Research Article

Input-Output Example-Guided Data Deobfuscation on Binary

Yujie Zhao,^{1,2} Zhanyong Tang^{1b},² Guixin Ye,² Xiaoqing Gong^{1b},² and Dingyi Fang²

¹Department of Informationization Construction and Management, Shaanxi Normal University, Xi'an 710119, China

²School of Computer Science and Technology, Northwest University, Xi'an 710127, China

Correspondence should be addressed to Zhanyong Tang; zytang@nwu.edu.cn and Xiaoqing Gong; gxq@nwu.edu.cn

Received 6 July 2021; Accepted 18 November 2021; Published 13 December 2021

Academic Editor: Zhiyuan Tan

Copyright © 2021 Yujie Zhao et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Data obfuscation is usually used by malicious software to avoid detection and reverse analysis. When analyzing the malware, such obfuscations have to be removed to restore the program into an easier understandable form (deobfuscation). The deobfuscation based on program synthesis provides a good solution for treating the target program as a black box. Thus, deobfuscation becomes a problem of finding the shortest instruction sequence to synthesize a program with the same input-output behavior as the target program. Existing work has two limitations: assuming that obfuscated code snippets in the target program are known and using a stochastic search algorithm resulting in low efficiency. In this paper, we propose fine-grained obfuscation detection for locating obfuscated code snippets by machine learning. Besides, we also combine the program synthesis and a heuristic search algorithm of Nested Monte Carlo Search. We have applied a prototype implementation of our ideas to data obfuscation in different tools, including OLLVM and Tigress. Our experimental results suggest that this approach is highly effective in locating and deobfuscating the binaries with data obfuscation, with an accuracy of at least 90.34%. Compared with the state-of-the-art deobfuscation technique, our approach's efficiency has increased by 75%, with the success rate increasing by 5%.

1. Introduction

Data obfuscation is the transformation that obscures the data structures used in the application [1]. The goal of this obfuscation technique simply consists of replacing standard binary operators (like addition, subtraction, or Boolean operators) with functionally equivalent but more complicated sequences of instructions. In recent years, almost all malware has used data obfuscation to cover up its internal logic to avoid detection or prevent security analysts from reverse analysis, such as virus [2], repackaging [3], code cloning [4, 5], and privacy theft [6]. It makes detecting and analyzing malware more difficult, let alone the obfuscated program binaries without source code.

Deobfuscation is a transform that can remove obfuscation effects from the target program to solve the above problems. It is a reverse process of code obfuscation. Specifically, deobfuscation tries to obtain program P' through reverse analysis of the target program P ; while P' and P are functionally equivalent, P' is simpler and easier to understand. At present, most of the existing deobfuscation

techniques work on a specific kind of obfuscation [7–12], for example, layout deobfuscation [7], opaque predicate deobfuscation [8], control flow flatten deobfuscation [9], and virtualization deobfuscation [10–12].

There is almost no deobfuscation approach specifically for data obfuscation. There are three reasons for this. First of all, compared with other obfuscation algorithms, data obfuscation is more stealthy. That is because the number of instructions involved in data obfuscation is small, so it is difficult to detect [13]. Afterward, when most obfuscation tools implement data obfuscation, they randomly select one from the equivalent instruction sequences to increase the diversity of the transformation [14–16]. Even if one of the transformations is successfully reverse analyzed, it still takes the same or even more time and effort to deobfuscate other transformations. Finally, recovering data flow from obfuscated code requires a lot of expert experience and domain knowledge, including static and dynamic analysis.

According to the underlying principle, the existing deobfuscation algorithms are divided into two categories. The first category draws on the idea of compiler

optimization. Yadegari et al. [17, 18] first use a combination of bit-level taint propagation and control-dependence analysis to identify instructions related to input and output values. Then, they borrow the idea of compiler optimization to delete redundant instructions in terms of constant propagation and constant folding. Although compiler optimization and deobfuscation are both to simplify the code, they are essentially different [19]. Deobfuscation aims to generate more readable code. Compiler optimization aims to generate code that runs faster or consumes less. However, code with high execution efficiency does not mean that it is readable. Therefore, deobfuscation in the way of compiler optimization may lead to the generated code being too simplified.

The second category of deobfuscation is based on the program synthesis technique [20–23], which treats the target program as a black box. The underlying intuition is that a program’s semantics can be understood as a mapping from input values to output values. Therefore, deobfuscation becomes a problem of finding the best instruction sequence to synthesize a program with the same input-output behavior as the target program. The best here means that it is the easiest to understand. The advantage of this category is that there is no need for lots of expert experience and domain knowledge. Program synthesis technology does not need to pay too much attention to the semantic information and obfuscation algorithms inside the target program. They transform the tedious, lengthy, and subjective manual binary code reverse analysis into an automated task. But the disadvantage is that this category’s accuracy and efficiency are limited in factors such as user intent, program space, and search technique.

Syntia [22] is a state-of-the-art deobfuscation approach based on program synthesis, which automates code deobfuscation using program synthesis guided by Monte Carlo Tree Search (MCTS). To the best of our knowledge, Syntia is the first to propose a framework that combines program synthesis with an artificial intelligence algorithm. However, Syntia assumes that the obfuscated instructions are known. Thus, the first research question raised is **RQ1**: how to locate the obfuscated instructions in the target program? This is a prerequisite of enabling deobfuscation. In addition, MCTS is essentially a stochastic search algorithm, which leads to low efficiency of program synthesis. Thus, the second research question raised is **RQ2**: how to optimize the search efficiency of program synthesis while improving the success rate? This motivates us to do our work.

In this paper, we propose an approach of simplifying the binaries with data obfuscation and name it AutoSimpler. We borrow the idea of Syntia’s combining the program synthesis [21] and artificial intelligence algorithm. Specifically, our approach’s main idea is to replace the stochastic Monte Carlo Tree Search with a heuristic Nested Monte Carlo Search (NMCS) [24]. But there are some challenges in our work. On the one hand, how to locate the obfuscated instructions? Taking into account the fact that data obfuscation involves some arithmetic and logical operators, we select Entropy of basic blocks and N-gram on instructions as the feature, using Support Vector Machine (SVM) [25] and Logistic Regression with Gradient Descent (LR-GD) [26] as

a classifier to find the obfuscated instructions. On the other hand, how to improve search efficiency? The Nested Monte Carlo Search based on heuristic is the best choice, which combines nested calls and randomness in the playouts and memorizes the best sequence of moves.

Unlike past approaches, AutoSimpler relies on the advantages of the Nested Monte Carlo Search’s heuristics to improve program synthesis efficiency. Besides, AutoSimpler adopts machine learning to identify obfuscated instructions. That is different from previous work since they all assume that the obfuscated instructions are known [22, 23].

Contributions. This paper makes the following contributions:

We propose a fine-grained obfuscation detection for locating obfuscated code snippets, which selects the Entropy of basic blocks and N-gram on instructions as the feature to train the machine learning model.

We replace the stochastic search with a heuristic search in the deobfuscation framework of combining program synthesis and artificial intelligence to improve the efficiency of program synthesis while ensuring a slight increase in the success rate.

We implement a prototype of AutoSimpler and evaluate its accuracy and efficiency. The experimental results indicate that AutoSimpler is a deobfuscation tool for data obfuscated binaries with high accuracy of 90.34% and 23 seconds per task.

2. Background

In this section, we first analyze the complexity of data obfuscation by user study in Section 2.1. Next, we describe the details of program synthesis in Section 2.2. Finally, we define the problem of simplifying the obfuscated binaries in Section 2.3.

2.1. Complexity Analysis of Data Obfuscation. Data obfuscation is the transformation that obscures data structures used in source application [1]. Although data obfuscation usually involves only a few instructions (as shown in Figure 1), the difficulty of analyzing it is not reduced at all. On one hand, this kind of obfuscation transforms the data flow of the program. It is almost impossible to restore its complete semantics only through static analysis without dynamic analysis. On the other hand, this kind of obfuscation presents the characteristics of randomness and diversity. Even if we can reverse analyze a program with data obfuscation successfully, it does not mean that we can still analyze other data obfuscated programs in the same way.

