Article:
Zhao, Y, Tang, Z, Ye, G et al. (4 more authors) (2020) Semantics-aware obfuscation scheme prediction for binary. Computers & Security, 99. 102072. p. 102072. ISSN 0167-4048

https://doi.org/10.1016/j.cose.2020.102072

Semantics-aware Obfuscation Scheme Prediction for Binary

Yujie Zhao[1], Zhanyong Tang[1], Guixin Ye[1], Dongxu Peng[1], Dingyi Fang[1], Xiaojiang Chen[1], Zheng Wang[2]

***Abstract*—By restoring the program into an easier understandable form, deobfuscation is an important technique for detecting and analyzing malicious software. To enable deobfuscation, one must know if the target program is obfuscated and what types of obfuscation schemes may be used. However, obtaining such information is challenging without having access to the original program source code.**

This paper presents a new way to estimate the obfuscation scheme of a compiled binary. It achieves this by using semantic information of the disassembled binary to predict if the program has been obfuscated and if so, what type of obfuscation scheme may be used. At the core of our approach is a set of deep neural networks that can effectively characterize and leverage the contextual information available in the assembly code. Our models are first trained offline, and the learned models can then be applied to new previously unseen obfuscated binaries. We evaluate our approach by applying it to a large dataset of over 277,000 obfuscated samples with different individual obfuscation schemes and their combinations. Experimental results show that our approach is highly effective in identifying the obfuscation scheme, with a prediction accuracy of at least 83% (up to 98%).

***Index Terms*—deobfuscation, reverse engineering, deep neural networks, disassembled binary analysis, semantic expression.**

I. Introduction

Code obfuscation has flourished in illegal areas, such as virus [42], repackaging [19], code cloning [45], [36], and privacy theft [15], etc. It changes the behavior characteristics of the program through code transformation to avoid detection by virus scanners or hinder reverse engineering by security analysts. Numerous studies have shown that the current state-of-the-art malicious code detection cannot cross the gap of code obfuscation [65], [36]. Therefore, deobfuscation issue has aroused widespread concern in academia, intending to recover the original code as much as possible by identifying and removing obfuscation codes [61].

Currently, most of the deobfuscation techniques focus on the topic, like how to automate the process of trying to simplify obfuscated code to restore the original code. All of them have a strong assumption that the obfuscation algorithm is known, so they can only work on a specific kind of obfuscation [6], [41], [52], [59], [44], [16]. For example, layout deobfuscation [6], opaque predicate deobfuscation [41], control flow flatten deobfuscation [52] and virtualization deobfuscation [59], [44], [16]. While important and useful, such approaches are of limited utility against obfuscations that are different from the specific ones they target.

1 are with the School of Computer Science and Technology, Northwest University, Xi'an, 710127, China.

2 is with the University of Leeds, United Kingdom.

In reality, we usually face with completely unknown malware in the form of executable code. This raises two closely related questions. The first question, from a deobfuscation perspective, is: *Whether the target program is obfuscated?* For example, if the existing deobfuscation techniques are applied to analyze the target program that does not contain obfuscated code, then they are not only useless but are worse than useless. Because it breaks the internal logic of the original program, causing the analyst to spend a lot of time and energy doing useless work. The second question, from a reverse engineering perspective, is: *What kinds of obfuscation algorithm are employed on the target program?* It is worth mentioning that new obfuscation algorithms [12], [29], [54], [13], [41], [56] and tools [48], [46], [50], [49], [25], [14] have emerged in an endless stream in recent years. Obviously, it is very ambitious for security analysts to understand the characteristics of each code obfuscation and quickly analyze which obfuscation algorithms the target program adopts. However, if the obfuscation algorithm employ on the target program is identified through automated means, it will greatly reduce the difficulty of reverse analysis and the requirements for security analysts. Thus, regarding techniques for identifying obfuscated code and what kinds of obfuscation employed can lead to better deobfuscation schemes and concomitant improvements in reverse engineering.

Compared with the automated process of simplifying the obfuscated code to restore the original code, addressing the raised two questions usually requires much more time and effort. That is because it requires a wealth of knowledge and experience from security experts, including not only all kinds of code obfuscation, but also reverse analysis. But even for experts, it is a subjective, tedious, and sometimes error-prone task due to the complexity of the problem. Moreover, different experts have different levels of technology and experience, which means that the quality of the selected obfuscation features and the effectiveness of the resulting detection system varies with the person who defines them. In particular, it should be emphasized that multiple obfuscation schemes are usually a combination of several types of obfuscation, which have higher requirements for experts. Therefore, it is essential to liberate experts from the tedious obfuscation identification work through technical means to improve efficiency and accuracy of obfuscation scheme prediction.

