Hindawi
Security and Communication Networks
Volume 2022, Article ID 6903370, 15 pages
https://doi.org/10.1155/2022/6903370

*Research Article*

# COOPS: A Code Obfuscation Method Based on Obscuring Program Semantics

**Yang Li, Fei Kang, Hui Shu, Xiaobing Xiong, Zihan Sha, and Zhonghang Sui**

*State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China*

Correspondence should be addressed to Fei Kang; kfminnie@163.com

Received 27 March 2022; Accepted 10 August 2022; Published 25 September 2022

Academic Editor: Mohamed Nassar



As reverse engineering technology develops rapidly, the financial loss caused by software security issues is urgent. Therefore, how to effectively protect software is a critical problem to solve. The software protection method based on code obfuscation is an effective way, and constructing an effective obfuscation algorithm can increase the cost of reverse software. It is conspicuous that current development of code obfuscation focuses on increasing the complexity of the code structure without paying much attention to the protection of program semantic information, which may help experienced attackers improve their analysis efficiency. This paper proposes *COOPS* for protecting software based on program semantic information, in which functions are regarded as basic semantic units. The switch relationship between the intrafunction control flow and the interfunction calling is established. The interfunction calling can be hidden in the intrafunction control flow, and in reverse, the intrafunction control flow can also be converted to interfunction calling. In this way, considering intraprogram function semantic unit level discrete, this method reconstructs the intraprogram semantic relationship. To determine the relative effectiveness, we have evaluated *COOPS* on *OpenSSL* and *SpecInt-2000* test sets. For both of them, the function calling graphs before and after obfuscation differ more than 90%, which means *COOPS* significantly changes the control flow of the program. The evaluation shows that compared with *O-LLVM*, *COOPS* manifests strong resistance to *Asm2vec* and other program similarity analysis techniques and significantly improves the level of software protection rather than necessitating time-consuming and heavyweight problems.

## 1. Introduction

At present, the commonly used code obfuscation technology is to construct an effective obfuscation algorithm. Collberg [7] divides the code obfuscation into data obfuscation and control flow obfuscation. Currently, there are several studies on control flow obfuscation. And among various ways of that, the most popular is to obfuscate the intraprocess structure. State-of-art obfuscation algorithms include the opaque predicate algorithm [8–15] and the control flow flattening algorithm [16–20]. The former uses one-way transparency of opaque predicates to create a bogus branch that will not be executed, while the latter flattens the control flow of the program and uses a switch-case to jump. Rajba et al. [21] propose a code protection method for JavaScript. The core idea is to encrypt and encode data such as string arrays and identifiers in the code. Ahire et al. [22] protect the data arithmetic operation process and propose four obfuscation techniques applicable to all arithmetic operators. The abovementioned two methods achieve data protection by encrypting and replacing the data in the program.

The existing obfuscation algorithms aim at improving the complexity. For example, Sharif et al. [23] proposed a conditional obfuscation algorithm, which used hash value to replace the constant in the path branching condition to strip it from the original program besides encrypting the code block to increase the difficulty of reverse engineering. Xie et al. [24] proposed a binary code obfuscation method based on the random fuzzy table and hash coding to prevent reverse analysis of the stack trace, where the fuzzy table maps the calling address. If the program runs, the hash code and random value encode/decode the data of the stack frame. Linn et al. [25] focused on the static disassembly process and proposed two algorithms to disturb the static disassembly, which failed the disassembly of executable files. Sha et al. [26] protect the function calling process using multi-

threaded mutually exclusive execution. The method creates multiple threads in a program. It assigns different functions to different threads for execution, so that the program execution trajectory with functions is converted as a unit from calling execution within a single thread to repeated switching execution between multiple threads. Yadav et al. [27] and Hataba et al. [28] implement code protection at the source code level, where the former uses a pointer multiplication mechanism to replace integer operations, and the latter diversifies the generated software by changing the control flow of the input program at the compiler level.

However, the abovementioned code obfuscation algorithm mainly focuses on how to increase the complexity of the internal structure of the code, and the complex code structure fail to hide the semantic information behind the structure. The stage semantic information obtained by the attacker during the reverse analysis process is still available. Experienced attackers can effectively improve the analysis efficiency by using the semantic information. Therefore, how to improve the protection of program semantics by code obfuscation technology is our necessary technique in this paper.

From the perspective of protecting program semantics, this paper proposes a code obfuscation method named *COOPS* to obscure program semantics. It is important to note that the code obfuscation is converted on the basis of the equivalence of overall semantics of a certain program. The obscure program semantics in this paper does not change the overall semantics of the program, but obscures the intraprogram semantic layers and the relationship between levels. To destroy the interfunction semantic relationship and the semantic information of the function itself, we regard the function as a basic semantic unit, and semantic obscurity is implemented both the control flow inside and outside the function. So that attackers are faced with more abstract function units and calling relationships. The core approach is based on function inlining and outlining. The initial mention of function inlining and outlining in code obfuscation is in the Collberg article [7]. This article only describes the obfuscation effect of this method, but there are no specific implementation details. In this paper, we initially combine the function inlining and outlining technology with the obfuscation strategy and present a novel control flow obfuscation method.

In order to verify the effectiveness of the obfuscation algorithm, we designed a code obfuscation system based on the *LLVM* [29] compiler. The system considers software programs written in C/C++ as input and *PE* binary files after obfuscation as outputs. We choose *OpenSSL* and *SpecInt-2000* as test set. Extensive tests are used to evaluate the effectiveness of the system in terms of code protection and the costs of operation caused by the obfuscation process. Experiments show that the system has effective obfuscation and is suitable for various complex code structures. After obfuscation, the difference in the calling graph of the software is more than 90%, and the complexity of the calling graph circle also increases by about 20%, which significantly improves the analysis difficulty. We select *Asm2vec* [30], *DeepBindiff* [31], *Safe* [32] and *BinDiff* [33] for similarity analysis, and the results show the system is strongly resistant to these techniques. The main contributions of this paper are summarized as follows:

(i) We initially propose the idea that code obfuscation based on semantic obscurity. A function is regarded as a semantic unit, and the switch relationship between the intrafunction control flow and the interfunction calling is established. In the level of control flow inside and outside the function, the intraprogram semantic hierarchy is changed. Through experimental comparison and analysis, it can effectively resist the similarity analysis techniques such as *Asm2vec*.

(ii) We apply function inlining and outlining technology to code obfuscation, and three outlining methods are proposed novelly.

(iii) We implemented an automatic code obfuscation system-based on *LLVM*, which can automatically obfuscate the input source code program. The source code and the experimental results are available at *GitHub* (https://github.com/Rookiellvm/COOPS).

## 2. Motivation

Figure 1 presents a high-level abstraction how an attacker can perform analysis on a program using semantic information. The figure below on the left depicts the program under the protection of conventional obfuscation techniques, where the semantic level inside the program has not changed. The functions themselves and interfunction relationship have clear semantics. The attacker deduces the behavior of the program through obtained semantic information and semantic relationship of each function. The design goal of the obfuscation method in this paper is to destroy interfunction semantic information and the function itself, and change the intraprogram semantic level. The figure below on the right depicts that the internal semantic level of the program is obfuscated by *COOPS*, where interfunction semantic logic is disordered. It becomes infeasible for the attacker to understand whether the analyzed function has the correct semantic function, let alone deeply understand the program.

## 3. System Design

In this chapter, we describe the obfuscation system design method in this paper. In Section 3.1, we provide an overview of the obfuscation system. In Section 3.2, we describe the control flow transformation methods used in the obfuscation system. In Section 3.3, we describe how to reconstruct the data relationship, that is, broken during control flow transformation. Combining control flow transformation methods with obfuscation strategies to implement program control flow protection is described in Section 3.4.

*3.1. Overview.* The overview of *COOPS* is shown in Figure 2. It is mainly divided into two stages. The first stage analyzes the calling relationship of the original program and inline the functions with higher importance. The so-called inlining

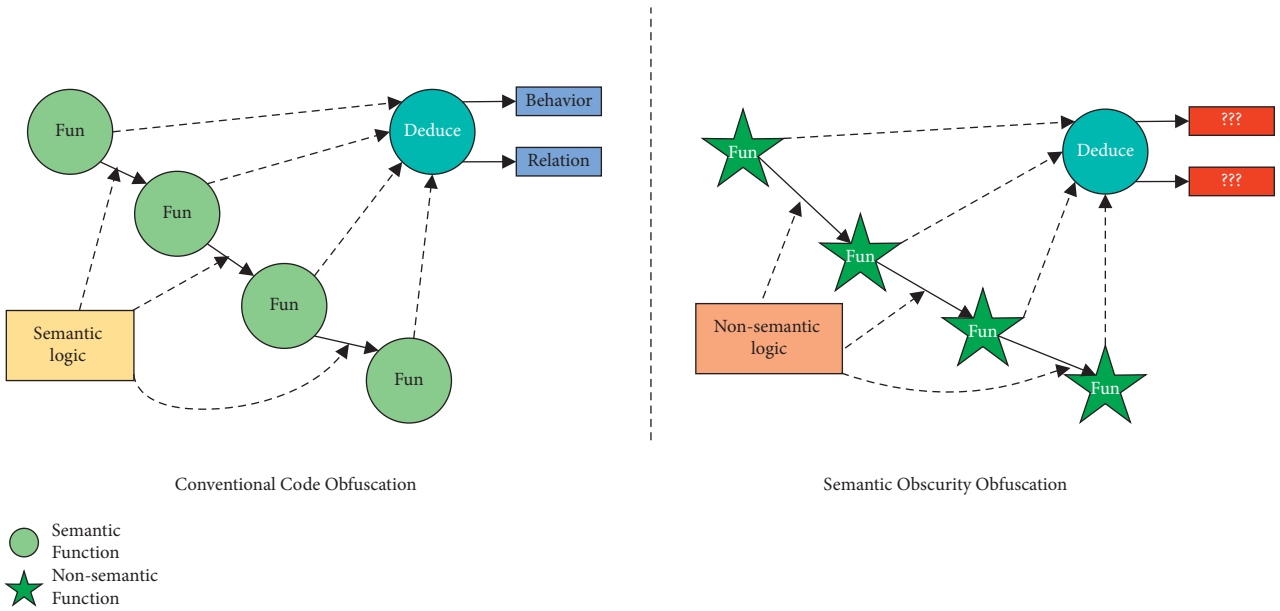


Figure 1: Semantic obscurity affects attack efficiency.

