Get $corpus$;
10:        $Str\ replacedLocalVariable$;
11:        Reset $Sol\ And\ Json$;
12:    end for
13: end while
```

other local variables within code blocks, assign the local variable to the global variable using *'globalVariable = localVariable;'*. This assignment changes the state of the local variable to match the value of the global variable.

- Static data is converted into dynamic data based on four types of constants in Solidity, namely integer, Boolean, string and hexadecimal string. Our tool first obtains the positions and corresponding values of the four types of constants by parsing the json ast file. Next, we declare a new array based on the collected values for each type of constant, i.e., the arrays for the constants of integer, Boolean, string and hexadecimal string respectively. We then devise a function to return the corresponding elements from each array. Finally, the constant is replaced with the corresponding function call. We will not try to convert constants involving capital transactions (such as require and assert statements) into dynamic variables. This is because users need these constants to observe their transactions in real-time.
- All integer constants in smart contract source files are converted into complex arithmetic expressions (i.e., expressions that can generate original values). Since Solidity does not support floating-point numbers, the generated arithmetic expressions do not contain floating-point numbers or use division.
- Splitting Boolean variable is to approximate all the Boolean constants in the source code and add a suffix after the Boolean constant. The splitting rule is: if the original Boolean constant is true, the "||" operator is adopted to connect with subsequent expressions. The subsequent expression may be a Boolean expression or an arithmetic expression. If the original Boolean constant is false, the "& &" operator is used to connect with subsequent expressions. The subsequent expression may be a Boolean expression or an arithmetic expression. The procedure for splitting Boolean variable is introduced in Algorithm 4. This algorithm involves splitting boolean variables. For instance, if we have a variable with a value of 'true', it can be modified to *'true || false'* or *'true & & true'*. Both alternatives yield the same result as the original value but provide an additional level of complexity.

Algorithm 4 Split boolean variables

```
Input: $solidity\_source\_code$, $boolean\_variables$
Output: $solidity\_source\_code$,
    $splited\_boolean\_variables$
1: while $boolList$ $!= findBoolList$ do
2:     if $len(boolList) == 0$ then
3:         return $content$;
4:     else
5:         Find $Variable\_Declaration\_Statement$,
           $Assignment$, $VariableDeclaration$;
6:         for $statement$ in $statementList$ do
7:             $splitBoolVariable$ and $append$;
8:         end for
9:     end if
10:    Replacing $hash\ values$ with $variable\ names$;
11:    Reset $Sol\ And\ Json$;
12: end while
```

Algorithm 5 Variable name replacement

```
Input: $solidity\_source\_code$, $solidity\_json.ast$
Output: $solidity\_source\_code$
1: while $Identifier$ $!= Null$ do
2:     Store the location of the matching $identifier$;
3:     if $Identifiers\ do\ not\ involve\ money\ transactions$
       then
4:         $SHA{-}1$ hash algorithm generates $hash\ values$;
5:     end if
6:     Replacing $hash\ values$ with $variable\ names$;
7:     Reset $Sol\ And\ Json$;
8: end while
```

- The scalar-to-vector function is to collectively declare the state variables of integer, Boolean, address, string, and bytes in a structure, and to enable the member variables to be called through the structure. All initialization, assignment, and use of the original state variables will be replaced with those of the corresponding structure member variables. The steps of variable name replacement are shown in Algorithm 5. Consider a variable named 'gasConsumption', its corresponding SHA-1 value can be contained by: '$f0eb29ec79e2b6f94bc5a4266012b74b21eb6181$'. This cryptographic hash function provides a unique and irreversibly transformed value for the original variable. This kind of operations will force the call and generation of state variables to be connected with the structure, making the directions of data flows more complicated.

D. Layout Obfuscation

Layout obfuscation refers to removal or obfuscation of auxiliary text information in software source code or intermediate code that is not related to execution, making it more difficult for an attacker to read and understand the code [35]. The procedure of layout obfuscation is shown in Algorithm 6.

Algorithm 6 Layout obfuscation

```
Input: $file\_Path$, $json\_File$
Output: $solidity\_source\_code$
1: if $len$ != 3 then
2:     print: wrong parameters;
3: else
4:     Get $configuration$, $file\ content$;
5:     if $comment.length$ != 0 then
6:         Delete $comment$;
7:     end if
8:     Disrupt $format$;
9:     for $variable\ name$ in $source\ code$ do
10:        while $identifiers$ != $Null$ do
11:            Store the location of the matched $identifiers$;
12:            if $identifiers\ do\ not\ involve\ money\ transactions$ then
13:                $SHA-1$ hash algorithm generates $hash\ values$;
14:            end if
15:            Replace $hash\ values$ with $variable\ names$;
16:            Reset $Sol\ And\ Json$;
17:        end while
18:    end for
19: end if
```

It mainly implements scrambling identifiers and debugging information in the code to increase the workload of a reverse attacker to read or analyze. It mainly implements three functions: *deleting comments*, *scrambling layout*, and *replacing variable names*. Deleting comments and scrambling layout specifically focus on identifying the content in the source code through regular expressions and replacing the content with corresponding meaningless characters, with the purpose of enhancing the difficulty to code content parsing. Replacing variable names is performed by parsing the json ast file, obtaining the name and corresponding location of each identifier in the source code, and then using the SHA1 hash algorithm to generate the hash value of each identifier. The reason for choosing the SHA1 hash algorithm is that the output of the algorithm is a hash value of 160 bits in length, which is the same as the address type provided by Solidity. The identifiers in the source file are then replaced by using "OX" plus the hash value corresponding to each identifier. Although the SHA1 algorithm can be cracked, cracking the "variable name" is actually valueless.

IV. Experiment

Our experiment is conducted using a dataset of 1,000 buggy smart contracts randomly obtained from the Ethereum chain and other public sources, including the open-source Ethereum smart contract security vulnerability test case set SWC Registry[4] provided by Smart Contract Security, the Jiuzhou [3] smart contract security vulnerability data dataset published by Xiao et al., the vulnerability-marked smart contract security vulnerability dataset SB Curated[5], Ethereum ETL project[6] and test data samples from smart contract tools such as Slither, Smartcheck, Solhint, and Mainticore. SWC Registry is an essential knowledge base for Ethereum security personnel and developers. It contains descriptions and consequences of common security issues in the development of Ethereum Solidity smart contracts, such as reentrancy, overflow, etc., It also provides CWE (Common Weakness Enumeration) Vulnerability classification, solutions and contract program code as examples. Jiuzhou is a collection of statistics and classification of Ethereum smart contracts. It also provides a brief introduction, solutions and test cases for each bug, which helps smart contract developers or researchers understand the current security state of Ethereum and get a benchmark dataset for testing smart contract analysis tools. SB Curated is a curated dataset containing 143 vulnerability-labeled smart contracts with 208 labeled vulnerabilities. It is used to evaluate the accuracy of analytical tools. In comparison to the above two datasets, SB Curated's vulnerability markers are more accurate in terms of the range of each vulnerability in each contract. In addition, in order to broadly collect the required smart contract data with vulnerabilities, we used related keywords such as "Smart Contract Bugs", "Smart Contract Vulnerabilities" and "Smart Contract Security" to search for valid data in IEEE and ACM libraries and Github. A total of 183 relevant papers or projects were retrieved. Next, we manually checked the rationality and logical correctness of the relevant data. We manually recorded the location of the corresponding vulnerability in the smart contracts with reference to the vulnerability marker of SB Curated, and removed some duplication of functions and derived data. Finally, we collected 1,000 smart contracts with vulnerabilities, including nine vulnerability data sets (privilege control vulnerability (160), reentrant vulnerability (167), integer overflow (106), randomness vulnerability (79), timestamp dependency vulnerability (150), unchecked return value vulnerability (46), denial of service vulnerability (141), front-running transaction vulnerability (55), unknown function vulnerability (96)). We also collect a set of real smart contracts run on Ethereum. These real contracts are randomly selected using etherscan[7], which is a blockchain browser based on Ethereum. The 109 actual smart contract datasets are utilized in the complexity experiment, where we employ the box plot and Cohn's d value to demonstrate that the complexity of the obfuscated smart contract is closer to the real smart contract.

All the obfuscated smart contracts in the dataset can be compiled successfully. Each vulnerable smart contract in the vulnerability dataset contains only one type of vulnerability. The compiler version numbers of the smart contracts range from $0.4.\times$ to $0.8.\times$. For the current dataset of 1,000 smart contracts, the average number of if statements is 6.2, the average number of loops is 4.1, and the average number of lines of code is 493.7.

The primary purpose of the experiments is to explore the following research questions:

- *RQ1*: Is the chaotic performance of the CMP chaotic map better than the existing ones?

[4]https://swcregistry.io/
[5]https://smartbugs.github.io/
[6]https://github.com/blockchain-etl/public-datasets
[7]https://cn.etherscan.com/

- *RQ2*: Are the code functions in smart contracts unchanged after using the *BiAn* tool to obfuscate the code?
- *RQ3*: Are the obfuscated smart contract more resistant to decompilation?
- *RQ4*: How the complexity of smart contracts is impacted after being obfuscated by *BiAn*?
- *RQ5*: How gas consumption of smart contracts is affected after being obfuscated by *BiAn*?
- *RQ6*: How the performance of the existing smart contract bug detection tools is varied after using *BiAn*?