In this paper, we propose a scheme based on deep neural networks of obfuscation prediction for binary, which builds a classification model using the contextual semantic information of the disassembled binary in the whole document. The inspiration for this approach comes from the classification

problem in NLP. Unlike past approaches, we take advantage of neural network models to improve the accuracy and efficiency of obfuscation detection. Further, when selecting feature representation for neural network models, we exploit the context semantic features of the disassembled binary. That's because code obfuscation [13], [53] is essentially a technique that transforms a program P into another program P' through a specific code transformation method, while P and P' maintaining at least semantic equivalence in observable behavior.

There are some challenges in our work. First of all, what kind of features of the target program are suitable for applying deep neural networks to obfuscation detection? Taking into account three characteristics of obfuscated programs, such as custom identifiers, strong structure, and long dependency, we adopt the contextual semantic representation of the entire target program to capture as many obfuscated features as possible. Specifically, the approach models contextual semantics of the disassembled binary in three steps. In the first step, it takes instructions as words and uses word2vec in terms of skip-gram to produce the instruction representations. Then, it treats basic blocks as sentences and uses convolutional neural networks(CNN) to generate basic block representations from instruction representations. Afterward, long short term memory networks(LSTM) is employed to adaptively encode the semantics of basic blocks and their inherent relations in target program representations. These representations are naturally used as features to classify the code obfuscation label of each target program.

Secondly, out-of-vocabulary(OOV) [62], [5] is a well-known problem in NLP. The OOV word means that it has never appeared during training. Inevitably, code obfuscation can introduce a lot of OOV words. For example, EncodeLiterals [12] is one of the obfuscation transformations, which replaces function names or variable names in the original code with meaningless strings to increase the difficulty of reverse analysis. How to effectively avoid the OOV issue while ensuring the accuracy of this kind of obfuscation scheme detection is another challenge. We deal with the OOV problem in a pre-processing manner.

Finally, there is a lack of datasets for training and testing our obfuscation detection model. High-quality labeled datasets are more conducive to the knowledge of deep neural networks. However, in the field of code analysis, publicly labeled datasets for binary are scarce, let alone datasets with obfuscation labels, since it needs lots of expert experience. Therefore, how to obtain high-quality code obfuscation labeled data is one of the significant challenges and the basis of all our work. We spend three months using the OLLVM [25] and Tigress [14] to construct datasets of obfuscated codes. The datasets include eight types of single obfuscation and six types of multiple obfuscations, with 277,131 labeled data in total.

We develop a tool of semantic-aware obfuscation scheme prediction for binaries with the name of OBFEYE and evaluate its accuracy and efficiency. The experimental results show that the average accuracy of single obfuscation detection is 89.40%, and the average accuracy of multiple obfuscations is 82.79% when implementing testing cases on our own dataset. When taking the *Obfuscation Benchmarks* from TUM [51] as a testing dataset, the accuracy of a single obfuscation is 91.81%, and the accuracy of multiple obfuscations is as high as 97.84%. In a word, OBFEYE is a highly accurate code obfuscation detection tool with a modest cost, especially for multiple obfuscations and single obfuscation algorithms like control flow flatten, add opaque predicates, virtualization.

Contributions: This paper makes the following contributions:

- Inspired by NLP, We initiate the study of using deep neural networks for obfuscation scheme prediction on binary. We treat instructions as words, basic blocks as sentences, and target programs as documents, and build a deep neural network model based on contextual semantic information of the disassembled binary for obfuscation detection. This research successfully demonstrates that it is promising to approach binary analysis from the angle of language processing by adapting methodologies, ideas, and techniques in NLP.
- We build a high-quality labeled dataset with specific obfuscated information. There are 277,131 labeled data in total in the dataset, including eight kinds of single obfuscation and six kinds of multiple obfuscations. It may be valuable to other experts in the field of code obfuscation.
- We implement a prototype of OBFEYE and evaluate its accuracy and efficiency. The experimental results show that it is a highly accurate code obfuscation detection tool with a modest cost, especially for multiple obfuscations and most single obfuscation.