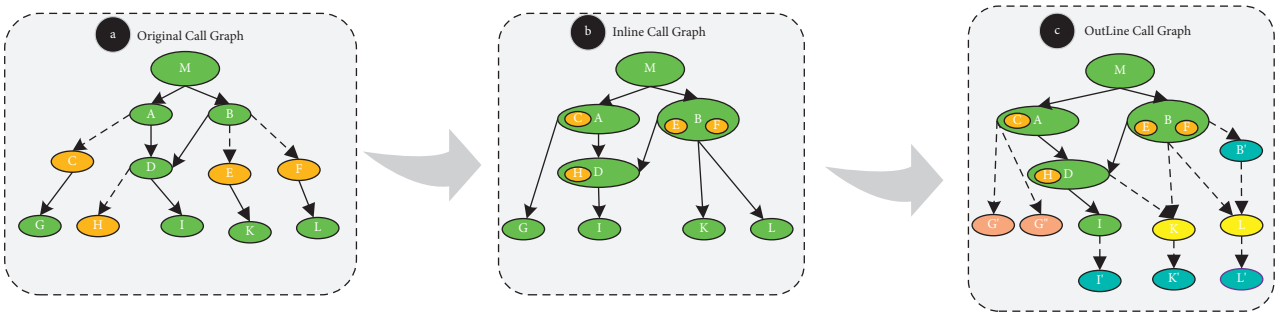


Figure 2: Overview of our approach. (a) Original call graph. (b) Inline call graph. (c) Outline call graph.

is to hide the called function into the caller's function, thereby eliminating the function calling relationship, which corresponds to the process of Figures 2(a) and 2(b). Assuming that functions $C$, $E$, $F$, and $H$ are functions of high importance, we inline these functions into $A$, $B$, and $D$. In the second stage, on the basis of inlining transformation, function outlining transformation is performed on the program. The so-called outlining is to transfer the code snippet to the outside of the function. *COOPS* consists of three outlining methods, corresponding to the process of Figures 2(b) and 2(c). The split outlining corresponds to function $G$ being converted into $G'$ and $G''$. The parent-child outlining corresponds to functions $B$, $I$, $K$, and $L$ being converted into subfunctions $B'$, $I'$, $K'$, and $L'$. The cross outlining corresponds to the outline code snippets of functions $B'$ and $D$ being converted to functions $K$ and $L$, in which there originally is not calling relationship between functions $B'\backslash D$ and $K\backslash L$. Section 3.2 of this chapter introduces inlining obfuscation methods and strategies, and Section 3.3 introduces three outlining methods and strategies. Three outlining methods are applied at different stages of the obfuscation process. *COOPS* uses inlining and outlining as the control flow transformation method, and combines the obfuscation strategy to achieve the purpose of control flow obfuscation.

*3.2. Control Flow Transformation Method.* Function inlining and outlining is a way to implement control flow transformation. In this paper, function inlining and outlining are used to implement control flow transformation. Combining inlining and outlining can effectively change the control flow structure of a program.

*3.2.1. Inlining Transformation.* Inlining transformation is a control flow transformation technique that eliminates function callings by inlining the called function into the caller's function. The inlined program will make the inlined function and its related callings disappear, and the semantic function unit composed of the original instruction set will be destroyed. In reverse analysis, it is only necessary to understand the function semantic unit, but it becomes the understanding of the instruction semantic unit. It is much more difficult for a reverse analysis program to understand a piece of instruction than to understand an independent function unit, because it is not clear that this piece of code

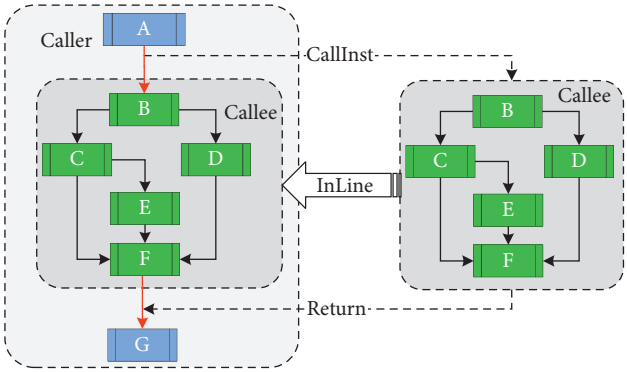


Figure 3: Inlining transformation.

represents a function semantic unit, so it will inevitably lead to the weakening of function semantics. For example, for the object-oriented *C++* language, when using the functions in one object, the program behavior can be quickly understood through the key function names and the reference relationship between the functions. When inlining these functions, we will find that these clues are hidden.

The function inlining transformation is shown in Figure 3, which inline the function named Callee into the function named Caller. The relationship between them is hidden in the control flow jump, which realizes the conversion from function calling to intrafunction control flow.

*3.2.2. Outlining Transformation.* Outlining transformation is a way of implementing intrafunction control flow to functions callings. Outlining transfers, the intrafunction code snippet to the interfunction calling, which destroys the control flow logic and function semantic information of the program, and increases the complexity of interfunction calling. The combination of inlining and outlining with the original function forms functions without semantic behavior. This paper mainly implements three outlining methods.

*Cross outlining* moves code snippets from one function to another, and there is no direct reference relationship between the two functions. As shown in Figure 4, there is no calling relationship between the function *F*1 and the function *F*2. The code snippet *D* in the function *F*1 is outlined to the function *F*2. When the function *F*1 needs to execute the code snippet *D*, the function *F*2 is called. This outlining method has two advantages, one is that a calling relationship can be formed between any two functions; the other is that when the called function is executed, a piece of its own code will be executed first, and then the outlining code snippet will be executed. The code snippet that really needs to be executed is hidden into the code of the called function.

*Parent-child outlining* moves the code snippet out of the function to form a subfunction call. When the code snippet is executed, the subfunction is called. As shown in Figure 5, the parent-child outlining is to outline the code snippets *S*1 and *S*2 in the function *F*1 to form a child function calling. The advantage of this outlining method is that it can expand the number of functions in the program and improve the

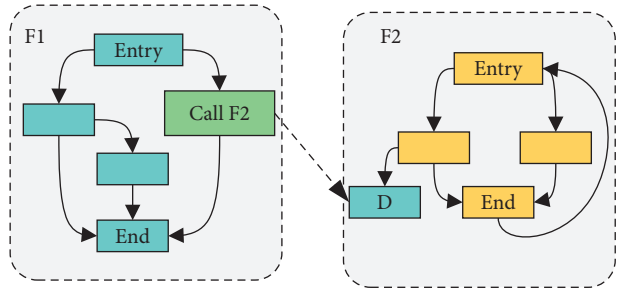


Figure 4: Cross outlining.

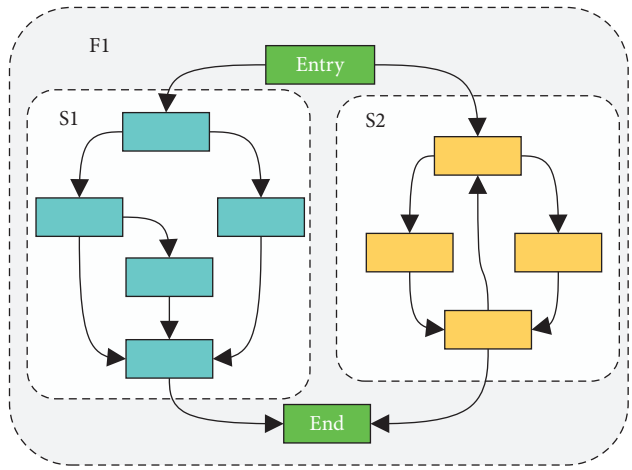


Figure 5: Parent-child outlining.

diversity of obfuscation. Both parent-child outlining and cross outlining transfer the code snippet within the function to execution outside the function. The two outlining methods in this paper move structures such as branches and loops inside functions to outside functions. Branches and loops expose important control flow information in functions. Outlining these structures can effectively protect control flow information.