We use *BiAn* to generate obfuscated contracts for the set of buggy contracts. A sample smart contract before and after the obfuscation is shown in Fig. 4. All the obfuscated contracts can be compiled with *solc* to generate bytecode. We recruit a group of developers to verify the functional variation of the contracts after the obfuscation (Section IV-B). We also conduct experiments to find out how the obfuscated contract can resist the mainstream decompilation tools (Section IV-C), the complexity variation (Section IV-D) and the gas consumption (Section IV-E) of the obfuscated contract. In addition, we manually label the error locations in the obfuscated contracts and build a public buggy contract dataset. This dataset is used to assess how the obfuscated contracts challenge state-of-the-art static smart contract vulnerability detection tools (Section IV-F).

A. Chaos Mapping Performance

To answer *RQ1*, this experiment selects two performance indicators (chaotic bifurcation diagram and Lyapunov exponent [36]) to verify the superiority of the CPM chaotic map over the traditional Chebyshev and PWLCM chaotic maps.

The chaotic bifurcation diagram reflects the state change of a chaotic system, indicating whether a system has chaotic behavior under each parameter. The more states a system can enter under the same parameter, the better its chaotic performance. In order to control the variables, the PWLCM, Chebyshev and CPM chaotic maps all use the same initial value.

The chaotic bifurcation diagram of three chaotic maps is shown in Fig. 5. Compared with the other two chaotic maps, the bifurcation diagram of the Chebyshev chaotic map has a slightly smaller chaotic range. In the diagram of the PWLCM chaotic map, there are certain intervals where the chaotic behavior is lost. In contrast, the points of the CPM chaotic map can cover the entire space and are more evenly distributed within the parameter range.

The Lyapunov exponent is an important index to measure the initial value sensitivity of a chaotic system. It quantifies the separation rate between infinite and near orbits in a dynamical system. The maximum Lyapunov exponent of a system determines the main evolution trend of the system. The larger the Lyapunov exponent, the worse the local stability of the system. Therefore, whether the maximum Lyapunov exponent of a system is greater than zero can be used to judge whether the system is in chaos.

The Lyapunov exponents of the three chaotic maps are shown in Fig. 6. The Lyapunov exponents of the CPM chaotic map are all greater than 0, which indicates that they have high

```
pragma solidity 0.6.2;
/*
Solidity doesn't support floating point Numbers very well, and all integersthat are
divided are rounded down. The use of integer division to calcul-ate the amount of
ethers may cause economic losses.
*/
    contract getwageNumber {
    uint256 public coefficient;
    uint256 public DailyWage;
    address public boss;

    constructor() public{
    DailyWage = 100;
    coefficient = 3;
    boss = msg.sender;
    }

    modifier onlyowner{
        require(msg.sender == boss);
        _;
    }

    function setDailywage(uint256 _wage) external onlyowner{
        Dailywage =_wage;
    }

    function setcoefficient(uint256 _co)external onlyowner{
        coefficient =_co;
    }

    function calculatewage(uint256 dayNumber)external view onlyowner returns(uint256)
    {
        //Until now, Solidity doesn't support decimals or fixed-point numbers, and
        all integer divisionresults are rounded down, which can lead to a loss of
        accuracy. Avoid using integer division to calculatethe amount of ethers. If
        you have to, try multiplying before dividing to offset the loss of accuracy.
        uint256 basewage = Dailywage / coefficient;
        return baseWage * dayNumber;
    }
}
```

(a) Smart contract before obfuscation.

```
pragma solidity 0.6.2;contract 0x98923cc55cbcd5bd396ad8416b3622500b16e07b{constructor
()public{0xb852185ad6cc3fb5baf4401da89c5ca111f8ba7e.0x8a9af61f0afc93a1f990b838080f8b2
d71c7da92=uint256(0x590599ec111c7fc88c9e703311afe5dc1c9a06f2(0 +(9* 6 )* 9-486));0xb
852185ad6cc3fb5baf4401da89c5ca111f8ba7e.0x3ca5a127725b250dc0ca365c87f6fe7c2c44ce6e=
uint256(0x590599ec111c7fc88c9e703311afe5dc1c9a06f2(8 * 6 * 6 + 5-292));0xb852185ad6c
c3fb5baf4401da89c5ca111f8ba7e.0x08bd1e2ac9fa91e3e4d821821297330d97e852f1 = msg.sender
;}modifier 0x0b8d7cc91fd21fdfa9558e31a58f72d5b1b084ce{require(msg.sender ==
0xb852185ad6cc3fb5baf4401da89c5ca111f8ba7e.0x08bd1e2ac9fa91e3e4d821821297330d97e852f
1);_;}functionOxdbf2268deca82cedb034a1c6446b66763aed8016(uint2560x6503f9dbbaod7f77e3
84965d35cf28c3c19a4b7f)external0x0b8d7cc91fd21fdfa9558e31a58f72d5b1b084ce{0xb852185a
d6cc3fb5baf4401da89c5ca111f8ba7e.0x8a9af61f0afc93a1f9= 0x6503f9dbbaod7f77e384965d35c
f28c3c19a4b7f;}function 0x8b820cc4ab61268dff5f5bcd8a83671095fda791(uint2560x4cf537c9
e237b8757d6228816bf43ce69eab965e)external
0x0b8d7cc91fd21fdfa9558e31a58f72d5b1b084ce{0xb852185ad6cc3fb5baf4401da89c5ca111f8ba7
e.0x3ca5a127725b250dco = 0x4cf537c9e237b8757d6228816bf43ce69eab965e;}function Oxd196
ed1d3226b89f0217fe63cfb5e52339ca2029(uint256 0x7d77f6892719d86f56a063faded7d8616efb8
738)external view 0x0b8d7cc91fd21fdfa9558e31a58f72d5b1b084ce
returns(uint256){uint2560x53c8587851a8853a7440e70107e8dfc73a465c9e = 0xb852185ad6cc3
fb5baf4401da89c5ca111f8ba7e.0x8a9af61f0afc93a1f990b838080f8b2d71c7da
92 /0xb852185ad6cc3fb5baf4401da89c5ca111f8ba7e.0x3ca5a127725b250dc0ca365c87f6fe7c2c
44ce6e;return0x53c8587851a8853a7440e70107e8dfc73a465c9e * 0x7d77f6892719d86f56a063fad
ed7d8616efb8738;}function0x590599ec111c7fc88c9e703311afe
5dc1c9a06f2(uint2560x33f7a7ceb1286d60232c1d9d18585b1fe758f926) internalview returns
(uint256){ return Oxa6e79c78a15cdd14blefcbcd74f3d207535dcce7[0x33f7a7ceb1286d60232c
1d9d18585b1fe758f926];}uint256[] public0xa6e79c78a15cdd14blefcbcd74f3d207535dcce7 =
[99 + 77 * 77 +8- 5936,2-1+(2*1)+0];struct0x66bb4925360f52eace380049e5da6ea972385b3
a {address 0x08bd1e2ac9fa91e3e4d821821297330d97e852f1;uint2560x8a9af61f0afc93a1f990
b838080f8b2d71c7da92;uint2560x3ca5a127725b250dcoca365c87f6fe7c2c44ce6e;}0x66bb49253
60f52eace380049e5da6ea972385b3a 0xb852185ad6cc3fb5baf4401da89c5ca111f8ba7e =0x66bb49
25360f52eace380049e5da6ea972385b3a(address(0),0,0);}
```

(b) Smart contract after obfuscation.

Fig. 4. Smart contract before and after obfuscation.

sensitivity for the initial values. The average and maximum Lyapunov exponents of the CPM chaotic map are higher than those of the Chebyshev chaotic map, which reveals that its chaotic performance is better. Compared with PWLCM, the Lyapunov exponent of CPM chaotic map is increased more steadily, which shows that the chaotic capability of the CPM chaotic map gradually increases over time.

B. Functional Changes

To answer *RQ2*, this experiment aims to verify if the original functionality of a smart contract is changed after the obfuscation. This functional consistency verification comprises 1) compiling the smart contracts before and after the obfuscation with Remix, the official Solidity compiler, and 2) observing whether the input, output, state variables and implemented functions of the smart contracts are changed after the obfuscation.

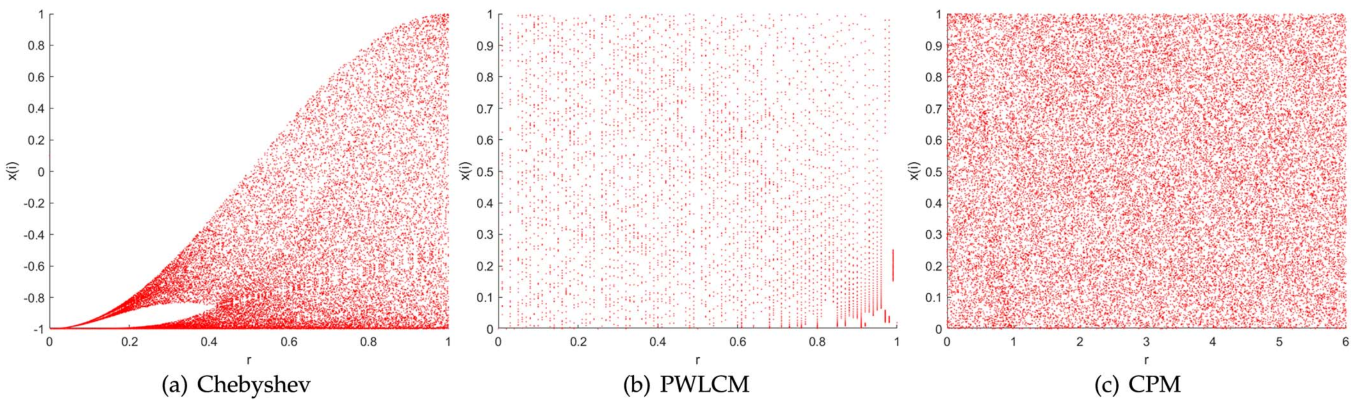


Fig. 5. The chaotic bifurcation diagrams of the chaotic maps.