II. Background

In this section, we first summarize the characteristics of code obfuscation in Section II-A. Next, we describe the application of deep learning in the field of code analysis in Section II-B. Finally, we propose three guidelines for the code obfuscation detection model. According to the guidelines, a semantic-aware obfuscation detection model is presented in Section II-C.

A. Code Obfuscation

Obfuscation is a technique that transforms the original code into obscure code while preserving the functionality of the program such that it is harder to analyze and tamper with. Collberg et al. [12] proposed a general taxonomy of obfuscating transformations. According to effectiveness and efficiency, obfuscating transformations are classified with four categories, including layout obfuscation, data obfuscation, control obfuscation, and preventive obfuscation. So far, the research on code obfuscation algorithms has been very mature, and various algorithms have emerged endlessly [12], [29], [54], [13], [41], [56]. Table I shows eight typical code obfuscation algorithms, most of the existing algorithms are derived from them.

Through in-depth research on these obfuscation algorithms, we summarize the following characteristics of the obfuscated codes.

TABLE I
OBFUSCATION TRANSFORMS. OLLVM AND TIGRESS ARE TWO OPEN SOURCE OBFUSCATION TOOLS. THE SECOND COLUMN LISTS EIGHT TYPICAL CODE OBFUSCATION TRANSFORMS.

Tool	Transform	Description
OLLVM [25]	Instructions Substitution(sub)	Replace binary operators like addition, subtraction or boolean operators.
	Bogus Control Flow(bcf)	Add opaque predicates making a conditional jump to original basic block.
	Control Flow Flattening(fla-o)	Break down the program's control flow [29].
Tigess[11]	Flatten(fla-t)	Break down the program's control flow [54].
	AddOpaque(opa)	Add opaque predicates to split up control-flow.
	EncodeArithmetic(ari)	Replace integer arithmetic with more complex expressions.
	EncodeLiterals(lit)	Replace literal integers and strings with less obvious expressions.
	Virtualize(vir)	Replace code with virtualized instructions and execute them by interpreter.

Custom identifier. Obfuscated code usually contains a lot of user-defined identifiers without semantic information, such as function names, variable names, class names, etc. When building a word embedding model in NLP, it is necessary to pre-build a vocabulary that can cover most words in the dataset. However, this modeling method cannot be directly transferred to code embedding since the pre-built code vocabulary often cannot satisfy the needs of coverage. No matter how large the code vocabulary is, new codes that always appear outside this vocabulary. So, it makes the OOV problem much more severe when implementing code embedding.

Strong structure. Obfuscated code is highly structured with lots of loops and nesting. For example, the program after control flow flattens [29], [54] will expose the typical structural characteristic of $switch-case$ statements. Adding opaque predicate transformations [13], [41] will cause many $if-else$ statements in the program. These structural features are vital clues to identify code obfuscation algorithms. How to retain as much structural information as possible is a significant challenge for identifying obfuscation.

Long dependence. In a programming language, the dependency interval between contexts may be very long. For example, variables defined at the beginning of the program may be used at the end of the program. This phenomenon is more common in obfuscated codes, with larger dependency intervals between contexts. For instance, code virtualization [59], [63] defines lots of virtual instructions, a set of bytecode handlers that first decode the virtual instruction and then translate it into native machine code, and a dispatcher that determines which instruction is ready for execution. A complete set of virtual instructions requires a lot of code implementation, which directly leads to a wider coverage of the context of an instruction in the virtualized code. How to capture as much contextual information between codes as possible requires careful consideration

Figure 1 are examples of obfuscated code snippets with semantic information. As shown in Figure 1 (a), control flow flatten usually transforms $if-else$ statements into $switch-case$ statements. In Figure 1 (b), virtualization enables program execution to switch continuously between the native environment and the virtual machine environment through push and pop operations. Therefore, contextual semantic information is an essential factor in achieving confusion detection.

```
LFB4:
    ···// initialize
    mov  dword ptr [esp+24], eax
    mov  dword ptr  [esp+28], 0
L10:
    mov  eax, dword ptr  [esp+28]
    cmp  eax, 1
    je     L3
    cmp  eax, 2
    je     L4
    test   eax, eax
    je     L5
    jmp   L8
L5:
    cmp  dword ptr  [esp+24], 0
    jne    L6
    mov  dword ptr  [esp+28], 1
    jmp   L8
L6:
    mov  dword ptr  [esp+28], 2
    jmp   L8
L3:
    mov  eax, 1
    jmp   L9
L4:
    mov  eax, 10
    jmp   L9
L8:
    jmp   L10
L9:
    leave
```