*Split outlining* splits the control flow of a function into several independent parts, and different parts form different functions. However, due to the existence of conditional branches, the snippet of control flow needs to ensure the correct operation of the program. Therefore, it is necessary to analyze the function control flow graph before split to find that the control flow graph must pass through the nodes, and start from the must-pass nodes to cut. As shown in Figure 6, through the static analysis of the control flow graph, it can be found that the *basic blocks* that must pass when the function is executed are *Entry*, *A*, *B*, *C*, *D* and *End*. By cutting the function between *B* and *C*, two independent subfunctions *F*1 and *F*2 can be formed, and further, if there are multiple necessary points in the function, it can be divided into multiple independent functions. Different from the above-mentioned two kinds of outlining, there is no calling relationship between the split functions, but to ensure the execution of the split function correct in the caller. Split outlining can increase the number of functions in the program, and cooperating with cross outlining and parent-

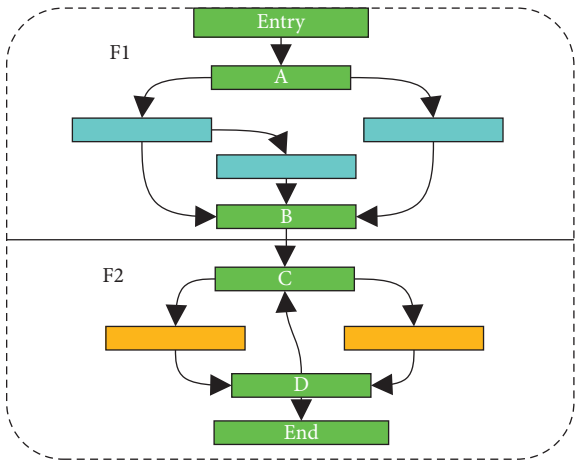


Figure 6: Split outlining.

child outlining can increase the complexity of the function calling relationship.

The abovementioned three outlining transformation methods move the code snippet intrafunction to the outside for execution. As shown in Figure 7, the outlining destroys the control relationship between *basic blocks* within a function and generates a new interfunction call. In the process of outlining transformation, the control relationship of *basic blocks A* to *B* is converted into *CallInst* instruction, in which data dependencies are passed through parameters; the control relationship between *basic blocks F* to *G* is converted into *Return* instruction, in which data dependencies are passed through return values; the control relationship within the outlining code snippet remains unchanged. Therefore, the conversion from intrafunction control relations to interfunction calls is proved equivalent. The construction of parameters and return values is described in Section 3.3.

*3.3. Data Relationship Reconstruction.* During inlining, the entire code segment of the function is inlined at the calling point. However, this process destroys the original data relationship, which needs to be reconstructed. Figure 8 shows the implementation process of inlining obfuscation. First, the code in the inline function is replicated to the calling point, and then variable remapping and *basic block* logic processing are performed. When the parameters inlined by the function are processed, the process of calling function parameters is transformed into the process of instruction assignment. Since the exit of the function is not unique, the return value is uncertain. Hence, all the exit *basic block*s at the intermediate semantic level are collected, and a tag variable is added when processing the return value. Based on the runtime result, the return value is determined at the *switch basic block* according to the tag scalar. Finally, the calling instruction and the function prototype are removed.

When the system performs outlining, outlining code snippets to form functions. The outlining of the code will destroy the data dependencies during program execution, so it is necessary to analyze the intraprogram data flow, find the relevant dependencies and complete the reconstruction.

*Definition 1.* Assume that *basic block* is composed of instructions, and Inst is an instruction in the *basic block*:

$$\begin{aligned}&\mathrm{def}[\mathrm{Inst}]\\&=\{v|\text{the set of variables defined when Inst is executed}\},\end{aligned}\tag{1}$$

$$\begin{aligned}&\mathrm{use}[\mathrm{Inst}]\\&=\{v|\text{the set of variables referenced when Inst is executed}\}.\end{aligned}\tag{2}$$

*Definition 2.* If variable $v \in \mathrm{def}[\mathrm{Inst}_i]$ and $v \in \mathrm{use}[\mathrm{Inst}_j]$, $\mathrm{Inst}_i \neq \mathrm{Inst}_j$, where $\mathrm{Inst}_j$ and $\mathrm{Inst}_i$ have an executable path. On this path, there is no statement to redefine $v$; hence, instruction $\mathrm{Inst}_j$ directly depends on $\mathrm{Inst}_i$ or instruction $\mathrm{Inst}_i$ directly dominates $\mathrm{Inst}_j$ on the data stream of variable $\mathrm{Inst}_j$, which is denoted as

$$DD\left(\mathrm{Inst}_j, \mathrm{Inst}_i, v\right).\tag{3}$$

As shown in Figure 9, "Code Snippet" is a code snippet obfuscated by outlining. Each *basic block* is represented as a set of instructions, in which each instruction is represented as a set of corresponding variables. The dependent instruction data in the outlining *basic block* set are analyzed to determine the return value and the parameters of the outlining function.

For each instruction $\mathrm{Inst}_i$ in the outlining code snippet, the set of variables referenced by instruction $\mathrm{use}[\mathrm{Inst}_i]$ are calculated (Phase ①). For each instruction $\mathrm{Inst}_j$ in the predecessor instruction set (Prev in Figure 9) of the outlining code snippet, the variable set $\mathrm{def}[\mathrm{Inst}_j]$ (Phase ②) defined by the instruction $\mathrm{Inst}_j$ are calculated. Each variable $v$ in $\mathrm{use}[\mathrm{Inst}_i]$ is used as the parameter of the outlining function (Phase ③), if the variable $v$ is in $\mathrm{def}[\mathrm{Inst}_j]$.

For each instruction $\mathrm{Inst}_i$ in the outlining code snippet, the variable set $\mathrm{def}[\mathrm{Inst}_i]$ (Phase ④) defined by the instruction $\mathrm{Inst}_i$ are calculated. For each instruction $\mathrm{Inst}_j$ of the subsequent basic block set (*Succ* in Figure 9) of the outlining code snippet, the variable set $\mathrm{use}[\mathrm{Inst}_j]$ (Phase ⑤) referenced by the instruction $\mathrm{Inst}_j$ are calculated. Each variable $v$ in $\mathrm{def}[\mathrm{Inst}_i]$ is used as the return value of the outlining function, if the variable $v$ is in $\mathrm{use}[\mathrm{Inst}_j]$. All variables used as the return value are stored in the *Ret Struct* (Phase ⑥).

*3.4. Control Flow Obfuscation Strategy.* Two types of control flow transformation methods are described in Section 3.2. In summary, inlining can hide function calls into intrafunction control flow. Outlining can transfer the intrafunction to interfunction call. The design goal of this paper is to destroy the internal semantic unit level of the program as much as possible, and obfuscate the control flow inside and outside the program. This section designs the control flow transformation strategy in stages, applies the abovementioned control flow transformation methods to different stages to achieve the destruction of the internal control flow of the program, and then protects semantic information of the program.

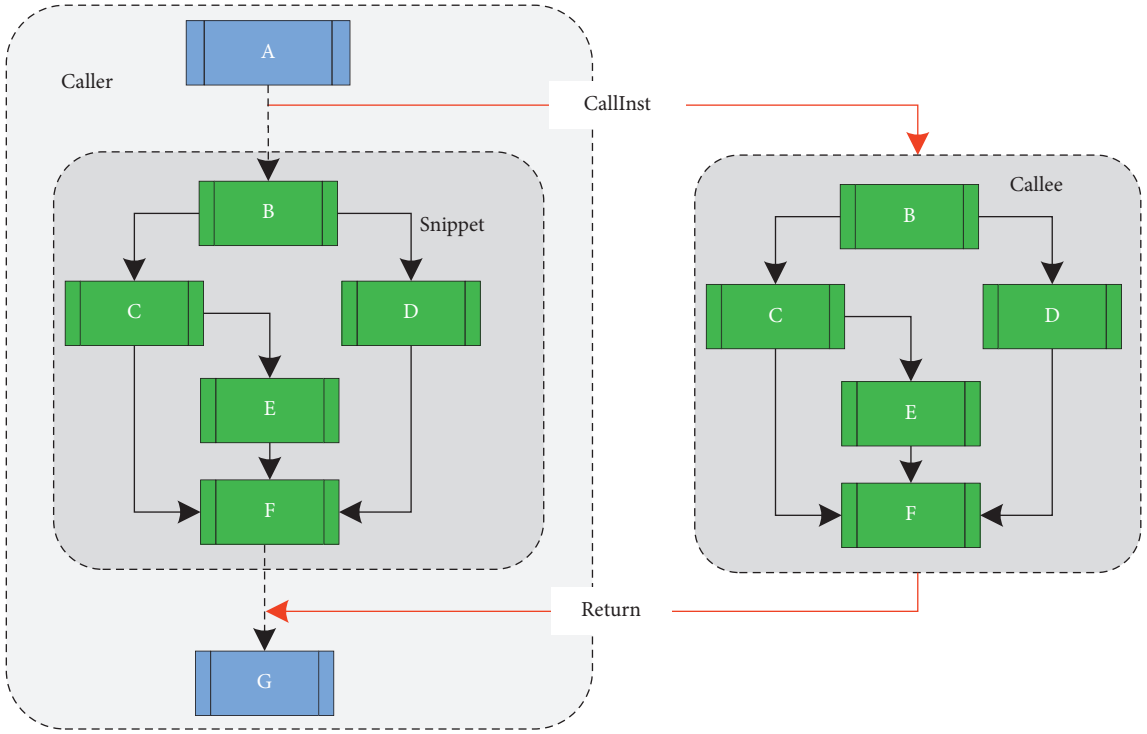


Figure 7: Outlining transformation.