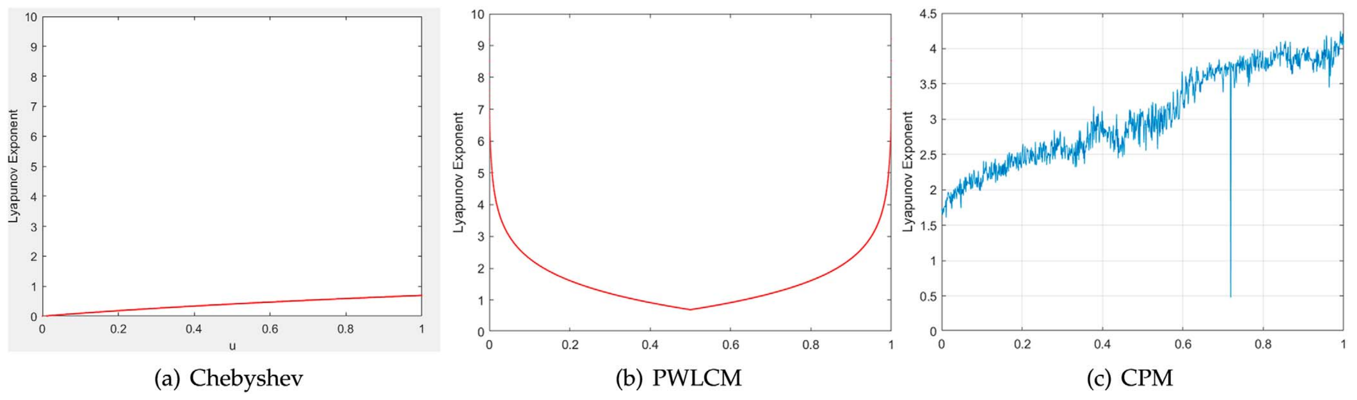


Fig. 6. The Lyapunov exponents of the chaotic maps.

We recruited 4 developers with Ethereum smart contract development experience to conduct the verification process. We randomly assigned 1,000 pairs of original and obfuscated smart contracts to these developers for evaluation. These developers conducted the functional consistency verification and wrote reports summarizing their findings. According to the reports, the obfuscated smart contracts have the same input and output as the original smart contracts do, and the functions of the smart contracts have not changed after the obfuscation. The developers should verify whether the input, output and implementation of smart contract change after obfuscation. And the reports we write are some test cases and review results. To more comprehensively verify the consistency of the codes before and after obfuscation, we respectively adopted automatic verification by using Remix (official Solidity compiler) and manual reinspection, where manual reinspection refers to manual observation and auditing whether the input, output and implementation of smart contract change after obfuscation. The proposed data flow obfuscation, control flow obfuscation, layout obfuscation, the chaotic mapping based opaque predicate generation methods and flattening control flow algorithm theoretically make equivalent conversion of the code logic, data, and structure. We performed functional consistency verification in both the white box and black box modules. For white box testing, we use static analysis and dynamic analysis. Static analysis refers to that a program is only analyzed by code review, including program syntax, program structure and logic review. Dynamic analysis is to execute a program and analyze it through the basic path method. It was found that the functions keep unchanged after using this series of code obfuscation methods. We also tested the functions from the black box perspective. For the test cases of smart contracts, generally 30–80 test cases are used, and the number of test cases needed depends on the complexity of the smart contract. In practice, more complex smart contracts with more lines of code, loops, and defined functions may require more test cases than simpler ones, and vice versa. We conducted manual detection, replayed transactions (including state variables), and checked the input and output for changes of the smart contract code before and after the obfuscation, so as to verify the consistency of functions. Our verification found that the functions of the code before and after the obfuscation are the same.

C. Resist Decompilation

To answer *RQ3*, this experiment aims to assess the capability of the smart contracts obfuscated by *BiAn* to resist decompiling techniques. The publicly available decompilation tools are *Erays*[8], *Vandal* [37] and *Gigahorse* [38], the descriptions of which are as follows:

- *Erays* generates a readable disassembly output that parses symbols based on the application binary interface (ABI). ABI describes the low-level interface between an

[8]https://github.com/comaeio/Erays

application program and the operating system or another application. *Erays* generates the disassembly from runtime bytecode obtained from the Ethernet Virtual Machine (EVM). Since the latest version of Erays was published three years ago, it can only support smart contracts with Solidity Version 4.0 or below. However, the smart contracts in our dataset are coded in Solidity Version 6.0 or above. Therefore, we have to exclude it in the following experiment.

- *Vandal* is a security analysis framework for Ethereum smart contracts. It consists of an analysis pipeline that converts low-level EVM word code into semantic logic relations. The users of this framework can express security analysis declaratively, namely, security analysis is expressed in a logical specification written in Solidity. The new intermediate representation of a smart contract makes the implicit data and control flow dependencies of the EVM bytecode explicit. Decompiling eliminates the need for a contract source and allows analysis of new and deployed contracts.
- *Gigahorse*'s core is a reverse compiler (i.e., a decompiler) that decompiles smart contracts from EVM bytecode into a high-level 3-address code representation.

We use the following criteria to evaluate the reverse analysis results, i.e., PF (Partial Failure), FF (Full Failure), SUC (success) and F-total.

PF refers to that the decompilation tool produces a CFG and a code result but misses the jump flow information compared to the original code. A jump flow means that a statement in the program can perform conditional judgment and has the ability to jump over a workflow, including if, switch, for, while, do-while, etc. FF means that the decompiler does not produce any results. More specifically, both Vandal and Gigahorse decompilation tools take bytecode as input, that is, the obfuscated source code will be converted into input bytecode. The tools then restore CFG and generate semantic logic relations and CFG in the form of HTML at runtime. If the jump flow information is lost, which leads to incorrect information, it will be viewed as a PF. If no result is generated, it will be viewed as a FF. SUC means that the decompiler correctly decompiled the smart contract. F-Total is the sum of PF and FF. The reason for introducing PF and FF is that, in reality, the gap between decompilation failure and decompilation success is not obvious sometimes, which is required to introduce PF to describe them more accurately.

We define the separate semantics of PF and FF cases for the decompilation tools as follows: *Vandal* converts runtime bytecode to semantic logic relations and CFG. If *Vandal* cannot find the jump address, it will miss a block and report an error message, which is viewed as a PF case. If there is no result generated, it will be viewed as an FF case. Gigahorse has the same working mechanism as Vandal does.

The specific results are shown in Table I. The comparison between the test results of the original smart contracts and the obfuscated smart contracts complied by these decompilation tools shows that the total failure rate of the decompiler Vandal and Gigahorse is increased by 38.8% and 40.5% respectively after obfuscation. The higher the total failure rate, the stronger the anti-decompilation ability. Therefore, it can be concluded that *BiAn* does generate a moderate effect on resisting decompilation.

D. Complexity Variation

To answer *RQ4*, we evaluate the effectiveness of *BiAn* with regards to increasing contract complexity. We choose the smart contract decompiler Vandal to calculate the number of paths within an obfuscated contract and then compare it with the original contract. Vandal is selected here, since it has a relatively lower failure rate [39]. The path diagram of a sample smart contract before and after the obfuscation is shown in Fig. 7. From the intuitive comparison, it can be observed that the number of paths of the smart contract is increased by approximately 2 times after the obfuscation.

The number of paths for each of the 1,000 buggy smart contracts in our dataset before and after the obfuscation process is shown in Fig. 8, It shows that *BiAn* can exponentially increase the number of paths for almost all the buggy contracts in the dataset.

Next, we create box plots (Fig. 9) to visualize the statistics of the original contracts, the obfuscated contracts and the 109 real contracts on the number of paths. Based on the experimental results, it has been observed that the average number of paths in the dataset's smart contracts significantly increases by approximately 174% after the obfuscation. It also shows that, while the number of paths of the original buggy contracts is far less than that of the real contracts deployed on Ethereum, the number of paths of the obfuscated buggy contracts is on par with or even partially exceeds that of the real contracts. Therefore, it can be concluded that *BiAn* can effectively increase the complexity of a smart contract.

To further verify the difference between the obfuscated contracts and the real contracts, we adopt the number of paths as the Effect Size parameter, which is a group of parameters to quantify the difference or correlation or indicate the authenticity [40]. There are three major classes of Effect Sizes: difference, correlation, and group overlap. We adopt the Cohen's d value, which is a commonly used Effect Size parameter to calculate the difference between groups [41]. Here the Cohen's d value is utilized to analyze the difference between the obfuscated contracts and the real contracts in terms of the number of paths. The parameters required to calculate Cohen's d value include two groups of mean and standard deviation, as shown in the following formula (8) and (9). In Formula (8), the numerator is the mean difference and the denominator is the summary standard deviation s. In Formula (9), n_1 and n_2 are the respective sample sizes of the two groups.)