(a) Control Flow Flatten

```
···// native program execution
push  edi // context saving
push  esi
push  ebx
push  edx
push  ebp
push  ecx
push  eax
pushfd
jmp   loc_400550
··· //
··· // virtualized snippet begin
mov ...
add ...
xor …
···//
··· // virtualized snippet end
popfd
pop   eax // context restoring
pop   ecx
pop   ebp
pop   edx
pop   ebx
pop   esi
pop   edi
jmp   loc_401890
··· // continue native program
··· // execution
```

(b) Virtualization

Fig. 1. Examples of obfuscated code snippets with semantic information.

B. Deep learning in binary code

Deep learning provides an end-to-end learning paradigm. Deep learning provides an end-to-end learning paradigm. The entire learning process delivers the problem to a deep neural network model for learning the mapping from the original data to the expected output. In recent years, deep neural networks have received widespread attention in many fields, such as speech recognition, image processing, and natural language processing. Some NLP models can be transplanted to programming language analysis because there are many similarities between these two kinds of language. Firstly, both of them are composed of tokens. Secondly, they can be parsed into the form of grammar trees. Most importantly, they both

have high repetitiveness [21].

Some empirical studies have demonstrated the feasibility of neural networks such as Recurrent Neural Network(RNN), CNN, LSTM in the field of code analysis and proved that deep learning technology could bridge the semantic gap between the programming language and natural language [64]. For example, in recent years, many neural network models have been used in programming language processing, such as binary code clone detection [65], [60] and vulnerability Detection [57], [33].

However, these existing programming language processing models cannot easily transplant on code obfuscation detection. After all, there are big differences in their business logic. Code clone detection focuses on judging whether two code fragments are syntactically similar or two pieces of cross-platform compiled code originate from the same source code but has little processing power for things like changing, adding, or deleting statements [65]. However, code obfuscation usually introduces a large number of statements. For example, adding opaque predicates transformation introduces false branches [41], and virtualization introduces statements for dispatcher and handler [28], [20]. Vulnerability detection is more inclined to pin down the locations of its vulnerability [57], [33]. In other words, the code fragments of vulnerability are generally short and concealed to avoid detection as much as possible. In contrast, most of code obfuscation algorithms require longer code fragments to disguise the original code.

In general, this means that the existing code analysis neural network model is not suitable for code obfuscation detection. We have to select those neural network models that better characterize code obfuscation to implement detection.

C. Semantic Neural Network Model

To better represent the features of code obfuscation, we have proposed some guiding principles suggesting the construction of neural network models for code obfuscation detection.

Guiding Principle 1: Whether a piece of code is obfuscated may depend on the context, so a neural network that can handle the context may be suitable for obfuscation detection. Through observation, We find that most obfuscation transformations require multiple instructions to implement instead of one instruction.

Guiding Principle 2: The larger the context window, the more semantic information obtained, and the more useful information provided to the neural networks model. In particular, most code obfuscation transformations include conditional statements, such as control flow flatten and the opaque predicates insertion. There is a long interval between the conditional statements and the execution statements.

Guiding Principle 3: In the span of context, keeping the order of instructions provides the possibility for the neural networks to track their dependencies. We observe that no matter how the code obfuscation transforms in form, it is based on a big premise, that is, the programming language is a logical deduction language with a strict order.

Based on the above guiding principles, let us analyze which kind of neural networks is suitable for code obfuscation detection. CNN [30] is a type of feedforward neural network with a deep structure that includes convolutional calculations. It has a strong ability to extract local features of shallow text with high efficiency when categorizing short text fields. However, it has limited capabilities in long-distance modeling and is insensitive to word order, since it mainly relies on the filter window to extract features. LSTM [22] contains three gates: forget gate, input gate, and output gate. Because of the existence of the gates, LSTM can well learn and grasp the front-to-back dependencies in the sequence, so it is more suitable for dealing with NLP problems of long sequences. But it is time-consuming because of the deep network and a large amount of calculation.