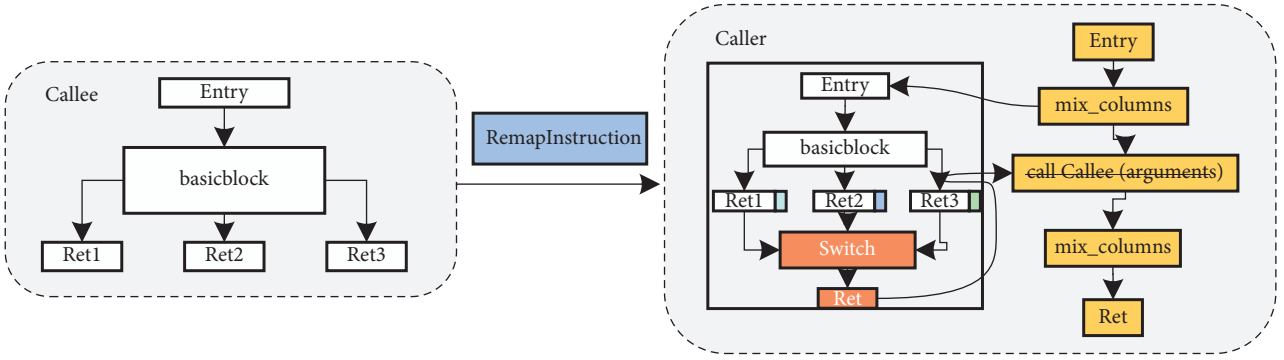


Figure 8: Reconstruction of the inlining data flow.

*Stage 1.* We hide highly important functions by inline transformation. The inlining transformation hides the function calling into the intrafunction control flow, and the inlining of the function with complex calling relationship can destroy the control flow logic to the greatest extent. Therefore, the object of the inline transformation is the important function in the program calling relationship. Function calling graphs can be abstracted into directed graphs with explicit pointing relationships, so vertex importance can be used to evaluate the importance of functions in function calling relationships. The connectivity between vertices and other vertices is of great significance for judging the importance of vertices. That is, the more paths through a vertex, the more important that vertex is. Therefore, the path-based vertex importance is specifically defined as:

*Definition 3.* For a vertex, the number of all paths passing through the vertex is used to judge the importance of the vertex. Let $v_i$ represent the $i$ function in the program, then the calculation formula of the importance can be expressed as

$$S_p(v_i) = I_p(v_i) \times O_p(v_i), \tag{4}$$

where $I_p(v_i)$ represents the number of paths ending with vertex $v_i$, and $O_p(v_i)$ represents the number of all paths starting with vertex $v_i$. Different from the conventional directed graph, the function calling graph has two particularities: (1) the function calling graph is a directed graph with rings; (2) the function calling graph has a definite starting point and end point.

For the former, the ring of the directed graph may cause the vertex path to explode, so the back-edge in the ring is pruned to convert the graph with rings into an acyclic graph.

For the latter, since the function calling graph has a definite starting point (main function) and a definite end point (leaf node), the calculation formula for the number of

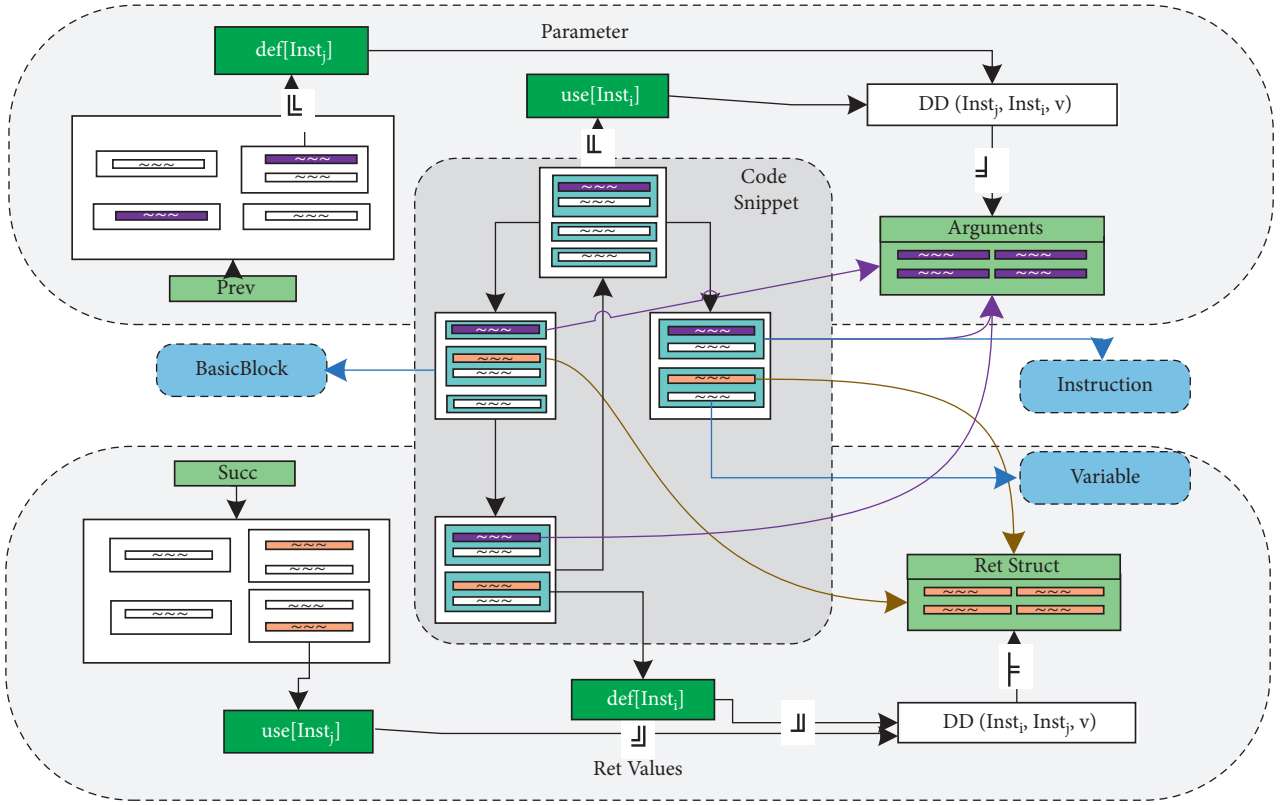


Figure 9: Reconstruction of the outlining data flow.

vertex paths can be quickly converged after the pruning of the backward edges is completed. It can be seen from formulas (2) and (3) that the number of paths passing through a vertex can be solved by calculating $I_p(v_i)$ and $O_p(v_i)$ respectively, and this calculation process is recursive:

$$I_p(v_i) = \sum_{u \in I(v)} I_p(u),$$
$$O_p(v_i) = \sum_{u \in O(v)} O_p(u). \quad (5)$$

Among them, $I(v)$ represents the set of vertices that have a direct calling relationship with vertex $v_i$, and $O(v)$ represents the set of vertices that have a direct calling relationship with vertex $v_i$. Since the function calling graph has a definite start and end point, the recursive calculation process is deterministically convergent. The function with high importance in the program is obtained by formula (1) for inline transformation.

*Stage 2.* The split outlining can split functions into different functions. Therefore, the split outlining is performed on the function that was inlined in the first stage, and the inline code and the function interleaved by the inline function are split into different ones. As a result, the function semantic information with high importance is discrete.

*Stage 3.* The function call spanning tree is constructed to implement parent-child outlining, and the function call process is similar in form to the depth-first-based multi-fork tree node traversal. Therefore, building a multi-fork tree would be an abstract solution for implementing function outlining: suppose the function calling graph is shown on the left of Figure 10. Traverse the function calling graph, generate a multi-fork tree $A$ that describes the function calling relationship, and build a target spanning tree $B$ on the basis of the multi-fork tree $A$. The tree $B$ has the following requirements: (1) complete multi-fork tree; (2) tree B has more layers than tree A. In this example, the number of layers of the target spanning tree $B$ is 3, and the number of forks is 3. The nodes generated in the process of constructing the target spanning tree $B$ are used as candidates for parent-child outlining to provide a variety of choices for the third stage.

*Stage 4.* The real function calling graph has back-edges, and Cross outlining is implemented on the constructed target spanning tree with the help of the Barabasi–Albert model [34] to generate the final function calling graph. The Barabasi–Albert model is a graph generation algorithm for the preferential connection model. Preferential attachment means that the more connections between vertex, the greater the possibility of accepting new connections. The core idea of the algorithm is to add an edge between the vertex and the vertex in the original graph for a new vertex in the graph. The probability of connection is $k_i / \sum_{i=1}^{n} k_i$, where $k_i$ represents the degree of vertex $i$ in the original graph. Referring this model to the calling graph, the more connections between vertexes, the greater the importance of functions. However, the purpose of the obfuscation in this paper is to hide the importance of the function, and the probability of transforming the model connection is as follows:

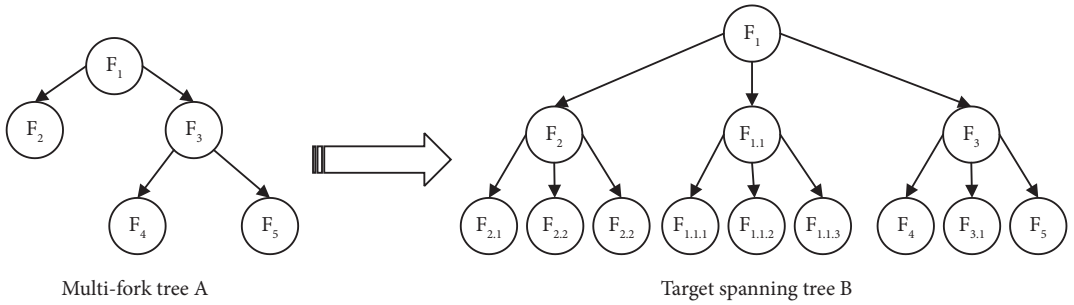


Figure 10: Build target spanning tree.