To evaluate data obfuscation’s resilience, we follow a similar evaluation methodology described in Kuang et al. [27] to conduct a small-scale user study. Our user study involves five postgraduate students majoring in computer science. All the students have hands-on experience in software reverse analysis. They are asked to reverse engineer six binary programs. The samples are coming from the

sub	esp, 10h	mov	eax, [ebp+arg_C]	((((((a b) << 1) - (a & b)) c) + (((a b) << 1) - (a &
mov	eax, [ebp+arg_10]	add	eax, edx	b)) & c)) ^ ~ d) + (((((a b) << 1) - (a & b)) c) +
or	eax, [ebp+arg_18]	or	eax, [ebp+arg_8]	(((a b) << 1) - (a & b)) & c)) d) + (((((a b) << 1) -
lea	edx, [eax+eax]	mov	edx, eax	(a & b)) c) + (((a b) << 1) - (a & b)) & c)) d))) + 1
mov	eax, [ebp+arg_10]	mov	eax, [ebp+arg_10]	
xor	eax, [ebp+arg_18]	or	eax, [ebp+arg_18]	
sub	edx, eax	lea	ebx, [eax+eax]	
mov	eax, [ebp+arg_C]	mov	eax, [ebp+arg_10]	
add	edx, eax	xor	eax, [ebp+arg_18]	
mov	eax, [ebp+arg_8]	sub	ebx, eax	
not	eax	mov	eax, [ebp+arg_C]	
mov	ecx, edx	add	eax, ebx	
xor	ecx, eax	or	eax, [ebp+arg_8]	
mov	eax, [ebp+arg_10]	add	eax, edx	
or	eax, [ebp+arg_18]	add	eax, ecx	
lea	edx, [eax+eax]	add	eax, 1	
mov	eax, [ebp+arg_10]	mov	[ebp+var_8], eax	
xor	eax, [ebp+arg_18]	mov	eax, 0	
sub	edx, eax	add	esp, 10h	

(a)

(b)

FIGURE 1: Examples of data obfuscation. The original expression is $a + b + c + d$. It is obfuscated by Tigress with the option of *-EncodeArithmetic*. (a) Assembly code snippets with data obfuscation. (b) Expression with data obfuscation.

source code of gcc and have been obfuscated by OLLVM with an option of *-sub* or Tigress with an option of *-EncodeArithmetic*. Each participant is given 12 hours to accomplish a task: try to restore the original semantics of the obfuscated programs as much as possible and make the restored code have the same input-output behavior as the given program.

Table 1 shows the results of the user study. Although these programs only have 100 to 300 instructions, one of the five candidates does not complete any tasks, and the rest complete at least one. Only one candidate completes two tasks. The user study shows that manually implementing reverse analysis is a very difficult task, requiring a lot of time and effort. Therefore, it is necessary to transform the tedious, lengthy, and subjective deobfuscation process into an automated way without much more expert experience and domain knowledge.

2.2. Program Synthesis. Program synthesis is a technique to synthesize an executable program meeting user intent expressed in the form of some constraints [20]. There are three key dimensions in program synthesis: expression of user intent, space of programs over which to search, and the search technique. Program synthesis has been widely used in the following areas: automatic programming [28], peephole optimization [29, 30], and generating the best code snippets [31, 32]. Some works have proved the feasibility of program synthesis on deobfuscation [21–23]. Specifically, taking a given program as the specification of the equivalent program required in the target language, program synthesis can translate the given piece of code into a semantically equivalent code written in a certain target language. The code is more readable and easier to understand. The difference between these works is their realization of the three key dimensions.

The typical case of program synthesis applied in deobfuscation is the oracle-guided component-based program synthesis [21]. It takes the deobfuscation process as a black

TABLE 1: The results of user study on data obfuscation’s resilience.

Tools	Methods	Candidates				
		S1	S2	S3	S4	S5
OLLVM	Const_to_b100b	✓	✓	×	×	×
	if0_b100w_bitfield_0	×	×	×	×	×
	vrp07	×	×	×	×	✓
Tigress	Const_to_b100b	×	✓	×	✓	×
	if0_b100w_bitfield_0	×	×	×	×	×
	vrp07	×	×	×	×	×

box and learns a series of input and output mappings of the target program. Then, it uses a synthesizer to generate code fragments that have the same input-output behavior as the target program but are easier to understand.

2.3. Problem Definition. To explain the oracle-guided program synthesis more clearly, we give the problem definition. Let us denote the target program that needs to be deobfuscated as P , $\vec{I}$ is the input set of P , and O is the output set of P . So the input-output examples set E of program P can be expressed as $E = \{\langle \vec{I}, tO \rangle\}$. Then, $\langle \vec{I}_j, O_j \rangle$ represents one of the input-output examples in E . Among them, the number of input variables can be one or more, denoted as $(i_1, i_2, \dots, i_n) \in \vec{I}_j$.

For the convenience of description, we assume that all inputs and outputs are of the same type, and the program P has only one output. Besides, we also assume that the set of operators used in the obfuscated program, such as addition, subtraction, multiplication, and division, is recorded in the component library C .

The goal of deobfuscation on program P is to synthesize a candidate program P' according to a given component library C and input-output examples E , requiring P' to have the same input-output behavior as P , which is expressed explicitly as follows:

- (1) For any set of input $\vec{i}$, the output of program P on $\vec{i}$ is o and the output of program P' on $\vec{i}$ is also o .
- (2) The program P' is more readable than P .

The worst result is that all operators in the component library C are used up, and the target program cannot be synthesized. Ideally, the program can be synthesized by some of the operators in the component library C .

Note that the generated candidate program P' may use all or part of the operators in the component library C . If many candidate programs $\{P'_1, P'_2, \dots, P'_n\}$ satisfy the above conditions, we select the candidate program with the fewest operators as the deobfuscated result of the target program P .

In order to explain the problem more clearly, here is an example: given a piece of obfuscated code fragment P_0 whose internal logic is $((x \wedge y) + ((x \& y) \ll 1)) | z) + (((x \wedge y) + ((x \& y) \ll 1)) \& z)$, we only have its input-output examples such as $\langle (2, 2, 2), 6 \rangle$, $\langle (1, 5, 3), 9 \rangle$, and the given component library is $\langle +, *, \&, | \rangle$.

According to the input-output example of $\langle (2, 2, 2), 6 \rangle$ and the given component library of $\langle +, *, \&, | \rangle$, it can produce the following candidate programs, including $x + y + z$, $x * y + z$, $(x \& y) + (x | y) + z$.

When the input is $(1, 5, 3)$, the output of the program P_0 is 9, but the output of candidate program $x * y + z$ is 8, which is different from target program P_0 's output, so this candidate program is excluded.

After many times of verification, $x + y + z$ and $(x \& y) + (x | y) + z$ have the same output on any input $\vec{i}$ as the target program P_0 . However, $x + y + z$ uses fewer components than $(x \& y) + (x | y) + z$, so $x + y + z$ is the final deobfuscated result for P_0 .

In general, the deobfuscation problem effectively transforms into a program synthesis problem: a candidate program with the same input-output behavior as the target program is synthesized using a given component library. Therefore, the most important thing is to study how to reduce the program space and improve search efficiency.

3. Overview of Our Approach

AutoSimpler is an input-output-guided deobfuscation tool for binaries with data obfuscation. It has essentially put our idea into practice. At a high level, AutoSimpler contains three components: an obfuscation detector, a program synthesizer, and a search engine. It takes in a target program with obfuscated instructions and outputs a much easier program to understand. The obfuscation detector's goal is to find the obfuscated code snippets in the target program through a trained machine learning model. It goes through the following steps as highlighted in Figure 2.

Step I. Samples Generation. The main purpose of this step is to generate labeled samples for training the obfuscation detection model. All samples are obfuscated by two open-source obfuscation tools of Tigress and OLLVM. The details are in Section 4.1.1.

Step II. Feature Selection. In this step, we employ Entropy of basic blocks and N-gram on instructions as

the selected features to describe the behavior of data obfuscation. The details are in Section 4.1.2.

Step III. Classification. The selected features are encoded into vectors and used as input to the trained machine learning model. It performs classification with Support Vector Machine (SVM) [25] and Logistic Regression with Gradient Descent (LR-GD) [26]. The details are in Section 4.1.3.

Step IV. Predication. This is the detecting phase. It takes in a given target code snippet in the form of assembly code. The trained model predicts if the code snippet has been obfuscated.