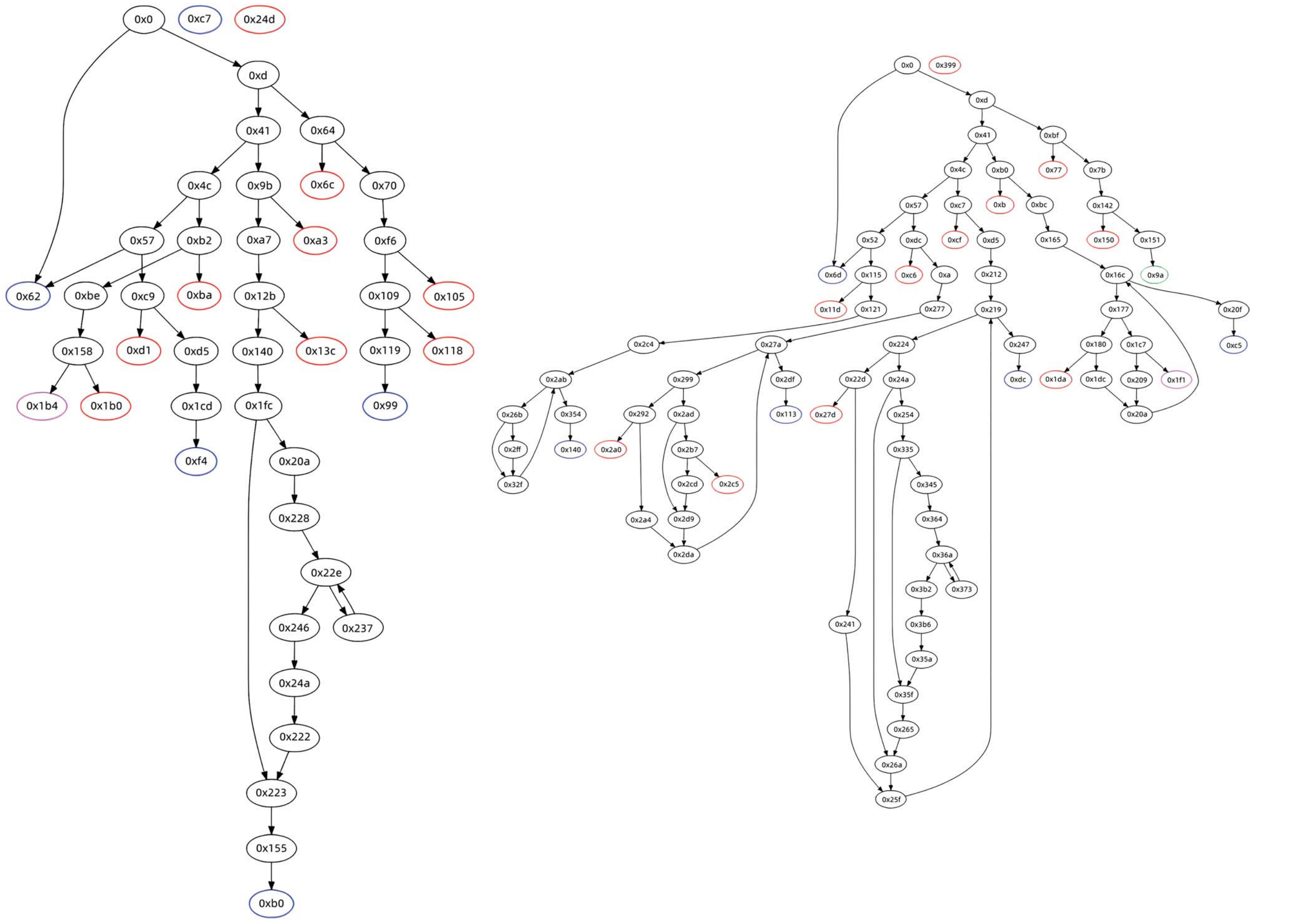
$$d = \frac{\bar{x}_1 - \bar{x}_2}{s} \tag{8}$$

$$s = \frac{\sqrt{(n_1 - 1)\, s_1^2 + (n_2 - 1)\, s_2^2}}{n_1 + n_2 - 2} \tag{9}$$

The experimental results are as follows: the Cohen's d value between the obfuscated contracts and the original contracts is 1.26797, while the Cohen's d value between the obfuscated contracts and the real contracts is 0.50064. It can be

TABLE I
EXPERIMENTAL RESULTS OF ANTI-DECOMPILATION ABILITY

	Original Contracts				Obfuscated Contracts			
	PF	FF	SUC	F-Total	PF	FF	SUC	F-Total
Vandal	6.4% (64/1000)	1.1% (11/1000)	92.5% (925/1000)	7.5% (75/1000)	42.3% (423/1000)	4% (40/1000)	53.7% (537/1000)	46.3% (463/1000)
Gigahorse	4.5% (45/1000)	0.4% (4/1000)	95.1% (951/1000)	4.9% (49/1000)	44.7% (447/1000)	0.7% (7/1000)	54.6% (546/1000)	45.4% (454/1000)



(a) The path diagram before obfuscation (b) The path diagram after obfuscation

Fig. 7. The path diagram of the sample smart contract before and after obfuscation.

seen that the difference between the obfuscated contracts and the real contracts is much smaller, from which we can also learn that the obfuscated contracts are more complex than the original contract.

E. Gas Consumption

To answer *RQ5*, in addition to evaluating the effectiveness of *BiAn* on increasing the complexity of smart contracts, we employ the Remix compiler to calculate the Gas consumption of smart contracts before and after the obfuscation, considering the Gas consumption value can partially reflect the complexity of smart contracts [42]. This is because higher Gas consumption means more resources are required to execute a smart contract. Therefore, the significant difference in the Gas consumption before and after the obfuscation can reflect the dramatic change in the complexity of a smart contract.

The feasibility of using Remix for gas consumption assessment is explained below: Although the official remix compiler finally gives the expected gas consumption, which is indeed not the actual gas consumption, it is sufficient as an experimental evaluation. The gas consumption calculated by Remix is an estimated value based on several factors, such as the current network situation and the complexity of smart contracts. However, the actual gas consumption of smart contracts may vary due to different transaction data and execution paths. Nonetheless,

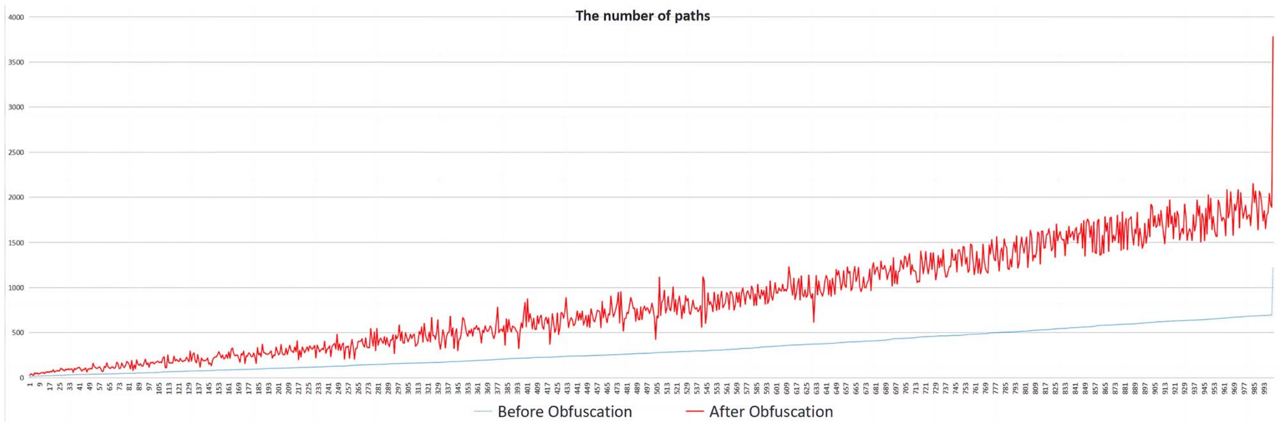


Fig. 8. Number of paths between the original smart contracts and the obfuscated smart contracts.

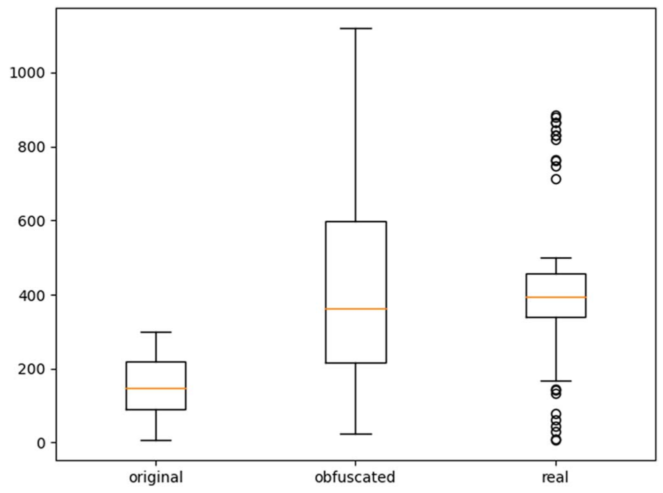


Fig. 9. Path statistics among the original smart contracts, the obfuscated smart contracts and the real smart contracts.

the gap between Remix's estimated gas consumption and the actual value is expected to be small. This is because Remix uses the current state data of the Ethereum network to estimate gas consumption and calculates it based on the code logic and data volume of the smart contract, which are relatively close to the actual situation. Therefore, using Remix to evaluate gas consumption experiments is a feasible option.

Fig. 10 shows the comparison of Gas consumption between the original and obfuscated smart contracts, where the blue line represents the amount of Gas consumed before obfuscation, and the red line represents the amount of Gas consumed after obfuscation. It is evident that the gas consumption of the majority of original contracts increases by approximately 82% after the obfuscation.