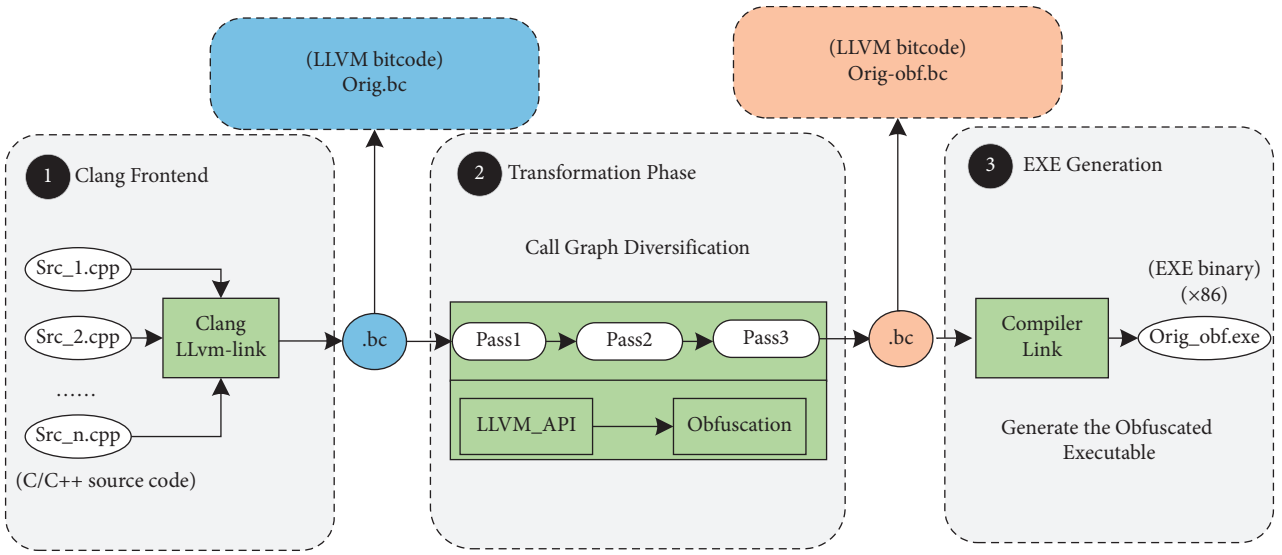


Figure 11: Architecture of the code obfuscation system.

$$P_i = \frac{\sum_{i=1}^{n} k_i - k_i}{\sum_{i=1}^{n} k_i}. \quad (6)$$

We select the subfunctions generated in the second stage and the functions with low importance in the original program as new vertexes, and perform cross outlining according to probability (5). The probability calculated from this will reduce the probability of connecting vertexes with high importance, increase the probability with low vertex degree, and increase the importance of low vertex degree. The calling graph constructed through the abovementioned three stages have no important vertexes. Each vertex has the same status in the calling graph, and the calling relationship between functions is completely reconstructed.

*Summary*: After the abovementioned four stages, the inlining and outlining is applied to the transformation of the program control flow, so that the transformed program calling graph has no important nodes, and each node has the same status in the calling graph. The calling relationship is reconstructed. Intrafunction code is discretized into new subfunctions and other unrelated functions, breaking the semantic hierarchy inside the program.

## 4. System Implementation

In this section, the design and implementation of a compiler-level tool developed to achieve automatic obfuscation of programs is presented. The LLVM framework, an extensible program optimization platform, provides a large number of APIs to analyze and modify the intermediate language codes. As shown in Figure 11, this paper implements the obfuscation system based on the LLVM. The system uses the source code written in C/C++ as the input and outputs the obfuscated binary file after the obfuscation system, which has three phases:

(1) Front-end code parsing phase

(2) Analysis and transformation phase

(3) Binary file generation phase

## 5. Experimental Design and Evaluation

In this chapter, we evaluate *COOPS* performance using 6 applications and compares it with the *O-LLVM* [35] obfuscation technique.

Table 1: Information of the benchmarks.

| Program | Size (kB) | Description | Instr number | Func number |
|---|---|---|---|---|
| *AES* | 87 | Encryption algorithm | 10046 | 36 |
| *RSA* | 60 | Encryption algorithm | 3078 | 49 |
| *Bzip* | 95 | Compression algorithm | 8131 | 75 |
| *Gzip* | 111 | Compression algorithm | 11321 | 96 |
| *Parser* | 200 | Grammar parser | 27815 | 324 |
| *Twoif* | 262 | Simulated annealing algorithm | 72353 | 191 |

*5.1. Evaluation Platform and Benchmarks.* In this section, we explain how the *COOPS* obfuscation method is tested and verified. The machine used for testing has an Intel Core i7-9700 CPU @ 3.00 GHz with 32 GB memory and Windows 10 as the operating system.

We evaluated the test sets commonly used in the field, including the cryptographic algorithms *RSA* and *AES* in *OpenSSL*, *Bzip*, *Gzip*, *Twoif*, and *Parser* in *SpecInt-2000* [36]. Table 1 shows the basic information of the test suite. The first column is the test program name, the second column is the size of the program, the third column is a brief description of the program function, the fourth column is the number of program instructions, and the fourth column is the number of program functions.

*5.2. Evaluation of System Protection Effect.* In this section, we analyze the antianalysis capability of the software under the protection of the obfuscated system. Antianalysis ability is an important indicator to evaluate the increase of the cost of software analysis by obfuscation methods. In this section, we intend to evaluate from two aspects. One is obfuscation system resistance to similarity analysis; the other is obfuscation strength.

Similarity analysis can eliminate the confounding factors faced by reverse analysis. A common situation is that a statically linked lib library is carried during software development, and a large number of library functions can be excluded by similarity analysis with known library function codes to improve analysis efficiency. This section evaluates the resistance of *COOPS* to similarity analysis using *Asm2vec*, *DeepBindiff*, and *Safe* academic similarity analysis techniques in 5.2.1, as well as commercial-grade *BinDiff*.

The obfuscation strength is used to evaluate the complexity increment of the obfuscated program. Anckaert et al. [37] proposed to evaluate obfuscation strength based on complexity measure in software engineering. In this section, we adopt the software cyclomatic complexity in Section 5.2.2 to evaluate the obfuscation strength.

*5.2.1. Program Similarity Analysis.* *Asm2vec*, *Safe* and *DeepBindiff* are the techniques used in the academic community to perform binary similarity analysis. By matching known functions through similarity analysis, the influence of obfuscation is eliminated and the analysis efficiency is improved. *BinDiff* is a tool used by the business community for similarity comparison analysis. It compares the similarity of binary files from five dimensions: *Function*, *Calls*, *BasicBlock*, *Jumps*, and *Instructions*.

Obfuscator-LLVM (*O-LLVM*) [35] is applied beyond *LLVM*'s intermediate language (*IR*) to generate more complex binary files by transforming *IR*. *O-LLVM* includes three different obfuscation techniques: control flow flattening (*FLA*), instruction substitution (*SUB*), and bogus control flow (*BCF*).

*FLA* flattens the control flow of the program, adding new conditional predicates to hide the control logic of the original program through switch-case. *FLA* greatly changes the internal control flow structure of the program.

*BCF* adds a large number of conditional judgments to the program by adding opaque predicates and garbage instructions to the program. The program after *BCF* obfuscation adds a lot of conditional branches, which differentiates between the control flow and the original program significantly.

*SUB* is to replace the operation in the program with a more complex operation, for example, replace the *AND* operation with $a = (b \sim c) \,\&\, b$. *SUB* significantly changes the sequence of instructions in the program.

*O-LLVM* has greatly changed the control flow structure of the original program. Based on the abovementioned similarity analysis techniques and tools, the following 4 experiments were set up.

*Experiment 1.* To compare similarities, we use *Asm2vec*, *Safe*, and *DeepBindiff*.

Experimental setup: we considered the original *AES* program as the benchmark comparison object, and the compare the similarities between *RSA*, *Bzip, Gzip, Parser*, *Twoif* and obfuscated *AES (AES-Obf)*.

Experimental results: the experimental results are shown in Figure 12. The similarity between *AES-Obf* and *AES* is at a low level, indicating that *COOPS* significantly changes the internal control flow structure of the program and can effectively resist the similarity comparison technology such as *Asm2vec*. The similarity comparison results obtained by *AES-Obf* and the complex program *Twoif* are similar, indicating that the degree of difference of the programs after *COOPS* obfuscation has reached the level of complex programs. Technologies such as *Asm2vec* extract the semantic information of the assembler by analyzing the assembly code. Since both *RSA* and *AES* are cryptographic algorithms, the similarity between the two is high, while the similarity of the obfuscated *AES-Obf* is low, which also reflects that *COOPS* obscures the semantic information of the program.

*Experiment 2.* Comparing *COOPS* with *O-LLVM* using *Asm2vec*.

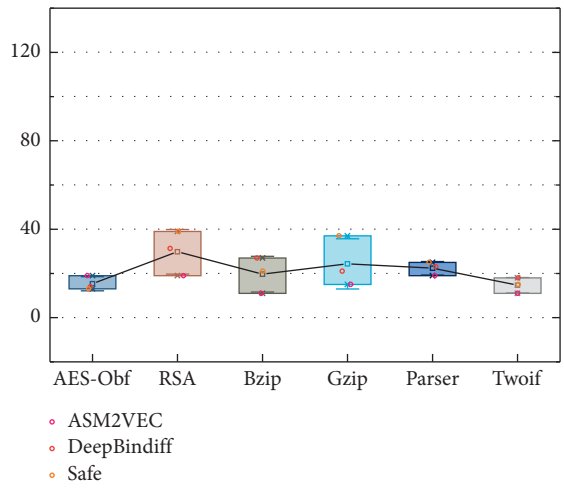


Figure 12: *Asm2vec*, *DeepBindiff* and *Safe* similarity comparison.