Program Synthesizer. The program synthesizer aims to synthesize candidate programs with given components by observing the input-output behavior of the target program. It goes through the following steps as highlighted in Figure 2. The details of the program synthesis algorithm are in Section 4.2.1. It takes in a code snippet with data obfuscation and outputs a simplified mathematical expression. For example, when the input is as shown in Figure 1(a), the output is $a + b + c + d$.

Search Engine. The search space is another key dimension in program synthesis [20]. In this paper, we select Nested Monte Carlo Search [24] as the search engine to promote the speed of program synthesis. It addresses the problem of guiding the search toward better states when there is no available heuristic. In the framework of AutoSimpler, Nested Monte Carlo Search plays two roles. One is observing and understanding the relationship between input and output of the target program. The other is searching for a candidate program that has similar input-output behavior to the target program. The reward obtained in each iteration will change the priority of selecting an operator from the given components, thereby avoiding the generation of repetitive and meaningless searching. The details are in Section 4.3.

4. Implementation Details

In this section, we start with introducing the classifier model for obfuscated instructions detection in Section 4.1. Then, we depict the program synthesis algorithm and describe how to constraint the program space by the context-free grammar in Section 4.2. Finally, we introduce how to search the program space by Nested Monte Carlo Search in Section 4.3.

4.1. Obfuscated Instructions Detector. The input-output-guided program synthesis is limited to the automated synthesis of loop-free programs. So we divide the target program into several basic blocks without a loop. These basic blocks will be processed as an input to AutoSimpler.

To identify input and output variables, we employ a binary classifier to find obfuscated instructions. Even though obfuscation can hide semantics very well, there can still be some hints left. Existing obfuscation detection work has high

FIGURE 2: Overview of AutoSimpler. It consists of obfuscation detector, program synthesizer, and search engine. Obfuscation detector uses a machine learning model to locate the obfuscated code snippets in the target program. The workflow of the program synthesizer is shown as the following steps: (1) Collect a set of input-output examples for the target program. (2) Generate grammar constraints according to the given component and use the grammar to guide the generation of candidate programs. (3) Select an input value from a set of input-output examples and calculate the output value of the candidate program on this input. (4) Compare whether the candidate program and the target program have the same output on the same input. The search engine is a heuristic search algorithm of Nested Monte Carlo Search, which helps to improve the accuracy and efficiency of program synthesis.

accuracy [13, 33–37]. Thus, we are motivated to employ a classifier to detect which basic block is obfuscated.

4.1.1. Samples Generation. To construct samples with obfuscated labels, we propose a code obfuscated sample generator. It takes in a source file written in C language and produces an obfuscated assembly file with some obfuscated labels. First, it checks whether the source file can be labeled. It is considered to label with obfuscation only when the function in the file meets the following three conditions: (1) The function contains obfuscation points. Obfuscation points are defined here as arithmetic and logical operators (such as $+$, $-$, $*$, $\wedge$, $\sim$, $|$). (2) The input and output types of the function are both *int*. It means that there is no need to spend extra time and effort to identify the input and output variables when collecting the input-output examples during the deobfuscation phase. (3) There are no loops inside the function. That is because the oracle-guided program synthesis used here cannot handle loops. Second, the functions satisfying the above three conditions are obfuscated by Tigress and OLLVM, and the function name is marked as obfuscated. Third, if the code transformation is successful, the obfuscated code will be compiled into assembly code by *gcc*. Fourth, take the function as a basic block, divide the obfuscated assembly file into basic blocks, and put each function into a *.txt file.

To establish the ground truth about the obfuscated assembly codes, we use two open-source code obfuscation tools, OLLVM [16] and Tigress [14]. Both of them support data obfuscation, such as OLLVM with the option of *-sub* and Tigress with the option of *-EncodeArithmetic*. OLLVM is

an open-source code obfuscator based on the LLVM framework [15, 16, 38]. In theory, OLLVM supports any language and machine architecture in the world. Tigress is a diversifying virtualizer/obfuscator for language C. It supports many novel defenses against both static and dynamic reverse engineering and devirtualization attacks. The reason why we chose these two open-source obfuscators is that, compared with commercial obfuscators [39–42], we can more easily understand their obfuscation principles and can flexibly label obfuscated codes as needed.

When building the code obfuscation sample generator, we implement an algorithm for detecting obfuscation points. Then, we use the command line provided by OLLVM and Tigress to obfuscate the obfuscation points. It is worth mentioning that obfuscated files are still in language C. So, to obtain obfuscated assembly samples, we use the *gcc* compiler to produce the *.s file.

4.1.2. Feature Selection. We propose the following two types of features: Entropy [43] and N-gram [44]. Both of them have been proven to have outstanding performance in binary code machine learning [35–37].

Entropy of Basic Blocks. The Entropy represents the statistical characteristics of character frequencies. Data obfuscation involves a lot of arithmetic and logical operations. That will lead to a significant increase in the number of opcodes in instructions, such as ADD, SUB, and XOR, which may influence the Entropy substantially in most cases. The Entropy is calculated as follows:

$$H = - \sum P_i \log_2^{P_i}, \quad (1)$$

where P_i represents the frequency of the i^{th} character. *N-Gram on Instructions*. N-gram is a contiguous sequence of N items extracted from the given samples. For program binary analysis, N-gram-based features are defined as a sequential pattern where an individual sample can be identified as binary instructions. In this paper, AutoSimpler generates feature vectors by counting the frequency of opcodes and using N-grams with the top N of opcodes' highest frequency. Suppose a piece of code is composed of N instructions. Any instruction I_j depends on the influence of the first instruction I_1 to every instruction I_{j-1} before it. The N-gram is calculated as follows:

$$p(P) = p(I_1)p(I_2|I_1) \dots p(I_N|I_{N-1} \dots I_2I_1). \quad (2)$$

The probability of each item is expressed by Maximum Likelihood Estimation (MLE), which is frequency statistics. So, there is

$$p(I_N|I_{N-1} \dots I_2I_1) = \frac{C(I_1I_2 \dots I_N)}{C(I_1I_2 \dots I_{N-1})}. \quad (3)$$

4.1.3. Classification. Machine learning has been widely used in the field of binary code analysis. Some previous works related to this area show that Support Vector Machine (SVM) [25] and Logistic Regression with Gradient Descent (LR-GD) [26] are more efficient. Thus, our choice will be SVM and Gradient Descent (LR-GD) to perform the classification.

- (i) *Support Vector Machine (SVM)*. SVM is a supervised learning algorithm suitable for solving classification problems. Its basic model is to find the best separation hyperplane in the feature space to maximize the interval between positive and negative samples on the training set.
- (ii) *Logistic Regression with Gradient Descent (LR-GD)*. Logistic Regression is a machine learning method used to solve two classifications (0 or 1) problems and estimate the probability of something. Logistic Regression introduces the Sigmoid function into the linear regression model so that the output value of the uncertainty range of linear regression can be mapped to the range of (0, 1).

As supervised methods, both SVM and LR-GD rely on two phases. In the training phase, the algorithm obtains knowledge about the class by checking the training set describing the class. The classification mechanism examines the test set and associates its members with available classes in the testing phase. Therefore, the data needs to be labeled before training the classification model as what we did in Section 4.1.1.

4.2. Program Synthesizer

4.2.1. Program Synthesis Algorithm. Algorithm 1 describes the deobfuscation algorithm based on program synthesis, and the execution process is described as follows:

Step 1. Input-Output Sampling. The goal of this step is to collect input-output examples of the target program. The input obfuscated code snippet is regarded as a black box. Give the input variable a random value and obtain a corresponding output value by simulating the target code fragment's execution. The number of input-output samples depends on the specific situation.

Step 2. Grammar Constraints. Generate the corresponding grammar based on the given component. The purpose of generating grammar is to reduce the program space dimension and reduce the amount of calculation. Here, we choose context-free grammar as grammar constraints. The details are in Section 4.2.2.

Step 3. Output the candidate programs. Based on the constraints of context-free grammar, AutoSimpler uses Nested Monte Carlo Search to select and generate candidate programs.