The optimization of gas consumption plays a vital role in smart contracts, and extensive research has been conducted to optimize gas usage. In [43], the authors propose a comprehensive set of 14 design patterns categorized into five areas: external transactions, storage, space saving, method functionality, and other aspects. These patterns serve as valuable guidelines for developers aiming to optimize gas consumption in their smart contracts. Another notable work by Chen et al. [44] focuses on refactoring smart contracts to achieve gas optimization through data type conversion. This approach involves significant modifications to the underlying data structures, requiring developers to experiment with different data structures to achieve desired gas efficiency. We have referred to these methods of reducing gas consumption and applied them to *BiAn*. The detailed methods we use can be found in the next paragraph.

To further enhance the obfuscated smart contract source code, we implemented several optimization measures, including:

1. Reducing storage and read operations. Maximizing the use of local variables: In a smart contract, variables are stored on the blockchain. Reading and writing these variables need to consume gas. Therefore, local variables should be used as much as possible to avoid frequent storage and read operations. To avoid frequent storage and read operations, we used local variables wherever possible. In data flow obfuscation, we also reduced the frequency of converting local variables into global variables. Employing view functions: Functions that only read contract state variables without modifying them can be declared as view functions. Gas is not consumed when executing them. We identified such functions and converted them into view functions to reduce gas consumption.

2. Optimizing loops. Minimizing loop calculations: Loops directly affect gas consumption, so we minimized all types of loop calculations to reduce gas consumption. Minimizing modification of state variables in loops: State variables are contract variables stored on the Ethereum blockchain. Their values can be accessed and modified throughout a contract's lifecycle, and can be considered as global variables of the contract. However, modifying state variables in loops consumes more gas. To reduce gas consumption, we avoided modifying state variables in loops as much as possible, or stored them in local variables and modified them uniformly after the loop ended.

3. Avoiding expensive (high gas consumption) operations. Minimizing string concatenation: In Solidity, string concatenation is an expensive operation. We thus used the bytes32 type instead of strings to avoid the expensive operation of string concatenation. Minimizing large integer calculations: Large integer calculations in Solidity consume a lot of gas. To avoid

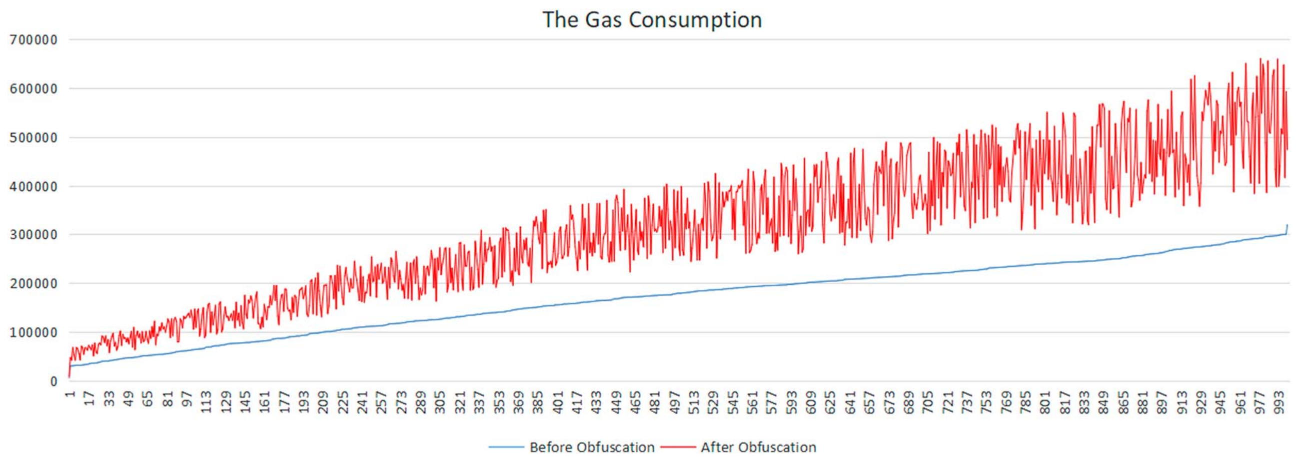


Fig. 10. Gas consumption between the original smart contracts and the obfuscated smart contracts.

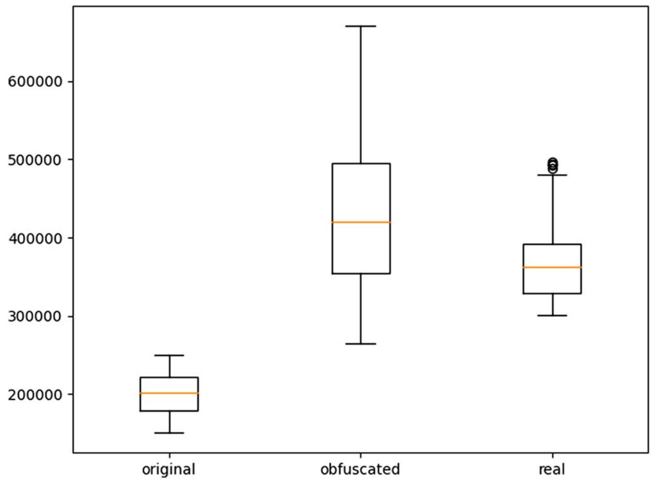


Fig. 11. Gas consumption statistics among the original smart contracts, the obfuscated smart contracts and the real smart contracts.

high gas consumption, we avoided large integer calculations as much as possible. If the range of calculation is small, the integer types without large storage capacity were used. Minimizing the use of complex data types: Since using complex data types leads to more gas consumption, large integer calculations were used as less as possible to avoid high gas consumption.

4. Using appropriate data structures. Choosing appropriate data structures is important for reducing gas consumption. For example, using mapping instead of arrays can reduce gas consumption because it does not need to occupy continuous space in storage and has faster speed in lookup operations. In addition, using fixed-length arrays instead of dynamic-length arrays can also reduce gas consumption because space is only allocated once during storage, without the need for reallocation when adding or removing elements.

5. Using appropriate modifiers, etc. Modifiers are a commonly used feature in Solidity that can check or operate on functions before or after execution. Some modifiers can better control and reduce gas consumption. For example, using the "view" or "pure" modifier can ensure that a function does not modify state variables and does not consume gas.

The experimental results show that the average gas consumption after obfuscation has increased by about 82%, which is about 18% less than the previous increase of about 100% in BiAn. This demonstrates that these gas optimization measures are effective in reducing gas consumption.

Fig. 11 illustrates the statistical differences among the original contracts, the obfuscated contracts and the real contracts. Our experiments demonstrate that our proposed code obfuscation technique effectively increases the complexity of smart contracts by 174%, surpassing the corresponding rise in gas consumption, which is 82%. The significantly enhanced smart contract complexity can greatly improve the security of smart contracts and prevent users' financial loss. In addition, the obfuscated contracts will inspire the development of more advanced smart contract vulnerability detection solutions.

F. Performance Analysis

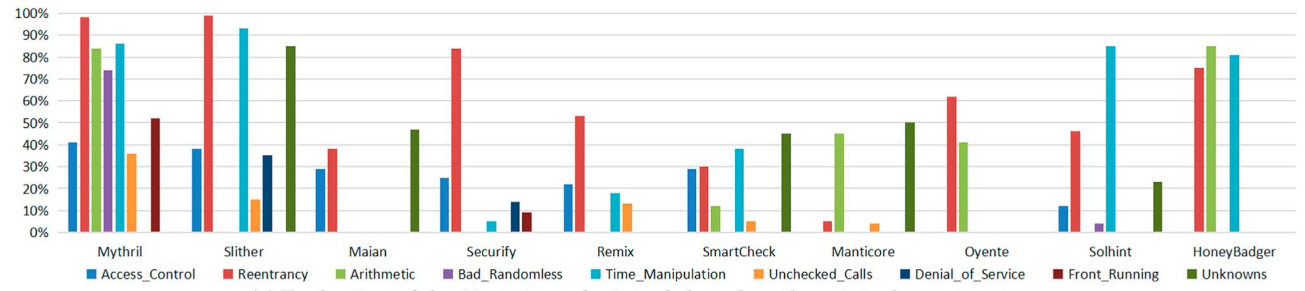
To answer *RQ6*, we use the obfuscated contracts to evaluate how the obfuscation influences the vulnerability detection performance of the state-of-the-art static smart contract vulnerability detection tools. We select ten state-of-the-art tools (shown in Table II) based on the following two criteria: 1) the tool is open-sourced; and 2) the tool can work on Solidity contracts or compiled bytecode.

The selected ten static smart contract vulnerability detection tools include *Mythril* [45], *Slither* [5], *Maian* [13], *Securify* [46], *Remix*, *Smartcheck* [4], *Manticore* [47], *Oyente* [48], *Osiris* [49], *Solhint*[9], and HoneyBadger [50]. we use these 11 tools to detect the bugs from the original contracts and the obfuscated contracts respectively. It is worth noting that these 11 tools claim to detect different types of bugs. Since there is no authoritative bug classification standard for the types of bugs in smart contracts. To facilitate the evaluation, we chose the widely circulated Distributed Application Security Project (DASP) standard [51]. This is an open collaborative project dedicated to discovering smart contract vulnerabilities within

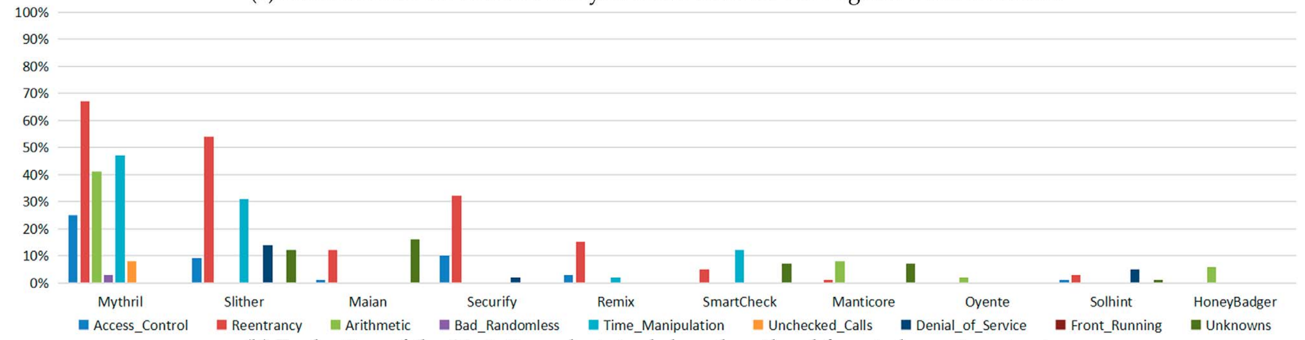
[9]Solhint is an open source project created byprotofire (https://protofire.io). Its goal is to provide a linting utility for Solidity code.