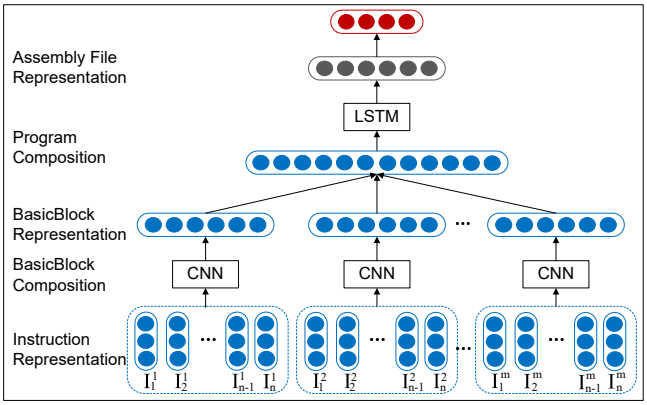


Fig. 2. A semantic neural network model for obfuscation detection. The neural network takes in assembly files and produces the probability of the target program being obfuscated. First, it takes instruction as a word and uses word2vec in terms of skip-gram to produce the instruction representations. Then, it treats basic blocks as sentences and uses CNN to generate basic block representations from instruction representations. Next, LSTM is employed to adaptively encode the semantics of basic blocks and their inherent relations in program representations. Finally, these representations are naturally used as features to classify the assembling files with the obfuscated label.

Through the combination of CNN and LSTM, we not only use CNN to obtain the local characteristics but also use LSTM to identify the instruction sequence, to fully obtain the context semantic information of the whole target program. Figure 2 highlights the semantic neural network model for obfuscation detection. It takes in the assembly files and generates the probability of the target program being obfuscated. In this model, each instruction is regarded as a word, including opcodes and operators. The instruction is converted into instruction embedding through word2vec in terms of skip-gram. The basic blocks are treated as sentences, which employ CNN to generate basic block representation from instruction representations. Afterward, LSTM is employed to adaptively encode the semantic of basic blocks and their inherent relations in program representations. Finally, the representations are naturally used as features to classify the code obfuscation label of each assembling file.

III. Overview of Our Approach

OBFEYE is a semantic-aware obfuscation scheme prediction tool that has essentially put our idea into practice. The approach is on the basis of semantic context information of the disassembled binary to predict if the program has been obfuscated, and if so, what type of obfuscation scheme may be