Step 4. Similarity Evaluation. Compare the candidate program's input-output behavior with that of the target program. If they have the same output on the same input, the candidate program is the target we are looking for. Otherwise, it returns failure. When measuring the similarity of the input-output behaviors between the candidate program and the target program, we choose the Hamming distance, which is also used by Syntia [22]. That is because data obfuscation includes not only arithmetic operations but also logic operations. The Hamming distance states how many bits two values differ so that it can address these operations.

4.2.2. Grammar Constraints. The grammar constraints in AutoSimpler try to reduce the search space of synthetic programs. It can be expressed in many forms, such as regular, context-free grammar, and logical representations. Here, we choose context-free grammar because it is more flexible in describing expressions. A context-free grammar is a collection of context-free phrase structure rules. It contains four elements: a group of nonterminal symbols, a group of terminal symbols, a group of productions, and specifying a nonterminal symbol as the start symbol.

Specifically, context-free grammar constructs expressions with semantic information based on the provided components. In general, the component library's size determines the complexity of the context-free grammar. Therefore, providing an appropriate component library is one of the most critical factors to ensure program synthesis's efficiency and effectiveness.

Suppose the target program we want to synthesize has two input parameters, such as $\vec{I}_i \rightarrow (a_i, b_i)$, and the given components include $\{*, +\}$. Therefore, there are four productions for this grammar as follows:

$$\begin{aligned} T &\rightarrow T * T \\ T &\rightarrow T + T \\ T &\rightarrow a \\ T &\rightarrow b. \end{aligned} \quad (4)$$

```

Input:  $C$ 
Output:  $P'$ 
(1)  $E = \{\langle \vec{T}, tO \rangle\}$ 
(2) while 1 do
(3) grammar = produce ( $C$ )
(4)  $L = \text{Nested\_MCS}(\text{grammar})$ 
(5) if  $L = \Phi$  then
(6) return Failed
(7) end if
(8) if Similar( $(\vec{i}, o), L$ ) then
(9)  $P' = L$ 
(10) return  $P'$ 
(11) else
(12) return Failed
(13) end if
(14) end while

```

ALGORITHM 1: Iterative synthesis algorithm.

According to the production derivation, there is more than one terminal string that satisfies the grammar, such as $T_1 \rightarrow a + a * b$, $T_2 \rightarrow a * b + b$, $T_3 \rightarrow a * a * b + b$, $T_4 \rightarrow a * a * a$, and $T_5 \rightarrow a * a + b$. All generated terminal strings can be called candidate programs. Obviously, if there is no suitable search algorithm, it is necessary to exhaust all the terminal strings that satisfy the grammar, which is a resource-consuming operation. Therefore, this paper introduces the Nested Monte Carlo Search to synthesize program with the guidance of context-free grammar.

4.3. Nested Monte Carlo Search Algorithm. Nested Monte Carlo Search [24] is an improvement of Monte Carlo Tree Search [45]. Monte Carlo Tree Search is a stochastic algorithm that directs the search toward an optimal decision in a given domain, with four steps in each search iteration: selection, expansion, simulation, and backpropagation. When selecting the following code to visit, MCTS uses the Upper Confidence bounds for Trees (UCT) [45] algorithm to weigh the exploration and exploitation problem. Nevertheless, the randomness and blindness of MCTS cause that it cannot always find the best result. Especially when the left and the right nodes have the equal UCT reward, MCTS will stochastically pick one of them for expansion with no more reasonable suggestions.

Nested Monte Carlo Search addresses the problem of guiding the search toward better states when there is no available heuristic. Nested Monte Carlo Search combines nested calls with randomness in the ployout and memorization of the best sequence of moves. In particular, Nested Monte Carlo Search uses random move instead of a heuristic at the base level but employs nested rollouts combined with a heuristic to choose the next move at the levels. It tries to search each possible move only once before each lower-level search. Besides, Nested Monte Carlo Search memorizes the best sequence found so far when the randomized search gives worse results than the best sequence.

Algorithm 2 and Algorithm 3 demonstrate how Nested Monte Carlo Search works. Algorithm 2 is essentially a

basic Monte Carlo Tree Search. It initializes the root node state, which is the left nonterminal of the production in the context-free grammar. In each iteration, MCTS repeats the operations of node selection, simulation, and backpropagation. First, it takes the node with the largest reward as the next node to be expanded. Second, it simulates the program's synthesis state when the current node is selected and evaluates it through similarity evaluation. If the currently synthesized candidate program's input-output behavior is consistent with the target program, then jump out of this iteration, and the search ends. If they are inconsistent, continue to the step of backpropagation. Third, backpropagation updates the reward value of each node in the simulation state. If the program still does not terminate, iterations increase by 1, and the process continues.

Algorithm 3 is the key part of the Nested Monte Carlo Search. Nested Monte Carlo Search always chooses the lower-level search's highest reward when trying to get a reward by simulation. It tries all possible moves at each iteration and their nested lower level to find the best reward. If the current reward is less than the best reward, Nested Monte Carlo Search updates the best sequence and a reward with the newly found one to prevent missing the best reward of the best sequence.

5. Experimental Setup

In the experiment, we want to answer the following questions related to our approach:

- (i) **Study 1:** What is the accuracy of obfuscation detector in AutoSimpler? Which machine learning classification method performs better?
- (ii) **Study 2:** What is the performance of program synthesizer in AutoSimpler? Including its success rate and execution time, the most critical question is how to determine that the deobfuscated result is correct and simplified?

Input: grammar, target, IO_inputs, uct_scalar, state, max_iter, max_time = 0, nested_level
Output: P'

```

(1) self.root.state = state
(2) self.max_iter = max_iter
(3) start_time = time ()
(4) required_time = 0
(5) current_iter = 0
(6) while current_iter < max_iter and not self.finished do
(7)   if max_time and required_time_max_time then
(8)     break
(9)   end if
(10)  self.current_iter = current_iter
(11)  node = selection (self.root)
(12)  reward = simu_nested (node, nested_level)
(13)  back_propagation (node, reward)
(14)  current_iter += 1
(15)  current_time = time ()
(16)  required_time = current_time - start_time
(17) end while

```

ALGORITHM 2: Nested_MCS.

Input: self.node, netsted_level
Output: best_reward

```

(1) best_reward = -1
(2) while not self.finished do
(3)   if level = 1 then
(4)     expr = generate (node.state.expr)
(5)     reward = self.score (expr)
(6)   else
(7)     reward = simu_nested (self.node, netsted_level-1)
(8)   end if
(9)   if reward > best_reward then
(10)    best_reward = reward
(11)    best_sequence = seq.aftermove
(12)   end if
(13)   best_move = best_sequence
(14) end while
(15) return best_reward

```

ALGORITHM 3: simu_nested.

- (iii) **Study 3:** How does our approach compare to Syntia? The biggest difference is that the search algorithm we use is heuristic, while Syntia's is stochastic.
- (iv) **Study 4:** How do the input and output examples of different scales affect the performance of AutoSimpler?

5.1. Evaluation Metrics

5.1.1. Metrics for Obfuscation Detection. We employ the widely used metrics in the field of machine learning of *Accuracy* and *F1-score* to evaluate the accuracy of obfuscation detector. Here is a brief introduction:

$$\text{Accuracy} = \frac{TP + TN}{TP + FN + FP + TN}, \quad (5)$$

$$F1 = 2 \cdot \frac{TPR \cdot FPR}{TPR + FPR}, \quad (6)$$

where TP is the number of samples with obfuscation detected correctly. FP is the number of samples with false obfuscation detected. FN is the number of samples with true obfuscation undetected, and TN is the number of samples with no obfuscation undetected. TPR is calculated as $TP/(TP + FN)$, and FPR is calculated as $FP/(FP + TN)$.

We also use *Detection_Time* to evaluate the effectiveness of obfuscation detector, especially for the testing time.

5.1.2. Metrics for Deobfuscation Result. Deobfuscation aims to remove the obfuscation effect in the obfuscated program, making it easier to understand and readable. To the best of our knowledge, there are no predefined metrics to measure whether the deobfuscation result is easier to understand. David et al. [23] provide a solution for evaluating the size reduction factor of the obfuscated expression against the synthesized one. In addition, we also use the user study in Section 6.6 to evaluate the deobfuscation results.