Experimental setup: we use *BCF*, *FLA*, *SUB*, *MIX (BCF + FLA + SUB)* and *COOPS* to obfuscate the 6 applications respectively, and use *Asm2vec to* compare the similarities between the programs before and after the obfuscation.

Experimental results: the experimental results are shown in Figure 13. Compared with the other four obfuscation methods, *COOPS* has better control flow obfuscation effect, and the program similarity before and after *COOPS* obfuscation is maintained at about 30%. It can be seen that *COOPS* is better than *O-LLVM* obfuscation techniques in the face of similarity analysis techniques such as *Asm2vec*.

*Experiment 3.* We used *BinDiff to* analyze the similarities among five dimensions of programs under *COOPS* protection.

Experimental setup: using *BinDiff* to compare the similarity of the five dimensions of *Function*, *Calls*, *BasicBlock*, *Jumps*, and *Instruction* before and after the obfuscation of the 6 applications.

Experimental results: the experimental results are shown in Figure 14. *COOPS* caused confusion of all five dimensions of the program. There are large differences in the five dimensions before and after confusion. The similarity between functions is slightly higher than that of the other four dimensions. This result shows that the obfuscation algorithm obtains a better control flow obfuscation effect without causing great damage to the function, reflecting the lightweight advantage. At the same time, it can be seen that the similarity between Calls and Jumps remains at the low point of the five dimensions. The results show that *COOPS* has a strong confounding effect on the function calling relationship and the control flow within the function.

*Experiment 4.* Comparing *COOPS* with *O-LLVM* using *BinDiff*.

Experimental setup: using *BCF, FLA, SUB, MIX (BCF + FLA + SUB),* and *COOPS* to obfuscate 6 applications respectively, and compare the similarities between programs before and after each obfuscation by *BinDiff*. The similarity

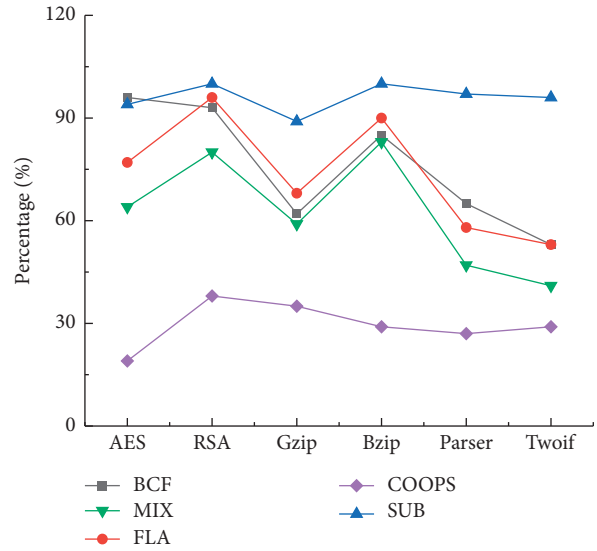


Figure 13: Comparing *COOPS* with *O-LLVM* using *Asm2vec*.

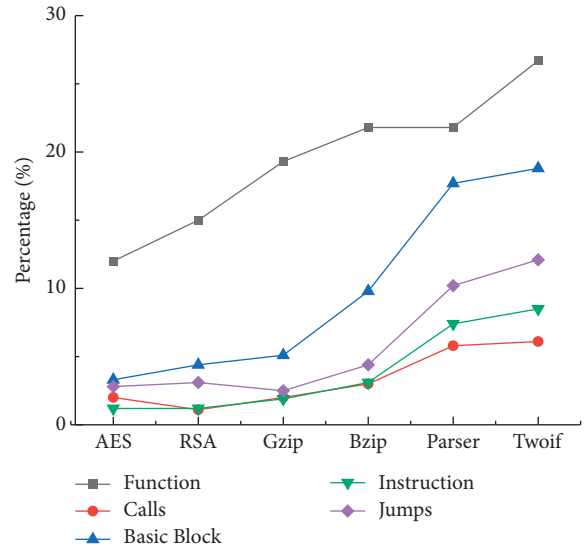


Figure 14: Analysis of the 5 dimensions of the program.

of the program is marked as the average of the similarity of the five dimensions of *BinDiff* comparison.

Experimental results: the experimental results are shown in Figure 15. In the comparison of the five dimensions of *BinDiff*, *COOPS* also reflects a better confusion effect than *O-LLVM*. The similarity of the five dimensions of the program before and after the obfuscation is about 15%, and the *O-LLVM* mixed obfuscation option MIX only obtains the similarity result of 30%. It shows that *COOPS* also has good resistance in the face of *BinDiff* commercial-level similarity analysis tool.

*5.2.2. Obfuscation Potency Evaluation.* Collberg [2] draws on the idea of complexity measurement in software engineering and proposes an index of obfuscation potency, in which the cyclomatic complexity increment is used to

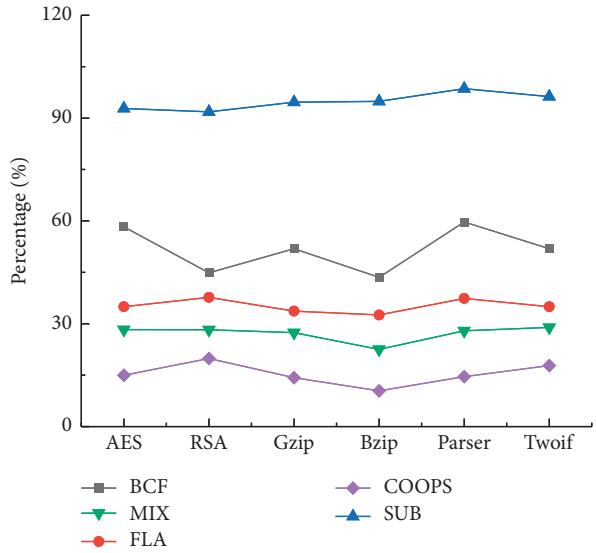


Figure 15: Comparing *COOPS* with *O-LLVM* using *BinDiff*.

measure the complexity of a software's internal structure. In this section, the cyclomatic complexity is used to measure the obfuscation potency, and the difference of the calling graph before and after obfuscation is included in the obfuscation potency evaluation.

Cycle complexity $C_p$ is used to measure the complexity of a module's decision structure, whose value is linearly independent of the number of paths, i.e., the minimum number of paths to be tested for reasonable error prevention. $C_p$ can be obtained by calculating the cyclomatic complexity of $\vartheta(G_P)$ and $C(G_P)$, i.e., $V(G_F) = C - F + 2P$, where $C$ and $F$ respectively are the number of edges and vertexes in the calling graph, and $P$ is the number of connected components of the graph, i.e., the maximum set of connected vertexes. Since the control flow graphs are all connected, $P$ is equal to 1. Therefore, the obfuscation can be expressed as

$$C_p\vartheta, G_P = 100\left[\frac{V(\vartheta(G_P))}{V(G_P)} - 1\right]. \tag{7}$$

The difference degree $D_p$, i.e., the difference of each calling graph $\vartheta(G_P)$ after obfuscation compared to $G_P$ before obfuscation, can be defined as

$$D_p\vartheta, G_P = 100\left[\frac{F+S}{T}\right]. \tag{8}$$

Here,

(i) $F = |C(\vartheta(G_P)) - C(G_P)|$ represents that the figure belongs to $\vartheta(G_P)$ but does not belong to $C(G_P)$ (the cardinality of the edge)

(ii) $S = |C(G_P) - C(\vartheta(G_p))|$ represents that the figure belongs to $C(G_P)$ but does not belong to $\vartheta(G_P)$ (the cardinality of the edge)

(iii) $T = F + S + |C(G_P) \cap C(\vartheta(G_P))|$, where $|C(G_P) \cap C(\vartheta(G_P))|$ is the cardinalate of intersection set between $\vartheta(G_P)$ and $C(G_P)$

Table 2: Obscurity of the benchmarks (%).

| Program | $C_p$ (%) | $D_p$ (%) |
|---|---|---|
| *AES* | 20.2 | 92.7 |
| *RSA* | 25.6 | 93.6 |
| *Bzip* | 13.6 | 96.9 |
| *Gzip* | 23.8 | 94.1 |
| *Parser* | 32.5 | 94.2 |
| *Twoif* | 33.5 | 97.2 |

As shown in Table 2, the difference degree of calling graph before and after obfuscation reaches more than 90%. As the number of functions of the obfuscated program increases, the difference degree of calling graph is greater. On the whole, the obfuscated calling graph is close to reconstruction, which is significantly different from the original program calling graph. At the same time, the cyclomatic complexity of the program has also increased, which shows that the increased complexity of the callings between the functions of the *COOPS* makes the calling relationship more complicated after obfuscation, and effectively improves the anti-analysis ability of the program.

*COOPS* can provide multiple rounds of iterative obfuscation, and multiple *COOPS* can be applied to improve the obfuscation intensity and increase the diversity and complexity of the function calling graph. The complexity of calling graphs and the cost of running time after applying *COOPS* for multiple times are summarized in Figure 16. It can be seen that under the time overhead of not more than 50%, when the number of iterations is 3 and 4, the cyclomatic complexity of the program increases by about 50%. In particular, the cyclomatic complexity of the complex program *Parser* and *Twoif* calling graphs increases by more than 90%. The number of obfuscated functions in complex programs is large, and the complexity of the generated calling graph is high, so the *COOPS* has better obfuscation effect on complex programs.