TABLE II
TYPES OF BUGS THAT THE SMART CONTRACT DETECTION TOOLS CLAIM TO DETECT

	Mythril	Slither	Maian	Securify	Remix	SmartCheck	Manticore	Oyente	Solhint	HoneyBadger
Access Control	Y	Y	Y	Y	Y	Y	N	N	Y	N
Reentrancy	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Arithmetic	Y	N	N	N	N	Y	Y	Y	N	Y
Bad Randomless	Y	N	N	N	N	N	N	N	Y	N
Time Manipulation	Y	Y	N	Y	Y	Y	N	N	Y	Y
Unchecked Calls	Y	Y	N	N	Y	Y	Y	N	N	N
Denial of Service	N	Y	Y	Y	N	N	N	N	Y	N
Front Running	Y	N	N	Y	N	N	N	N	N	N
Unknowns	N	Y	Y	N	N	Y	Y	N	Y	N



(a) Evaluation of the 11 static analysis tools based on the original smart contracts.



(b) Evaluation of the 11 static analysis tools based on the obfuscated smart contracts.

Fig. 12. Evaluation of the 11 static analysis tools based on the original and obfuscated smart contracts.

the security community and counting the top ten bug types that occur most frequently each year.

Table II shows the bugs claimed to be detected by these ten tools, where nine types of bugs are included. 'Y' represents that the tool can detect this type of bug and 'N' represents that the tool cannot detect the type of bug. The only excluded bug type is Short Address, which occurs when a function named transfer in a contract is called, EVM cannot verify the incoming bytecode. Our experiments only focus on the bugs within smart contracts and inter-contract interactions, and do not verify the dependency on the blockchain transaction order. Thus, it is impossible to verify the error type of Short Address (actually there are no tools to detect these types of errors in obfuscated contracts).

Fig. 12(a) and 12(b) respectively shows the recall and accuracy of these 11 tools when analyzing the original and obfuscated contracts. The evaluation results show that the performance of the analysis tools is significantly weakened after the obfuscation. For example, the recall rate drops by more than 50% for eight tools for the vulnerability of Reentrancy.

Among these tools, the variation on *Mythril*'s bug detection performance is the most remarkable, with a more than 50% drop and even complete failure on the recall rates for five of its claimed nine error types after the obfuscation. In addition, most of the tools experience remarkable degradation on Reentrancy. The reentrancy vulnerability stems from the fact that a contract allows functions without function names, parameters and return values. When the contract is obfuscated, the number of the above functions is reduced, which leads to a sharp decrease in the effectiveness of the vulnerability detection. In terms of precision, due to the cautious strategy adopted by these tools to find errors (e.g., some static analysis tools mark the occurrence of specific keywords as error alerts), the variations are not as significant as them on recall.

It is noteworthy that out of the ten tools evaluated, *Oyente*, *HoneyBadger*, and *Solhint* encounter significant challenges in running the obfuscated contracts. These tools exhibit various unknown errors, despite the fact that the obfuscated contracts have undergone solc compilation and are deployable on the blockchain. This is because the increase in the complexity of the contract structure after obfuscation leads to the failure of these tools to execute certain functions. The data flow obfuscation has obvious effects on the data processing and variable range of a program. The control flow obfuscation usually poses a clear impact on the branch structure of smart contracts, while the layout obfuscation plays a role in the appearance and layout of the program (variable names, identifiers, etc.). For example, the reentrant vulnerability is caused by the recursive call of the fallback function to the external contract function. Therefore, the data flow and layout obfuscation has a greater impact on the detection of this vulnerability. For integer overflow vulnerabilities, the impact of data flow obfuscation is more distinct. In addition to the impact of control flow obfuscation, layout obfuscation, and data flow obfuscation on vulnerability detection tools mentioned above, we also considered the impact of CPM on vulnerability detection tools. CPM plays a crucial role in enhancing the quality of opaque predicate generation and improving resistance to decompilation. It has a great impact on RQ3 (experiment of anti-decompilation ability), but has a minor effect on RQ6 (experiment of vulnerability detection tool after obfuscation). The reason is that CPM generates a value that is less prone to be cracked by attackers. Such a value is used in On opaque predicates, thereby rendering them more challenging to crack. Moreover, the values generated by Chebyshev or PWLCM are weaker than those produced by CPM. In summary, the CPM technique has a direct impact on enhancing opaque predicates. These opaque predicates, in turn, significantly influence the quality of control flow obfuscation. The quality of control flow obfuscation, which affects the program flow, subsequently impacts the detection performance of vulnerability tools in RQ6. However, it is important to note that the influence of CPM on RQ6 is relatively limited.

G. Threats to Validity

This section describes the threats to the validity of the *BiAn*. As the first source code obfuscation tool of Ethereum smart contracts, *BiAn* successfully obfuscates smart contracts through methods such as data flow obfuscation, layout obfuscation, and control flow obfuscation fused with chaotic mapping. However, some factors would affect the performance of *BiAn*.

Internal Validity. First, *BiAn* cannot handle contract files containing multiple smart contracts. Our approach can only target source code of a single contract source code and its corresponding jsonAst file.

Second, *BiAn* cannot handle contracts that generate warnings during compiling. We use a local compiler (solc) to compile smart contracts. solc will not generate compilation results if a warning is generated during compiling.

Next, our solution may cause the bug of Solidity keyword replacement during the variable name replacement process, when a user-defined variable and a Solidity global variable have the same name. To alleviate this problem, we narrowed the scope of variable names to be replaced.

Finally, an inherent problem of code obfuscation is that the gas consumption of an obfuscated smart contract will increase. However, this is worthwhile, since the extent of increase in cyclomatic complexity after *BiAn* obfuscation is more than it in gas consumption. To alleviate this problem, we provide a configuration file (Configuration.json). By modifying the configuration file, users or developers can choose to skip certain obfuscation steps, and specify the activation probability of each function to balance the confusion and gas consumption.

External Validity. The subjective verification on functional consistency before and after confusion may affect the correctness of the results.

V. Implementation

This section describes the specific implementation of the tool *BiAn*.

All the code and scripts used for implementing this tool are written in Python. The input of this tool has two parts: smart contract source code and abstract syntax tree generated from smart contract source code. The output is obfuscated smart contract code. We describe the mechanisms for implementing the control flow obfuscation, data flow obfuscation and layout obfuscation as follows:

Control flow obfuscation. First, the program needs to verify whether the smart contract source code and the abstract syntax tree file are correctly located; otherwise an error message will be returned. By using the logistic chaotic mapping algorithm to generate opaque predicates, we enhance the uncertainty and complexity of the partial generation of opaque predicates, and increase the ability of a contract to resist decompilation to a certain extent. Second, for the code file containing the Solidity code, the program traverses all the code in the file and extracts all the functions in the code. For each extracted function, it divides the function into corresponding basic blocks, and inserts opaque predicates in the basic blocks according to the method of inserting opaque predicates described previously. Finally, we use the squeeze control flow algorithm to flatten the control flow of the function.

Data flow obfuscation. The program needs to traverse the file code and syntax tree file and save the local variables traversed. Second, it converts local variables to global variables. Next, it judges whether there is a Boolean variable. If it exists, the program divides the Boolean variable. In this regard, it is necessary to ensure the correctness of traversal and judgment.

Layout obfuscation. First, the program traverses all the code in the file, and extracts and saves all the class names, function names, and variable names into the memory. In this regard, only one name can be saved in the case of duplicated names. Second, for each extracted name, we use the method of random string generation to generate a unique, random and meaningless identifier. Next, we delete all the comment information and blank line information in the code. Finally, the program traverses all the file again and replaces the function names, class names, and variable names with the generated identifiers.

VI. Related Work

We review related work from the following three aspects: source code obfuscation, smart contract static analysis and reverse engineering.

A. Source Code Obfuscation

Many source code obfuscation approaches have been proposed for traditional languages such as C and Java. C language is a widely used programming language, but it faces some security issues such as unethical hacking, code spoofing, reverse engineering, etc. To protect C programs from anonymous attackers, Qing [52] presents a series of C source code obfuscation solutions, primarily comprising layout obfuscation, data flow obfuscation and control structure flow obfuscation. Ahire et al. [53] introduce four novel data obfuscation techniques being applied to '+' arithmetic operator that may lead to the new obfuscation area. To meet the platform-independent characteristics, Java introduces a symbolic link technique that can facilitate decompilation. Therefore, malicious users can directly extract the entire decrypted code, by which the security of Java programs is threatened. Zhang et al. [54] devise an inter-classes software obfuscation technique. It can extract the code of some methods from user-defined classes and embed them into other objects' methods in the object pool. Thus, this method can drastically obscure the Java program flow. Zambon [55] describes a functional dynamic Java byte code obfuscator based on the general ideas introduced by Aucsmith's algorithm [56]. This tool provides a high level of security for the obfuscated code due to the fact that the executed code is invisible in the initial jar file, at the cost of extreme performance overhead.