5.1.3. Metrics for Performance of AutoSimpler. We use two metrics of *Success_Rate* and *Excution_Time* to evaluate the performance of AutoSimpler. Here is a brief introduction:

$$\text{Success\_Rate} = \frac{N_{\text{success}}}{N_{\text{success}} + N_{\text{failure}}}, \quad (7)$$

$$\text{Excution\_Time} = T_{\text{sampling}} + T_{\text{searching}}, \quad (8)$$

where N_{success} represents the number of times that AutoSimpler successfully synthesized the target program's deobfuscation results. Success means that AutoSimpler produced a program with the same input-output behavior of the target program but is easier to understand than the target program. N_{failure} is the number of times AutoSimpler could not find the correct deobfuscation result. There are two situations. One is that no program is generated until the end of AutoSimpler execution. The other is that AutoSimpler generates a candidate program consistent with the target program's input-output behavior but is not easier to understand than the original program. T_{sampling} is time for generating input/output examples. $T_{\text{searching}}$ represents the time for AutoSimpler to find a candidate program with the same input-output behavior as the target program.

5.2. Datasets. In the experiment, we use two kinds of data sources for verifying the performance of AutoSimpler. One is the source code coming from *gcc-7.4.0*. The choice of experimental samples should follow the principle of universality, which means that the experimental samples should exist in the real world. *gcc* is the most widely used compiler and contains a lot of source code in language C. The other is generated sample, which is produced by a mathematical expression generator. Both of them are considered as original samples and then implemented by the method described in Section 4.1.1 to produce obfuscated samples.

Finally, there are three data sets used in the experiment. Dataset1 is to train the obfuscation detection model. Dataset2 and Dataset3 are used to test the accuracy of deobfuscation. The details of each dataset are described as follows:

Dataset1: We use a dataset of OBFEEY [13], which contains over 277,000 obfuscated samples with different individual obfuscation schemes. The source codes of OBFEEY's datasets come from the real world like *GNU Toolkit* and *gcc-7.4.0*. All samples in this database are obfuscated by tools of OLLVM [16] and Tigress [14]. We manually selected 1000 obfuscated

samples (750 from the data obfuscated samples and 250 from the other types of obfuscated samples). All samples are finally divided into 3745 basic blocks, of which 1209 are obfuscated snippets, and the rest are original snippets.

Dataset2: The source code comes from a mathematical expression generator developed by us. It generates 750 arithmetic expressions with 2 to 5 input parameters and six common operations such as addition, subtraction, multiplication, AND, OR, and NOT. All the 750 samples are obfuscated by OLLVM [16] with the option of *-sub* and Tigress [14] with the option of *-EncodeArithmetic*. Therefore, there are a total of 1500 data obfuscated samples in this data set.

Dataset3: The source code comes from *gcc-7.4.0*. We select 300 samples that meet the three conditions mentioned in Section 4.1.1. All of them are obfuscated by OLLVM [16] with the option of *-sub* and Tigress [14] with the option of *-EncodeArithmetic*.

5.3. Implementation and Evaluation Platforms. Our prototype system, with the name of AutoSimpler, is implemented using Python v.3.7. Specifically, the AutoSimpler draws on the code framework of Syntia [22], utilizing the disassembler framework Capstone [46] and a CPU emulator framework Unicorn [47].

The experiments are performed on a notebook computer running the Windows 10 operating system with two 64-bit 2.9 GHz Intel (R) Core (TM) i7-3520 CPUs.

6. Experimental Results

In this section, we first evaluate the accuracy of obfuscation detection by machine learning. Then, we show the accuracy of our approach on the samples obfuscated with OLLVM and Tigress. Next, we compare our approach with Syntia, demonstrating that our approach is significantly better. Fourth, we study the impact of input-output examples on the sampling time, searching time, and iterations. Finally, we use a user study to verify the understandability of the deobfuscation results.

6.1. Evaluation of Obfuscation Detection. Accurate localization of obfuscated fragments is a prerequisite for AutoSimpler. For obfuscation detection, we apply two different classifiers based on two types of features mentioned in Section 4.1.

To train the classification model of obfuscation detector, we use Dataset1 as the training set. We use 10-fold cross-validation to select the best model during the training phase. It should be noted that each function in all examples is regarded as a basic block and saved in the TXT file. And the file name is taken as an obfuscated label.

Results. We apply classifiers of Support Vector Machine (SVM) [25] and logistic regression with gradient descent (LR-GD) [26] on the testing set. The results are shown in

Table 2. When we apply SVM as a classifier, the *Accuracy* of our approach is 96.82% with *F1-score* of 100%. Compared with SVM, the classifier of LR-GD performs better. When the classifier is LR-GD, the *Accuracy* of our approach can achieve 99.29% with *F1-score* of 100%. Based on the results, we do a further investigation that SVM and LR-GD have different loss functions. SVM only considers the most relevant few points to learn the classifier. Logistic regression reduces the weight of the points far away from the ground truth through nonlinear mapping and relatively increases the weight of the points most relevant to the classification. Therefore, the loss function of SVM directly ignores those instructions that involve calculations but are not obvious.

The experimental results reported in the table highlight that the performance of obfuscation detector in our approach is good. Particularly, the accuracy is as high as 99.29%. In addition, the execution time of our obfuscation detection is extremely fast, taking only 29 seconds for 1000 samples.

6.2. Evaluation on Deobfuscation Result. To the best of our knowledge, there are no predefined metrics to measure whether the deobfuscation result is easier to understand. Syntia [22] uses the number of expression layers to evaluate the deobfuscation results. For example, Figure 1(b) is a code snippet with data obfuscation, which has 87 expression layers. Its deobfuscation result is as shown in Figure 3(a) $a + b + c + d$, whose expression layer is 7. This evaluation metric is more suitable for arithmetic expressions rather than real-world cases since the real-world cases contain other characters in addition to arithmetic expressions. David et al. [23] provide a solution for evaluating the size reduction factor of the obfuscated expression against the synthesized one.

It is considered that our experimental samples include source code derived from *gcc* (in Dataset3) and constructed arithmetic expressions (in Dataset2). Therefore, we use both size deduction and expression layers to evaluate the deobfuscation result in the experiment.

In this experiment, we selected 300 samples from Dataset2 and Dataset3, respectively, and counted the expression layer and size of their original, obfuscated, and deobfuscated program. The number of expression layers for each sample is 4–10, and the size ranges from 1 kb to 11 kb. After obfuscation, the number of expression layers is 6–1361, and the size ranges from 13 kb to 3569 kb. It should be emphasized that when we count the expression layer and size, we prefer the obfuscated code snippets rather than the entire program.

Table 3 shows the statistical results. Compared to obfuscated code, the number of expression layers of synthesized code is reduced to 14.05% on average, and the size of the code is reduced to 5.67% on average. To guide AutoSimpler in automatically judging whether the generated result is correct, we also give a three-quarter statistical result. This value can be set as the threshold for the correctness of the deobfuscation result in the system. Compared with the original code, the average value of the code simplified by

TABLE 2: The accuracy of obfuscation detection.

Classifier	Accuracy (%)	F1-score (%)	Detection_Time (s)
SVM	96.82	100	37
LR-GD	99.29	100	29

AutoSimpler is slightly larger. That is because AutoSimpler cannot always get the most simplified code. For example, the obfuscated expression in Figure 1(b) is sometimes simplified to $(a \& b) + (a|b) + c + d$, as shown in Figure 3(b), but its most simplified expression should be $a + b + c + d$.

6.3. Evaluation of Performance. In this experiment, to avoid bias, we use 1000 samples from Dataset2 and Dataset3. All the samples are obfuscated by OLLVM [16] with the option of `-sub` and Tigress [14] with obfuscation of `-EncodeArithmetic`. The number of inputs for these samples varies from 2 to 5. Besides, the number of input-output examples we selected is 50. Through many experiments, we found that when the number of input-output examples is 50, AutoSimpler has the highest efficiency and accuracy (The details are in Section 6.5). We execute each sample 10 times and record whether the deobfuscation is successful. The statistical results are shown in Table 4.