*5.3. Evaluation of System Efficiency.* This section compares *COOPS* and *O-LLVM* with the abovementioned three obfuscation options to evaluate program execution efficiency and code inflation rate.

The execution efficiency of the program $T_{\text{cost}}$ is the obfuscated program execution time $T(\vartheta(P))$ divided by the original program execution time $T(P)$, i.e.,

$$T_{\text{cost}}\vartheta, P = \frac{T(\vartheta(P))}{T(P)}. \tag{9}$$

Similarly, the code inflation degree $S_{\text{cost}}$ is the division of program disk overhead after obfuscation $S(\vartheta(P))$ to the original program disk overhead $S(P)$:

$$S_{\text{cost}}\vartheta, P = \frac{S(\vartheta(P))}{S(P)}. \tag{10}$$

Each program in the test set is obfuscated. To accurately measure the running time of each program, they are executed one hundred times, and the average running time before and after obfuscation is calculated.

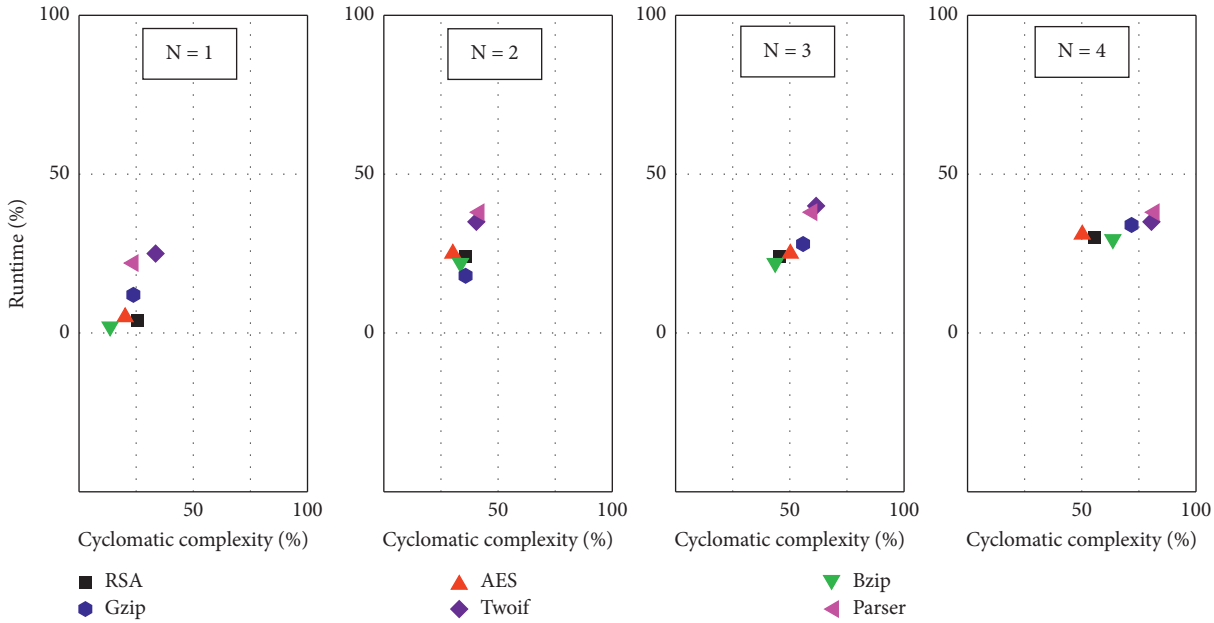


Figure 16: Iterative obfuscation cyclomatic complexity analysis.

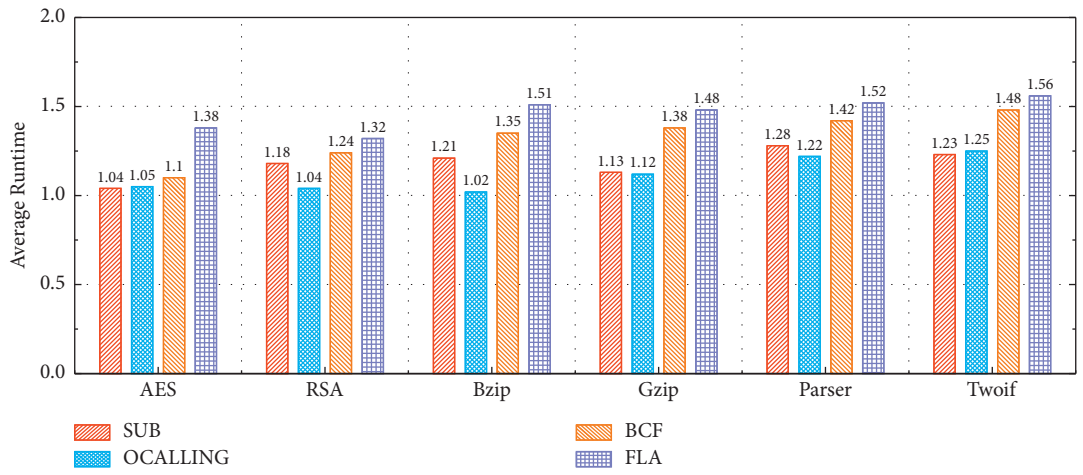


Figure 17: Comparison of average cost of runtime.

Figure 17 shows the time cost comparison between the obfuscation methods in *COOPS* and OLLVM before and after the obfuscation. The *x*-axis represents each test program, and the *y*-axis represents the ratio of cost of time of postobfuscation to preobfuscation. The results show that the cost of time of *COOPS* is close to that of OLLVM, and it has almost no effect on obfuscation of simple programs such as *AES* and *RSA*, and the overall the cost of obfuscation time is less than 25%.

Figure 18 compares the obfuscation methods in *COOPS* and OLLVM in terms of the cost of disk before and after the obfuscation. The *x*-axis represents each test program, and the *y*-axis represents the ratio of cost of disk of postobfuscation to preobfuscation. The results show that the cost of disk of *COOPS* is close to that of OLLVM, but the cost of disk of complex programs increases significantly.

## 6. Related Works

*COOPS* is a code obfuscation method based on control flow transformation, which effectively protects the program control flow. In recent years, the new direction of new control flow obfuscation is to expand the obfuscation process from intraprocedure to interprocedure. The control flow transfer becomes the program and the system to cooperate with. The state-of-art research summarized below is similar to the obfuscation method of function calling relationship proposed in this paper.

In [38], Miguel et al. proposed a mechanism based on the routing of function calls to realize the static obfuscation and diversification of the function calling graph. Compared to the original software, the obfuscated calling graph had an average difference of 25% in structure, and the fuzziness was increased from 20% to 30%. In all cases, only 3% of the

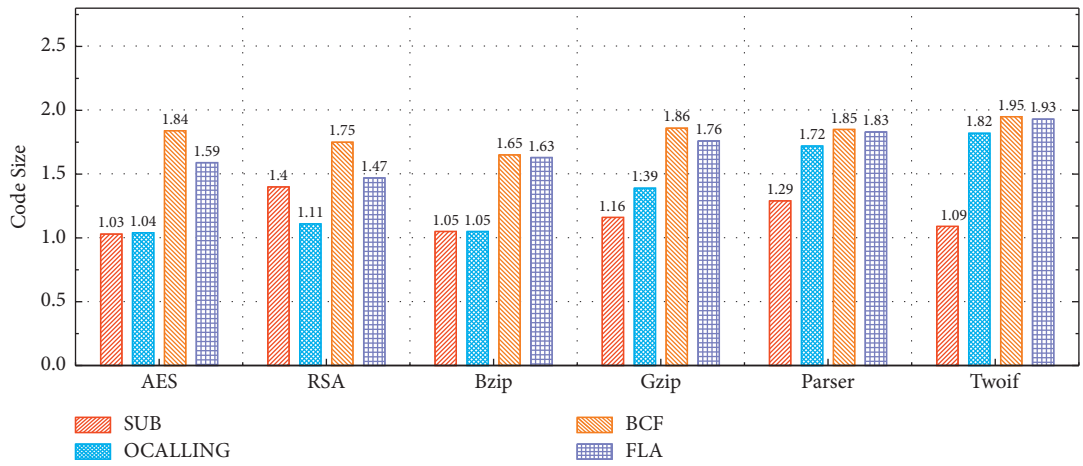


Figure 18: Comparison of code size.

execution time was spent. This mechanism allowed the efficient reconstruction of the entire calling graph, thereby improving the protection level without significantly affecting the software performance.

In [39, 40], the authors proposed an obfuscating method to extract code snippets from original functions and store them in another function, i.e., each function had code snippets of other functions. The correct execution of the program was realized by unconditionally jumping to the address of the basic block stored in the new function. In [40], Vivek et al. guaranteed the correct execution of the instruction by returning the instruction to the end function of the basic block. Therefore, there were many return instructions in the main body of the function, so the reverse tool was unable to correctly identify the boundary of the function. However, the program could execute correctly, which effectively obfuscated the calling relationship between the functions.

In [41], the authors obfuscated the branching condition containing the control flow logic of the program. They used lambda calculus to hide the computational semantics of the original condition and replaced the conditional jump instruction with the Lambda calculus function calling. This obfuscation method can protect the key branching conditions from symbolic execution technology with only moderate overhead.

Tatsuya et al. imitated the obfuscation of the control flow in functions, flattened and obfuscated the function calling graph, employed the *switchfunc* function, and used random numbers to ensure the proper scheduling of functions [42]. However, this method was easily influenced by dynamic analysis and had certain obfuscating features.