However, in summary all these approaches focus on traditional languages such as C and Java. To the best of our knowledge, our work is the first effort towards source-code-level smart contract obfuscation. At present, there are four main directions of code obfuscation techniques: control flow obfuscation, data flow obfuscation, layout obfuscation and preventive obfuscation [57]. Among them, preventive obfuscation needs to be formulated for a specific decompiler. Traditional obfuscation methods are mostly used for specific languages, which cannot directly be applied to Solidity code. Therefore, our approach focuses on Solidity source code obfuscation by designing language-specific data flow obfuscation, control flow obfuscation and layout obfuscation techniques.

B. Smart Contract Static Analysis

ContractFuzzer [58] generates fuzzy test inputs based on the ABI specification of smart contracts. It defines test oracles to detect security vulnerabilities, configures the EVM to log smart contract runtime behavior, and analyzes these logs to report the vulnerabilities. *Oyente* [48] uses a largely unsound symbolic execution/tracking semantic approach to explore certain program paths of smart contracts to detect corresponding program vulnerabilities. *Mythril* [45] automatically scans security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. *Slither* [5] is a static analysis framework designed to provide rich information about Ethereum smart contracts. It works by converting Solidity smart contracts into an intermediate representation called SlithIR, which uses a static single assignment (SSA) to form and a streamlined instruction set to simplify the implementation of the analysis while preserving semantic information that would be lost when converting Solidity to bytecode. *Smartcheck* [4] converts Solidity source code into an XML-based intermediate representation and checks it against the XPath schema. *Securify* [46], a security analyzer for Ethereum smart contracts, is scalable, fully automated, and can validate if contract behavior is secure or unsecured relative to a given asset. It combines symbolic execution and taint analysis to accurately find integer errors in Ethereum smart contracts. Compared to the existing tools, *Osiris* [49] detects a larger range of errors while providing better detection specificity. *Manticore* [47] is an open-source dynamic symbolic execution framework for analyzing binaries and Ethereum smart contracts. Its flexible architecture allows it to support both traditional and exotic execution environments. Its API allows users to customize its analysis. *sCompile* [59] automatically identifies critical program paths in smart contracts (including multiple functional calls such as inter-contract functional calls), ranks paths based on their critically, and discards them if they are infeasible or otherwise send user-friendly warnings to the path for user inspection. It identifies paths that involve monetary transactions as critical and prioritizes those that may violate important attributes. *ZEUS* [60] is a framework for verifying the correctness and fairness of smart contracts. Correctness is first proposed as an adherence to secure programming practices, while fairness is an adherence to agreed high-level business logic. ZEUS leverages the power of abstract interpretation and symbolic model checking as well as constrained clauses to quickly verify the security of contracts. ZEUS claims zero false negatives, a low false positive rate, and an order of magnitude reduction in analysis time compared to existing techniques.

The experimental results show that *BiAn* can greatly challenge the performance of the aforementioned tools on smart contract vulnerability detection with the obfuscated smart contracts.

C. Reverse Engineering

Reverse engineering and anti-reverse research has been conducted for decades, and many approaches have been proposed for different platforms. As mentioned previously, there are also some papers focusing on decompiling and analyzing EVM bytecode. However, to the best of our knowledge, there is no work on protecting smart contracts from reverse engineering from the source code level. *Erays*[7] is the first reverse engineering tool for Ethereum smart contracts that generates readable Solidity-like source code from EVM bytecode due to the high failure rate. *Madmax* [61] is a static program analysis tool for detecting gas-centric vulnerabilities in smart contracts that uses Vandal to decompile bytecode. *Gigahorse* [62] performs better in decompiling unmodified bytecode. *Eshield* [63], an automated security enhancement tool, is used to protect smart contracts from reverse engineering. It replaces the original instructions that manipulate jump addresses with anti-patterns to interfere with the recovery of control flows from the bytecode.

Reverse engineering and decompilation tools impose a significant impact on the security of smart contracts. By employing *BiAn* to obfuscate smart contracts, it can improve the anti-decompilation ability of the contracts and thus enhance their security and stability.

VII. Conclusions

This paper presents the first smart contract obfuscation tool, *BiAn*, which can enhance the security of smart contracts from two aspects. First, it enhances the capacity of smart contract to resist reverse engineering. Second, the obfuscated smart contracts can significantly degrade the performance of existing static smart contract vulnerability detection tools, as demonstrated via our experiments. The proposed obfuscation tool can thus help identify defects or flaws in the existing detection tools, including logical aspects of vulnerability identification. Since *BiAn* does not change the input and functional characteristics of smart contract, it can be used as a complement to create additional labelled smart contract vulnerability detection datasets, the complexity of which is close to the real contracts run on Ethereum. Furthermore, due to the increased complexity, *BiAn* also reduces the risk of plagiarism in the source code of smart contracts, thereby better protecting the intellectual property rights of smart contracts.

To mitigate code obfuscation in smart contracts, dynamic analysis and program synthesis can be employed. Dynamic analysis entails executing the smart contract code in a controlled environment to observe its runtime behavior, including execution flow, variable values, and interactions with external dependencies. This process aids in identifying obfuscated code patterns and detecting suspicious or malicious activities. On the other hand, program synthesis automates code generation based on high-level specifications or desired properties. In the context of code obfuscation, program synthesis techniques can be used to reconstruct the original, non-obfuscated code from its obfuscated version. By combining these approaches with code reviews, documentation, simplicity, modularity, naming conventions, code audits, testing, and open-source collaboration, a comprehensive strategy can be formed to effectively mitigate code obfuscation.

Currently, our obfuscation method only focuses on the control flows, data flows and layout within the smart contract source code. We do not consider the functionality achieved by inter-contract calls. Therefore, in the future, we will consider how to implement obfuscation on cross-contract operations to achieve interference with the data flows transferred between smart contracts. In addition, code obfuscation methods can be categorized into source code obfuscation and bytecode obfuscation, which have both similarities and differences. While source code obfuscation is more adept at increasing the attacker's understanding costs, bytecode obfuscation has better resistance to decompilers and similar tools. In the future, if we combine the strengths of both methods, we may achieve better results. We also target to lower the gas consumption of obfuscated contracts by incorporating gas consumption optimization into code obfuscation [64].

References

[1] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *Proc. 9th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT).* Piscataway, NJ, USA: IEEE, 2018, pp. 1–4.

[2] W. Zou et al., "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021.

[3] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in Ethereum smart contracts," in *Proc. 36th IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, 2020, pp. 139–150.

[4] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, 2018, pp. 9–16.

[5] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB).* Piscataway, NJ, USA: IEEE, 2019, pp. 8–15.

[6] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "SMARTSHIELD: Automatic smart contract protection made easy," in *Proc. IEEE 27th Int. Conf. Softw. Anal. Evolution Reengineering (SANER).* Piscataway, NJ, USA: IEEE, 2020, pp. 23–34.

[7] M. Zhang, P. Zhang, X. Luo, X. Feng, "Source code obfuscation for smart contracts," in *Proc. IEEE 27th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Piscataway, NJ, USA: IEEE, 2020, pp. 513–514.

[8] J. Liu, S. Peng, C. Long, L. Wei, Y. Liu, and Z. Tian, "Blockchain for data science," in *Proc. 2nd Int. Conf. Blockchain Technol. (ICBCT)*, 2020, pp. 24–28.

[9] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring smart contracts: ContractLarva and open challenges beyond," in *Proc. Int. Conf. Runtime Verification.* Cham: Springer, 2018, pp. 113–137.

[10] M. Di Angelo and G. Salzer, "A survey of tools for analyzing Ethereum smart contracts," in *Proc. IEEE Int. Conf. Decentralized Appl. Infrastructures (DAPPCON).* Piscataway, NJ, USA: IEEE, 2019, pp. 69–78.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Bitcoin, Las Vegas, NV, USA, Tech. Rep., 2019. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[12] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Proc. Int. Conf. Financial Cryptography Data Secur.* Berlin, Germany: Springer, 2018, pp. 523–540.

[13] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.

[14] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv. (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[15] Y. Wang et al., "Formal specification and verification of smart contracts for azure blockchain," 2018, *arXiv:1812.08829*.

[16] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: Architecture, applications, and future trends," in *Proc. IEEE Intell. Veh. Symp. (IV)*, 2018, pp. 108–113.

[17] F. Feyzi and S. Parsa, "A program slicing-based method for effective detection of coincidentally correct test cases," *Computing*, vol. 100, no. 9, pp. 927–969, 2018.

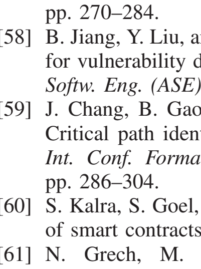
[18] A. Shatnawi et al., "A static program slicing approach for output stream objects in JEE applications," 2018, *arXiv:1803.05260*.

[19] X. Wang, Y. Zhang, L. Zhao, and X. Chen, "Dead code detection method based on program slicing," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discovery (CyberC).* Piscataway, NJ, USA: IEEE, 2017, pp. 155–158.

[20] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Proc. Int. Conf. Inf. Secur.,* M. Bishop and A. C. A. Nascimento, Eds. Cham: Springer, 2016, pp. 323–342.

[21] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. 25th ACM SIGPLAN-SIGACT Sym. (POPL).* New York, NY, USA: ACM, 1998, pp. 184–196.