It can be seen from Table 4 that the accuracy of AutoSimpler is above 90.34%, and it takes about 23 seconds to process an obfuscated program on average, requiring about 14 iterations. In addition, AutoSimpler has a higher success rate of deobfuscation when processing arithmetic expressions constructed by ourselves, with an average of 94.16% and an average execution time of 21.97 seconds. That is because its arithmetic expressions are more standard than the sample format randomly selected by *gcc* source code. It is worth mentioning that, compared with OLLVM, the samples obfuscated by Tigress have a slightly lower success rate and execution speed since there are different data flow obfuscation strategies between them. We use Tigress and OLLVM to perform data flow obfuscation on the expression $x + y$, respectively. Figure 4 shows the results of the obfuscation. Tigress uses a combination of arithmetic and logical operations, while OLLVM only uses arithmetic operations. Obviously, the transformation of Tigress is more complicated, so the time spent in the deobfuscation process is longer.

6.4. Comparison with Syntia. It is difficult to get Syntia working, making it unclear how Syntia locates input and output variables on its dataset. Therefore, we replace the Nested Monte Carlo Search of AutoSimpler with MCTS and use the same 1000 samples from Dataset2 and Dataset3 as the testing cases. The number of input-output examples is still 50. In addition, we set the exploratory constant to 1.42 as recommended. Note that we perform ten times for each sample since Monte Carlo Tree Search is stochastic.

The experimental results are shown in Table 4. The best accuracy of MCTS is 90.12%. The worst is 85.34%, and both are lower than Nested Monte Carlo Search. That is because

$$\begin{array}{cc}
 a+b+c+d & (a\&b)+(a|b)+c+d \\
 \text{deobfuscation result} & \text{candidate expression} \\
 (a) & (b)
 \end{array}$$

FIGURE 3: Two examples of deobfuscation results simplified by AutoSimpler. Their original obfuscated code is shown in Figure 1. (a) Deobfuscation result. (b) Candidate expression.

TABLE 3: The size deduction of the synthesized expressions. Orig, Obf, and Synt are, respectively, original, obfuscated, and synthesized expressions.

Metrics	Obf/Orig	Synt/Obf		Synt/Orig	
	Mean	Mean	75%	Mean	75%
Level	10.826 3	0.140 5	0.136 8	1.209 3	1.053 1
Size	25.472 9	0.058 7	0.056 7	1.843 8	1.921 7

TABLE 4: Evaluation on the performance of AutoSimpler and Syntia.

	AutoSimpler				Syntia			
	Gen_T	gcc_T	Gen_O	gcc_O	Gen_T	gcc_T	Gen_O	gcc_O
Success_Rate	93.20%	90.34%	95.12%	92.78%	87.35%	84.63%	90.56%	88.33%
T_searching (s)	23.17	25.20	20.57	21.12	90.28	93.25	89.98	90.38
Mean_Iteration	14	15	13	13	8405	8690	7908	7118

$$\begin{array}{cc}
 x + y = x - \neg y - 1 & x + y = x - (-y) \\
 = (x \oplus y) + 2 \cdot (x \wedge y) & = -(-x + (-y)) \\
 = (x \vee y) + (x \wedge y) & r = \text{rand}(); a = x + r; a = a + y; a = a - r \\
 = 2 \cdot (x \vee y) - (x \oplus y) & r = \text{rand}(); a = x - r; a = a + y; a = a + r \\
 \\
 \text{Tigress} & \text{OLLVM} \\
 (a) & (b)
 \end{array}$$

FIGURE 4: Comparison of data flow obfuscation strategies between Tigress and OLLVM. (a) Tigress. (b) OLLVM.

MCTS is a stochastic search algorithm. It has no way to guarantee the best results every time. But Nested Monte Carlo Search overcomes this limitation by memorizing the best sequence to improve the search's success rate.

In addition, it takes about 90 seconds and 8000 iterations for MCTS to process a deobfuscation task. In contrast, Nested Monte Carlo Search is faster. It is due to the search strategy of combining nested calls and recording the best sequence. Although the Nested Monte Carlo Search takes more time per iteration than MCTS, it usually takes only a few iterations to find the answer quickly. To further analyze the difference between the two search algorithms, we will conduct a more in-depth study in the next section.

In general, compared with Syntia, the accuracy of AutoSimpler has increased by 5%, and the execution efficiency has increased by nearly 75%.

6.5. Performance with Different Input. The most important dimension in program synthesis from the user's perspective is the mechanism for describing intent. The input-output examples are one of the simplest and most useful forms of the specification [20]. In each iteration of the Nested Monte

Carlo Search, it is necessary to compare the synthesized candidate program in the current state with the input-output examples of the target program sampled in advance. The number of input-output examples determines the efficiency and accuracy of the Nested Monte Carlo Search. In other words, if the number of input-output examples provided is too small, the target program's intention is not clearly described. Conversely, suppose that there are too many input-output examples provided. In that case, a lot of time will be spent on comparing the input-output behavior between the candidate program and target program in each iteration, thereby increasing the additional overhead of the system. Therefore, this experiment focuses on the effect of different numbers of input-output examples on AutoSimpler and Syntia.

When discussing the relationship between the number of input-output examples and the execution time, we consider both the sampling and the searching times. This experiment set the number of input-output examples as 20, 50, 100, and 200, respectively. Then, we use AutoSimpler to deobfuscate the same target program. The deobfuscation process is executed ten times under each sampling number, and an average value is recorded.

Execution Time. After many experiments, it was found that when the sampling number is 20, 50, 100, and 200, the average sampling time is 4.15 seconds. Therefore, we can conclude that the difference in sampling time introduced by the number of input-output examples is negligible.

Table 5 lists the statistical results of searching time. Nested Monte Carlo Search compares the candidate program's input-output behavior with the target program in each iteration. Therefore, when the number of input-output examples is larger, the time spent in each round is longer. It can be seen from Table 5 that when the number of input-output examples is 20, the average time spent in each iteration is 0.87 seconds. When the input-output examples increase to 200, the time spent in each iteration also increases to 3.80 seconds.

As for MCTS, it can be seen that when the number of input-output examples is 20, the average searching time of each iteration is only 3.9 milliseconds. Compared with Nested Monte Carlo Search, MCTS spends nearly 200 times less time on each iteration. That is because Nested Monte Carlo Search tries all possible movements of the lower layer and compares each node's reward with the best reward. Therefore, it takes longer in each iteration.

Iterations on NMCS. To investigate how many iterations are required for AutoSimpler to find the deobfuscated results successfully, we also do further research and analyze the experimental results. We also drew the box plot of iterations when the number of the input parameters is 2 and 3 parameters according to the experimental results, as shown in Figure 5. Figure 5(a) is a box plot when the number of the input parameters is 2. It can be seen from the figure that when the number of input-output examples is different, the number of iterations required to find the deobfuscation result successfully is different. The minimum number of iterations is 1, and the maximum number of iterations is 84. Both of them appear when the number of input-output examples is 20. It can be seen from the figure that when the number of input parameters is 2, regardless of the number of input-output examples, the median number of iterations is approximately 20. Therefore, the number of input-output examples does not affect iterations.

Figure 5(b) is the iterations required for AutoSimpler to deobfuscate with 3 parameters successfully. The median number of iterations is approximately 9. Compared with Figure 5(a), when the number of parameters is 2, the median of the number of iterations is 20. The fewer the number of input parameters is, the more the iterations are required. It seems to be against common sense. So, we conducted an in-depth exploration. In the case of a fixed number of operators (N) in the data flow component, when the input parameter is 2 (such as $\langle(i_1, i_2), o\rangle$), the operators in the component set are combined in pairs. If each operator is not used repeatedly, the number of possible combinations of operations is C_{N+1}^2 . When there are 3 input parameters of an arithmetic

expression (such as $\langle(i_1, i_2, i_3), o\rangle$), the number of possible operation combinations is C_{N+1}^3 . When the input parameters are 2, the program's search space is larger, so more iterations are required.

Iterations on MCTS. We also draw the box plot of iterations when the number of operations is 2 and 3 parameters according to the experimental results, as shown in Figure 6. It can be seen from the figure that the minimum number of iterations is 21, and the maximum number of iterations is 12 631. Both of them appear when the number of input-output examples is 20. When the number of input parameters is 2, the average median of the iterations is 9478. When the number of input parameters is 3, the average median of iterations is 3372. Compared with Nested Monte Carlo Search, the number of iterations for processing a task has increased by more than 3000 times. The reason is that Nested Monte Carlo Search memorizes the move associated with the best reward of the lower level. It can effectively control the number of iterations.