The algorithm proposed in [43] generating the bogus function calling graph, which changed the called function before running the target program, and the program called the Hook method at runtime to ensure its correct execution. Although this method effectively resisted the static analysis of programs, it failed to resist the dynamic analysis.

In [44], the authors removed the edge in the function calling graph, i.e., the address of the calling function was stored in a table at a different position from the obfuscated program. When the program was executed, the obfuscation program looked up the address table and restores the function calling relationship.

In [45], the authors proposed an obfuscating method based on *Return Oriented Programming* (ROP). This method focused on transforming the direct control flow into an indirect one, dividing the code in the binary file into a *basic block*, and converting that *basic block* into code snippets through binary tools. All identified direct control flows were hidden by RET instruction, and all code snippets were added to the original file to regenerate the obfuscated binary file.

The control flow mixing algorithms mentioned above can be divided into two groups. One is obfuscating a specific structure example in the function, as in [39, 40, 45], for jumping between *basic block*s. The others, e.g., [40–44], scramble and obfuscate the function calling graph. The difference between *COOPS* and the studies covered above is that our proposal obfuscates the jumps between *basic blocks* and function-calling graphs at the same time. Besides, according to dynamic and static program analysis, there occurs strategic obfuscation. To conclude, *COOPS* provides more diversified software calling graphs, achieving a better hiding effect on the key structures in the programs. This richer diversity can better protect software from code reverse engineering.

## 7. Conclusions and Future Work

This paper introduces *COOPS*, a code protection method based on the obscure semantics. *COOPS* starts from program semantics, regards functions as basic semantic units. The switch relationship between the intrafunction control flow and the interfunction calling is established, in which the original semantic level of the program is destroyed. Experiments show that *COOPS* can effectively change the control flow structure of the program, and has excellent resistance to similarity analysis techniques such as *Asm2vec*. The difference in the degree of program calling relationship between before and after obfuscation reaches more than 90%. *COOPS* makes the reverse analyzer have to face more abstract functional units and significantly increases the cost of code reverse engineering.

However, COOPS only implements inlining and outlining, and the obfuscation strategy is relatively simple. We can expand it in future work to increase the diversification of its obfuscation techniques.

## Data Availability

The source code and the experimental results are available at GitHub (https://github.com/Rookiellvm/COOPS).

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work was supported by the National Key Research and Development Project (2016YFB08011601).

## References

[1] M. Al Kadri, M. Nassar, and H. Safa, "Transfer learning for malware multi-classification," in *Proceedings of the 23rd International Database Applications & Engineering Symposium*, Athens, Greece, June 2019.

[2] M. Nassar and H. Safa, "Throttling malware families in 2d," 2019, https://arxiv.org/abs/1901.10590.

[3] Y. Awad, M. Nassar, and H. Safa, "Modeling malware as a language," in *Proceedings of the 2018 IEEE International Conference on Communications (ICC)*, IEEE, Kansas City, MO, USA, May 2018.

[4] H. Safa, M. Nassar, and Wael Al Rahal Al Orabi, "Benchmarking convolutional and recurrent neural networks for malware classification," in *Proceedings of the 2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, IEEE, Tangier, Morocco, June 2019.

[5] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," *International Workshop on Information Hiding*, Springer, Berlin, Heidelberg, 2011.

[6] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and Z. Wang, "Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling," *Computers & Security*, vol. 74, pp. 202–220, 2018.

[7] C. Collberg, C. Thomborson, and D. Low, *A Taxonomy of Obfuscating transformations*, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[8] C. Collberg, T. Clark, and L. Douglas, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, MA, USA, January 1998.

[9] G. Myles and C. Collberg, "Software watermarking via opaque predicates: implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[10] H. Xu, "Manufacturing resilient bi-opaque predicates against symbolic execution," in *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, Luxembourg, June 2018.

[11] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE, Miami Beach, FL, USA, October 2007.

[12] X. Xie, "Mixed obfuscation of overlapping instruction and self-modify code based on hyper-chaotic opaque predicates," in *Proceedings of the 2014 Tenth International Conference on Computational Intelligence and Security*, IEEE, Yunnan, China, November 2014.

[13] V. Sergeichik and I. Alexander, "Implementation of opaque predicates for FPGA designs hardware obfuscation," *Journal of Information, Control and Management Systems*, vol. 12, no. 2, 2014.

[14] Y. Yubo, F. Wenqing, H. Wei, X. Guoai, and Y. Yixian, "The research of multi-point function opaque predicates obfuscation algorithm," *Applied Mathematics & Information Sciences*, vol. 8, pp. 3063–3070, 2014.

[15] D. Xu, M. Jiang, and D. Wu, "Generalized dynamic opaque predicates: a new control flow obfuscation method," in *Proceedings of the International Conference on Information Security*, Springer, Bali, Indonesia, September 2016.

[16] K. Yakdan, "No More Gotos: Decompilation Using Pattern-independent Control-Flow Structuring and Semantic-Preserving Transformations," in *Proceedings of the The 2015 Network and Distributed System Security (NDSS) Symposium*, NDSS, San Diego, CA, USA, February 2015.

[17] T. László and Á. Kiss, "Obfuscating C++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[18] J. Cappaert and B. Preneel, "A general model for hiding control flow," in *Proceedings of the Tenth Annual ACM Workshop on Digital Rights Management*, Chicago Illinois USA, October 2010.

[19] J. Yi, "A security model and implementation of embedded software based on code obfuscation," in *Proceedings of the 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, IEEE, Guangzhou, China, January 2020.

[20] B. Johansson, P. Lantz, and M. Liljenstam, "Lightweight dispatcher constructions for control flow flattening," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, Orlando FL USA, December 2017.

[21] P. Rajba and W. Mazurczyk, "Data hiding using code obfuscation," in *Proceedings of the The 16th International Conference on Availability, Reliability and Security*, Vienna, Austria, August 2022.

[22] P. Ahire and J. Abraham, "Secure Cloud Model for Intellectual Privacy protection of Arithmetic Expressions in Source Codes Using Data Obfuscation Techniques," *Theoretical Computer Science*, vol. 922, 2022.

[23] M. I. Sharif, "Impeding malware analysis using conditional code obfuscation," *NDSS*, San Diego, CA, USA, February 2008.

[24] X. Xie, B. Lu, D. Gong, X. Luo, and F. Liu, "Random table and hash coding-based binary code obfuscation against stack trace analysis," *IET Information Security*, vol. 10, no. 1, pp. 18–27, 2016.

[25] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Singapore, March 2003.

[26] Z. Sha, "Model of execution trace obfuscation between threads," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[27] K. Yadav, "Source code obfuscation: novel technique and implementation," *ICT Systems and Sustainability*, pp. 197–204, Springer, Singapore, 2022.

[28] M. Hataba, A. Sherif, and R Elkhouly, "Enhanced obfuscation for software protection in autonomous vehicular cloud computing platforms," *IEEE Access*, vol. 10, Article ID 33943, 2022.

[29] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, IEEE, San Jose, CA, USA, March 2004.

[30] S. Ding, C. M. Benjamin, and P. Charland, "Asm2vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, San Francisco, CA, USA, May 2019.

[31] L. Massarelli, "Safe: self-attentive function embeddings for binary similarity," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, Lisbon, Portugal, June 2019.

[32] Y. Duan, "DeepBindiff: Learning Program-wide Code Representations for Binary Diffing," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, USA, February 2020.

[33] BinDiff, "BinDiff," 2001, https://www.zynamics.com/software.html.

[34] A. L. Barabási, "Network science," *Philosophical Transactions of the Royal Society A: Mathematical, Physical & Engineering Sciences*, vol. 371, Article ID 20120375, 1987 pages, 2013.

[35] P. Junod, "Obfuscator-LLVM--software protection for the Masses," in *Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection*, IEEE, Florence, Italy, May 2015.

[36] J. L. Henning, "Spec CPU2000: measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[37] B. Anckaert, "Program obfuscation: a quantitative approach," in *Proceedings of the 2007 ACM Workshop on Quality of protection*, Alexandria, VA, USA, October 2007.

[38] M. Rodríguez-Veliz, Y. Nuñez-Musa, and R. Sepúlveda-Lima, "Call graph obfuscation and diversification: an approach," *IET Information Security*, vol. 14, no. 2, pp. 241–252, 2020.

[39] V. Balachandran, W. K. Ng, and S. Emmanuel, "Function level control flow obfuscation for software security," in *Proceedings of the 2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*, IEEE, Birmingham, July 2014.

[40] V. Balachandran, S. Emmanuel, and W. K. Ng, "Return oriented obfuscation," in *Proceedings of the Eighth Int. Conf. On Networks & Communications (NETCOM-2016)*, Sydney, NSW, Australia, October 2016.

[41] P. Lan, "Lambda obfuscation," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, Springer, Washington, WA, USA, October 2017.

[42] T. Toyofuku, T. Tabata, and K. Sakurai, "Program obfuscation scheme using random numbers to complicate control flow," in *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, Springer, Berlin, Heidelberg, March 2005.

[43] K. Fukuda and H. Tamada, "An obfuscation method to build a fake call flow graph by hooking method calls," in *Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, IEEE, Las Vegas, NV, USA, September 2014.

[44] L. Jones, "Flowtables: program skeletal inversion for defeat of interprocedural analysis with unique metamorphism," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, Los Angeles, CA, USA, December 2015.

[45] D. Mu, "ROPOB: obfuscating binary code via return oriented programming," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, Springer, Washington, WA, USA, September 2017.