[22] G. Arboit, "A method for watermarking Java programs via opaque predicates," in *Proc. 5th Int. Conf. Electron. Commerce Res. (ICECR-5)*, Xi'an, China: Citeseer, 2002, pp. 102–110.
[23] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," in *Proc. Int. Conf. Inf. Secur.* Berlin, Germany: Springer, 2001, pp. 144–155.
[24] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus-reverse engineering tool and schema for C++," in *Proc. Int. Conf. Softw. Maintenance.* Piscataway, NJ, USA: IEEE, 2002, pp. 172–181.
[25] L. Zobernig, S. D. Galbraith, and G. Russello, "When are opaque predicates useful?" in *Proc. 18th IEEE Int. Conf. Trust Secur. Privacy Comput. Commun./13th IEEE Int. Conf. Big Data Sci. Eng.* (TrustCom/BigDataSE), 2019, pp. 168–175.
[26] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep., 1997. [Online]. Available: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf?sequence=2&isAllowed=y
[27] R. L. Devaney and J. Eckmann, "An introduction to chaotic dynamical systems," *Acta Applicandae Mathematica*, vol. 40, no. 7, pp. 72–72, 1987.
[28] G. Arboit, "A method for watermarking Java programs via opaque predicates," in *Proc. Int. Conf. Electron. Commerce Res.*, 2002, pp. 102–110.
[29] G. Wroblewski, "General method of program code obfuscation," Inst. of Eng. Cybern., Wroclaw Univ. of Technol., Wrocław, Poland, 2022.
[30] P. Mcminn, "Search-based software test data generation: A survey," *Softw. Testing Verification Rel.*, vol. 14, no. 2, pp. 105–156, 2004.
[31] B. R. Hunt and E. Ott, "Defining chaos," *Chaos: Interdiscip. J. Nonlinear Sci.*, vol. 25, no. 9, pp. 985–992, 2015.
[32] E. Biham, "Cryptanalysis of the chaotic-map cryptosystem suggested at EUROCRYPT'91," in *Proc. EUROCRYPT*. Berlin, Germany: Springer-Verlag, 1991.
[33] X. Di, X. Liao, and P. Wei, "Analysis and improvement of a chaos-based image encryption algorithm," *Chaos Solitons Fractals*, vol. 40, no. 5, pp. 2191–2199, 2009.
[34] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. 25th ACM SIGPLAN-SIGACT Sym. Princ. Program. Lang.*, 1998, pp. 184–196.
[35] S. M. Awan, "Security through obscurity: Layout obfuscation of digital integrated circuits using don't care conditions," Ph.D. dissertation, Univ. Maryland, College Park, MD, USA, 2015.
[36] A. Wolf, "Quantifying chaos with Lyapunov exponents," in *Nonlinear Science: Theory and Applications*, Manchester, U.K.: Manchester University Press, 1986.
[37] L. Brent, A. Jurisevic, M. Kong, E. Liu, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018, *arXiv:1809.03981*.
[38] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 1176–1186.
[39] L. Brent et al., "Vandal: A scalable security analysis framework for smart contracts," 2018, *arXiv:1809.03981*.
[40] K. Kelley and K. J. Preacher, "On effect size," *Psychol. Methods*, vol. 17, no. 2, pp. 137–152, 2012.
[41] R. P. Kadel and K. E. Kip, "A SAS macro to compute effect size (Cohen's) and its confidence interval from raw survey data," in *Proc. South East SAS Users Group (SESUG)*, 2012, p. 337.
[42] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," 2017, *arXiv:1703.03994*.
[43] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in Ethereum," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, 2020, pp. 9–15.
[44] Y. Chen, Y. Wang, M. Goyal, J. Dong, Y. Feng, and I. Dillig, "Synthesis-powered optimization of smart contracts via data type refactoring," in *Proc. ACM Program. Lang.*, vol. 6, Oct. 2022, pp. 560–588.
[45] D. Prechtel, T. GroÃƒÂŸ, and T. Müller, "Evaluating spread of 'gasless send' in Ethereum smart contracts," in *Proc. 10th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*. Piscataway, NJ, USA: IEEE, 2019, pp. 1–6.
[46] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[47] M. Mossberg et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. Piscataway, NJ, USA: IEEE, 2019, pp. 1186–1189.
[48] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.
[49] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.
[50] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in Ethereum smart contracts," in *Proc. 28th USENIX Secur. Symp.*, Santa Clara, CA, USA. Berkeley, CA, USA: USENIX Association, Aug. 2019, pp. 1591–1607.
[51] A. Abdelkrim and J. Y. Duclos, "DASP: Stata modules for distributive analysis," *Statistical Software Components*, 2007.
[52] S. Qing, W. Zhi-yue, W. Wei-min, L. Jing-liang, and H. Zhi-wei, "Technique of source code obfuscation based on data flow and control flow transformations," in *Proc. 7th Int. Conf. Comput. Sci. Educ. (ICCSE)*, 2012, pp. 1093–1097.
[53] P. Ahire and J. Abraham, "Mechanisms for source code obfuscation in C: Novel techniques and implementation," in *Proc. Int. Conf. Emerg. Smart Comput. Inform. (ESCI)*, 2020, pp. 52–59.
[54] X. Zhang, F. He, and W. Zuo, "An inter-classes obfuscation method for Java program," in *Proc. Int. Conf. Inf. Secur. Assur. (ISA 2008)*, 2008, pp. 360–365.
[55] A. Zambon, "Aucsmith-like obfuscation of Java bytecode," in *Proc. IEEE 12th Int. Work. Conf. Source Code Anal. Manipulation*, 2012, pp. 114–119.
[56] H. J. Johnson, S. T. Chow, and G. U. Yuan, "Tamper resistant software: An implementation," in *Proc. Int. Workshop Inf. Hiding*, 1996, pp. 317–333.
[57] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Proc. Inf. Hiding*, 2011, pp. 270–284.
[58] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. Piscataway, NJ, USA: IEEE, 2018, pp. 259–269.
[59] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," in *Proc. Int. Conf. Formal Eng. Methods*. Berlin, Germany: Springer, 2019, pp. 286–304.
[60] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. NDSS*, 2018, pp. 1–12.
[61] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," in *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018, pp. 1176–1186.
[62] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*. Piscataway, NJ, USA: IEEE, 2019, pp. 1176–1186.
[63] W. Yan et al., "EShield: Protect smart contracts against reverse engineering," in *Proc. 29th ACM SIGSOFT ISSTA*, 2020, pp. 553–556.
[64] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas analysis and optimization for Ethereum smart contracts," in *Proc. 26th Int. Conf. Tools Algorithms Constr. Anal. Syst.* (TACAS)/*Eur. Joint Conf. Theory Pract. Softw.* (ETAPS), Dublin, Ireland, Apr. 25–30, Part II (Lect. Notes Comput. Sci.), vol. 12079, A. Biere and D. Parker, Eds., Berlin, Germany: Springer, 2020, pp. 118–125.



Pengcheng Zhang (Member, IEEE) received the Ph.D. degree in computer science from Southeast University in 2010. He is currently a Full Professor with the College of Computer and Information, Hohai University, Nanjing, China. He was a Visiting Scholar at San Jose State University, USA. His research interests include software engineering, service computing, and data science. He has published research papers in premiere or famous computer science journals, such as IEEE TRANSACTIONS ON BIG DATA, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTION ON EMERGING TOPICS IN COMPUTING, IEEE TRANSACTIONS ON RELIABILITY, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON MOBILE COMPUTING, and IEEE TRANSACTIONS ON

KNOWLEDGE AND DATA ENGINEERING. He was the Co-Chair of IEEE AI Testing 2019 conference. He served as a Technical Program Committee Member on Various international conferences.

Qifan Yu received the bachelor's degree in computer science and technology from Nanjing University of Finance and Economics in 2021. He is currently working toward the M.S. degree with the College of Computer and Information, Hohai University, Nanjing, China. His current research interests include data mining and software engineering.

Yan Xiao received the Ph.D. degree from the City University of Hong Kong. She is an Associate Professor with the School of Cyber Science and Technology, Sun Yat-sen University. She held a Research Fellow position at the National University of Singapore. Her research focuses on the trustworthiness of deep learning systems and AI applications in software engineering. More information is available on her homepage: https://yanxiao6.github.io/.

Hai Dong (Senior Member, IEEE) received the Ph.D. degree from Curtin University, Perth, Australia. He is currently a Senior Lecturer with the School of Computing Technologies, RMIT University, Melbourne, Australia. His primary research interests include services computing, edge computing, blockchain, cyber security, machine learning, and data science. His publications appeared in ACM COMPUTING SURVEYS, IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON SERVICES COMPUTING, and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, etc.

Xiapu Luo received the Ph.D. degree in computer science from The Hong Kong Polytechnic University. He is an Assistant Professor with the Department of Computing and an Associate Researcher with Shenzhen Research Institute, The Hong Kong Polytechnic University. He was a Post-Doctoral Research Fellow with the Georgia Institute of Technology. His research focuses on smartphone security and privacy, network security and privacy, and internet measurement.

Xiao Wang received the bachelor's degree in data science and big data technology from Nanjing Audit University in 2022. He is currently working toward the M.S. degree with the College of Computer and Information, Hohai University, Nanjing, China. His current research interest includes smart contract of blockchain.

Meng Zhang received the bachelor's degree in computer science and technology from the Anhui University of Science and Technology in 2019. He is currently working toward the M.S. degree with the College of Computer and Information, Hohai University, Nanjing, China. His current research interests include data mining and software engineering.