The conclusions of this experiment are as follows. First, the difference in sampling time introduced by the number of input-output examples is negligible. The sampling time is about 4 seconds. Second, the fewer the number of input parameters is, the more the iterations are required. Third, NMCS is more efficient than MCTS. Taking an example that the number of input-output examples is 20, the searching time spent on each iteration by MCTS is nearly 200 times less than that of NMCS. Still, the number of iterations it takes for MCTS to complete a deobfuscation task is almost 3000 times higher than that of NMCS. In general, NMCS is more efficient in execution.

6.6. User Study. To verify the understandability of the deobfuscation results, we organized another user study with the same five postgraduate students mentioned in Section 2.1. This time, each participant is randomly given 100 deobfuscation results simplified by AutoSimpler. They are still given 12 hours to accomplish a task: try to understand the deobfuscation results. When they have completed all tasks, they can get the original program of these deobfuscation results. Then, these participants judge whether the deobfuscation result has the same semantics as the original sample.

Table 6 shows an understandability result in our user study. Everyone completed the task ahead of schedule, and the fastest participant only took 88 minutes. There are two participants who believed that all 100 deobfuscation results have the same semantics as the original program, and the deobfuscation results are easy to understand. The results of the other three people are 98, 99, and 96. It means that there are seven deobfuscation results not meeting expectations. To find the real reason, we do a further investigation. They have indeed been simplified from the perspective of size

TABLE 5: Searching time on each iteration (s).

	The number of input-output examples			
	20	50	100	200
AutoSimpler	0.867 4	1.517 2	2.434 8	3.799 4
Syntia	0.003 9	0.006 5	0.011 7	0.020 3

FIGURE 5: Iterations with Nested Monte Carlo Search: (a) when the number of arithmetic expression parameters is 2; (b) when the number of arithmetic expression parameters is 3.

FIGURE 6: Iterations with Monte Carlo Tree Search: (a) when the number of arithmetic expression parameters is 2; (b) when the number of arithmetic expression parameters is 3.

TABLE 6: Understandability of deobfuscation result by user study.

Participant	Understandability	Time (m)
Stu1	100/100	88
Stu2	98/100	136
Stu3	99/100	125
Stu4	100/100	97
Stu5	96/100	104

deduction, but they are not easier to understand than the original program because they also contain some logical operators.

The result of this user study verifies that AutoSimpler's deobfuscation results are indeed simplified a lot.

7. Limitations

There remain some limitations. First, AutoSimpler can only synthesize loop-free programs. In other words, our approach is restricted to synthesizing only straight-line programs. This limitation is also a common problem for all program synthesis techniques. In the future, it would need to be extended by synthesizing the program with a loop.

Second, although Nested Monte Carlo Search has shown powerful ability in program space search, there will still be cases of search failure. Therefore, in the future, we will consider changing to another search technique of MCTSnets [48], which is a neural version of the MCTS algorithm. MCTSnets aims to maintain the desirable properties of MCTS while allowing some flexibility to improve the choice of nodes to expand, the statistics to store in memory, and how they are propagated, all using gradient-based learning.

Third, due to the addition of the three restrictions, the data scale and actual cases in the experiment are very small. In the future, more real-world cases will be considered to evaluate the performance of the approach.

Fourth, the experiments only focus on two open-source obfuscation tools, which are still lacking in convincing. Larger experiments are planned to expand to other commercial obfuscators.

8. Related Work

Obfuscation Detection. Zhao et al. [13] propose an obfuscation detection method using deep neural networks to learn semantic information of the disassembled binary to predict the program's Obfuscation Scheme. But they do not discuss locating the obfuscated code snippet. Tofighi et al. [33] present a fine-grained detection framework of obfuscation transformations and constructions. Compared with this work, the same thing is that both of us consider the locating of obfuscated code snippets. The difference between us lies in the way to label obfuscation and machine learning techniques. First, Tofighi et al. use the Miasm2 intermediate language as the raw data, and we use the .s file generated after GCC compilation. Second, they use Extra Tree and Random Forest as the classification model, and we use Logistic Regression with Gradient Descent (LR-GD).

Deobfuscation. Yadegari et al. [17] propose a generic approach to the deobfuscation of executable code, which works on the intuition that the semantics of a program can be considered as a mapping from input values to output values. Our work is also working on this underlying intuition. The biggest difference between the two is the technique of simplifying redundant instructions. Yadegari et al. use taint propagation to track the flow of values from the program's

inputs to its outputs and keep only those codes that can affect the input and output values. But we directly use program synthesis to synthesize a deobfuscation result with the same behavior as the target program according to the input-output examples' guidance.

Coogan et al. [12] identify instructions that interact with the system and then use various analyses to determine which instructions affect the interaction. They use value-based dependence analysis and control flow analysis to discard the uninteresting instructions. Sharif et al. [49] propose an approach for automatic reverse engineering of malware emulators. They extract the syntax and semantics of the obfuscated bytecode instructions by dynamically analyzing a decode-dispatch-based emulator. Kruegel et al. [50] describe static analysis to disassemble the obfuscated Intel x86 binaries correctly. They present general control flow-based and statistical techniques to deal with hard to disassemble binaries.

These existing works are classified into the same category: deobfuscation based on reverse engineering. First of all, they require a lot of expert experience and domain knowledge. Secondly, reverse engineering is a tedious, lengthy, and subjective process, and the analysis results will vary depending on the analyst's ability. Finally, this category is not scalable, even if the same obfuscation algorithm has different characteristics on different objects.

Program Synthesis. Gulwani [20] describes three key dimensions of program synthesis: user intent, search space, and search technique. He also gives a brief description of various techniques for each dimension. Jha et al. [21] present an approach to automatic synthesis of a loop-free program with the guidance of input-output examples. We are also based on the guidance of input-output synthesis. The difference is that they use Satisfiability Modulo Theories (SMT) solvers to constraint the search space. We use context-free grammar.

The work philosophically closest to ours is that by Blazytko et al. [22], who present a tool named Syntia with using the program synthesis guided by input-output examples to synthesize obfuscated code. While their goals are similar to ours, the technical details are different. The biggest difference between the two is the search technique of program synthesis. Syntia employs Monte Carlo Tree Search to guide the program synthesis. Although Monte Carlo Tree Search has been widely used in various fields, it still has some significant drawbacks. The most important thing is that it is stochastic, leading to a low program synthesis success rate. Our approach employs Nested Monte Carlo Search with a heuristic and memorization of the best sequence, which overcomes Monte Carlo Tree Search's limitations. Importantly, Blazytko has an assumption that they have known where the obfuscated codes are. By contrast, we break this assumption by using machine learning to locate obfuscated code.

Menguy et al. [51] propose a new concept of AI-based blackbox deobfuscation, which refers to the new area of using artificial intelligence to formalize program space, such as Syntia [22]. They take deobfuscation as an optimization

problem rather than a single-player game and promote the application of S-metaheuristics instead of MCTS. Menguy's approach does not involve the obfuscated code detection, which is another difference between us.

Another deobfuscation approach based on program synthesis was proposed by David et al. [23], who present a tool named QSynth leveraging both Dynamic Symbolic Execution and program synthesis to synthesize programs with data obfuscation. QSynth proposes a synthesis algorithm with an offline enumerate synthesis primitive guided by top-down breath-first search. When evaluating the comprehensibility of the deobfuscation results, both of us considered the size reduction. The biggest difference between us is that they do not discuss locating the obfuscated expression.

9. Conclusion

This paper describes an approach to deobfuscation of binaries based on program synthesis. On the one hand, it has a fine-grained obfuscation detection for locating obfuscated code snippets by machine learning with an accuracy of 99.29%. On the other hand, it combines the program synthesis and a heuristic search algorithm of Nested Monte Carlo Search. We have applied a prototype implementation of our ideas to data obfuscation in different tools, including OLLVM and Tigress. Our experimental results suggest that this approach is highly effective in locating and deobfuscating the binaries with data obfuscation, with an accuracy of at least 90.34%. Compared with the state-of-the-art deobfuscation technique, our approach's success rate has increased by 5%, and efficiency has increased by 75%. In general, experiments indicate that our approach is effective in simplifying the obfuscated binaries with data obfuscation.

Data Availability

As follow-up research is underway, the data will not be open for the time being.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This project was supported by the National Natural Science Foundation of China under Grant nos. 61972314 and 61872294 and the International Cooperation Project of Shaanxi Province under Grant nos. 2020KWZ-013, 2021KW-15, and 2021KW-04.

References

- [1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," *CiteSeer*, 1997.
- [2] K. Murad, S. N.-u.-H. Shirazi, Y. B. Zikria, and N. Ikram, "Evading virus detection using code obfuscation," in *Proceedings of the International Conference on Future Generation Information Technology*, pp. 394–401, Springer, Jeju Island, Korea, December 2010.
- [3] L. Glanz, S. Amann, M. Eichberg et al., "Codematch: obfuscation won't conceal your repackaged app," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 638–648, ACM, New York, NY, USA, August 2017.
- [4] S. Schulze and D. Meyer, "On the robustness of clone detection to code obfuscation," in *Proceedings of the 7th international workshop on software clones*, pp. 62–68, IEEE Press, San Francisco, CA, USA, May 2013.
- [5] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 389–400, ACM, Los Alamitos, CA, USA, November 2014.
- [6] A. Continella, Y. Fratantonio, M. Lindorfer et al., "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, pp. 1–15, San Diego, CA, USA, March 2017.
- [7] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 343–355, ACM, New York, NY, USA, October 2016.
- [8] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: logic-oriented opaque predicate detection in obfuscated binary code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 757–768, New York, NY, USA, October 2015.
- [9] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*, November 2005.
- [10] D. Xu, J. Ming, Y. Fu, and D. Wu, "Vmhunt: a verifiable approach to partially-virtualized binary code simplification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 442–458, New York, NY, USA, October 2018.
- [11] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: from virtualized code back to the original," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 372–392, Springer, Saclay, France, June 2018.
- [12] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 275–284, New York, NY, USA, October 2011.
- [13] Y. Zhao, Z. Tang, G. Ye et al., "Semantics-aware obfuscation scheme prediction for binary," *Computers & Security*, vol. 99, Article ID 102072, 2020.
- [14] C. S. Collberg, S. Martin, and J. Myers, *The Tigress Diversifying C Virtualizer*, <https://tigress.wtf/>, 2018.
- [15] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm: software protection for the masses," in *Proceedings of the 1st International Workshop on Software Protection SPRO '15*, pp. 3–9, IEEE Press, Florence, Italy, May 2015.
- [16] Github, *Obfuscator-Llvm*, <https://github.com/obfuscator-llvm/obfuscator/wiki>, 2017.
- [17] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable

- code,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pp. 674–691, IEEE, San Jose, CA, USA, May 2015.
- [18] B. Yadegari and S. Debray, “Bit-level taint analysis,” in *Proceedings of the IEEE International Working Conference on Source Code Analysis & Manipulation*, Victoria, BC, Canada, September 2014.
  - [19] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic super-optimization,” *ACM SIGARCH - Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2012.
  - [20] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, New York, NY, USA, July 2011.
  - [21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, “Oracle-guided component-based program synthesis,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, Cape Town, South Africa, May 2010.
  - [22] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, “Syntia: synthesizing the semantics of obfuscated code,” in *Proceedings of the 26th USENIX Conference on Security Symposium*, Vancouver, Canada, August 2017.
  - [23] R. David, L. Coniglio, and M. Ceccato, “Qsynth - a program synthesis based approach for binary code deobfuscation,” in *Proceedings of the Workshop on Binary Analysis Research*, San Diego, CA, USA, January 2020.
  - [24] T. Cazenave and N. Jouandeau, “Nested monte-carlo search,” in *Proceedings of the International Joint Conference on Artificial Intelligence*, Pasadena, California, USA, July 2009.
  - [25] T. Joachims, “Making large-scale svm learning practical,” *Technical Reports*, vol. 8, no. 3, pp. 499–526, 1998.
  - [26] T. Zhang, “Solving large scale linear prediction problems using stochastic gradient descent algorithms,” in *Proceedings of the Twenty-First International Conference on Machine Learning*, p. 116, Banff, Alberta, Canada, July 2004.
  - [27] 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.
  - [28] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, “Programming by demonstration using version space algebra,” *Machine Learning*, vol. 53, no. 1/2, pp. 111–156, 2003.
  - [29] S. Bansal and A. Aiken, “Automatic generation of peephole superoptimizers,” *Computer Architecture News*, vol. 34, no. 5, pp. 394–403, 2006.
  - [30] S. Bansal and A. Aiken, “Binary translation using peephole superoptimizers,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation OSDI’08*, pp. 177–192, California, USA, December 2008.
  - [31] Massalin and Henry, “Superoptimizer: a look at the shortest program,” *ACM SIGOPS - Operating Systems Review*, vol. 21, no. 4, pp. 122–126, 1987.
  - [32] R. Joshi, G. Nelson, and K. Randall, “Denali: a goal-directed superoptimizer,” *ACM SIGPLAN Notices*, vol. 37, no. 5, 2002.
  - [33] R. Tofighi-Shirazi, I. M. Asävoae, and P. Elbaz-Vincent, “Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning,” in *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering SSPREW’19*, New York, NY, USA, December 2019.
  - [34] Z. Li, Q. A. Chen, C. Xiong, Y. Chen, T. Zhu, and H. Yang, “Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security CCS ’19*, pp. 1831–1847, New York, NY, USA, November 2019.
  - [35] P. V. Shijo and A. Salim, “Integrated static and dynamic analysis for malware detection,” *Procedia Computer Science*, vol. 46, pp. 804–811, 2015.
  - [36] C. Liangboonprakong and O. Sornil, “Classification of malware families based on n-grams sequential pattern features,” in *Proceedings of the 2013 8th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, Melbourne, VIC, Australia, June 2013.
  - [37] H. Xue, S. Sun, G. Venkataramani, and T. Lan, “Machine learning-based analysis of program binaries: a comprehensive study,” *IEEE Access*, pp. 65889–65912, 2019.
  - [38] K. Lim, J. Jeong, S.-j. Cho et al., “An anti-reverse engineering technique using native code and obfuscator-llvm for android applications,” in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems RACS ’17*, pp. 217–221, New York, NY, USA, September 2017.
  - [39] O. Technologies, *Code Virtualizer: Total Obfuscation against Reverse Engineering*, <http://oreans.com/codevirtualizer.php>, 2017.
  - [40] V. Software, *Vmprotect Software protection*, <http://vmprosoft.com> year =, 2017.
  - [41] S. Technology, *Execryptor: Bulletproof Software protection*, <http://www.strongbit.com/execryptor.asp> year =, 2017.
  - [42] O. T. Technologies, *Advanced Windows Software protection System*, <https://www.oreans.com/themida.php>, 2017.
  - [43] A. L. Berger, “A maximum entropy approach to natural language processing,” *Computational Linguistics*, vol. 22, no. 1, pp. 39–71, 1996.
  - [44] P. Laskov, “Language models for detection of unknown attacks in network traffic,” *Journal in Computer Virology*, vol. 2, no. 4, pp. 243–256, 2007.
  - [45] C. B. Browne, E. Powley, D. Whitehouse et al., “A survey of Monte Carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.
  - [46] N. A. Quynh and C. capstone, <http://www.capstone-engine.org>, 2020.
  - [47] D. H. V. Nguyen Anh Quynh, *Unicorn*, <http://www.unicorn-engine.org>, 2020.
  - [48] A. Guez, T. Weber, I. Antonoglou et al., *Learning to Search with Mcts nets*, <https://arxiv.org/abs/1802.04697>, 2018.
  - [49] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, “Automatic reverse engineering of malware emulators,” *Proceedings of the IEEE Symposium on Security & Privacy*, vol. 5144, pp. 94–109, 2009.
  - [50] C. Krügel, W. K. Robertson, F. Valeur, and G. Vigna, “Static disassembly of obfuscated binaries,” in *Proceedings of the Ssym04: Proc Conference on Usenix Security Symposium*, San Diego, CA, USA, August 2004.
  - [51] G. Menguy, S. Bardin, R. Bonichon, and C. Lima, *Ai-based Blackbox Code Deobfuscation: Understand, Improve and Mitigate*, 2021.