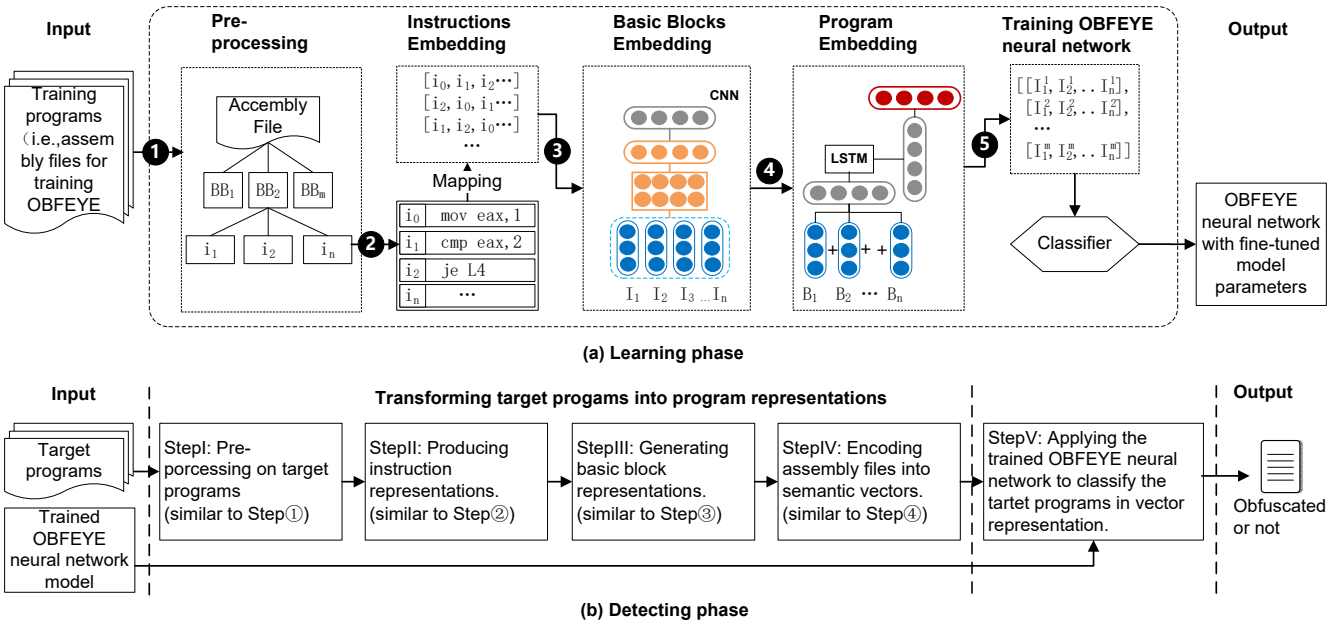


Fig. 3. Overview of OBFEYE. The learning phase takes in assembly files and outputs a trained semantic neural network with fine-tuned model parameters. In the detection phase, the trained semantic-based neural network model is used to detect whether the target programs being obfuscated and what kinds of obfuscation algorithms are employed.

used. At a high-level OBFEYE contains two phases: a learning phase and a detecting phase.

The learning phase:

The learning phase takes in assembly files with the obfuscated labels and produces a trained semantic neural network with fine-tuned parameters. It goes through the following steps as highlighted in Figure 3(a).

- Step 1. Pre-processing. In this step, there are two operations on the assembly file. One is using eight rules to normalize assembly instructions to address the OOV issue. The other is dividing the assembly file into basic blocks. The details can be found in Section IV-A1 and Section IV-A2.
- Step 2. Instruction embedding. A whole instruction(including operators and operands) is considered as a word in NLP and uses skip-gram to produce instruction embedding. The details are described in Section IV-B1.
- Step 3. Basic Block Embedding. Basic blocks are taken as sentences in NLP and CNN is employing on generating sentence representations from instruction representations. This is described in Section IV-B2.
- Step 4. Program Embedding. In Setcion IV-B3, LSTM is employed to adaptively encode the semantics of basic blocks and their inherent relations in program representations.
- Step 5. Obfuscation Classification. In the end, the representations are naturally used as features to classify each assembling file with the obfuscated labels in Section IV-B3.

The detecting phase:

The detection phase takes in a given one or multiple target programs in the form of assembly code. They are transformed into their semantic representations, which are encoded into vectors and used as input to the trained semantic neural network. The model predicts whether the program has been obfuscated, and if so, what type of obfuscation scheme may be used. The learning phase goes through the following steps, as highlighted in Figure 3(b). Step I - Step IV is similar to Step 1 - Step 4 in the learning phase. Step V uses the learned semantic neural network to classify the vectors corresponding to the target programs.

Learning and detecting data set:

Because of the lack of binary code datasets with the obfuscated labels, we have to build our own. The semantic model for obfuscation in OBFEYE is learning and detecting using obfuscation labeled datasets, which takes three months to build. The details can be found in Section IV-C. In order to prevent the impact of the quality of the self-constructed data set on OBFEYE, we find another *Obfuscation Benchmarks*, which are built by TUM [51] also as the detecting set. The experimental results are shown in Section **??**.

IV. Implementation Details

In this section, we start with depicting the pre-processing on both training programs and target programs in Section IV-A, then introduce the semantic neural network model for code obfuscation detection in Section IV-B, finally we introduce the construction of datasets with the obfuscated labels in Section IV-C.

A. Pre-processing

1) Preprocessing of program: The obfuscation program contains a large number of obfuscated identifiers, so when building the semantic neural network model, a vocabulary with an extensive words is needed. Even so, it is still impossible to cover all the tokens in the program, and there are still many

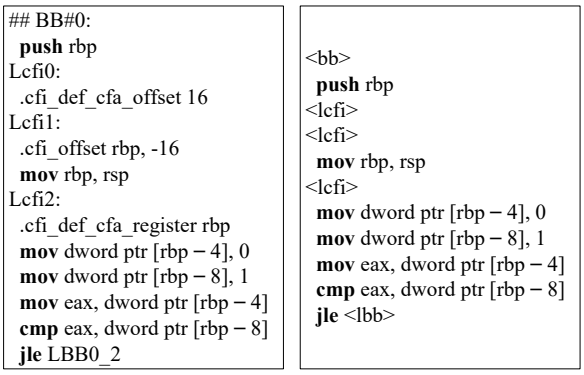


Fig. 4. Preprocessing example. The code snippet on the left is the original assembly code, and the right is the result after preprocessing.

tokens outside the vocabulary. Therefore, the OOV issue is a big challenge.

To address the OOV issue, we propose the following rules on the preprocessing of the program with assembly code:

(1) The number constant values are replaced with 0.
(2) The basic block identifiers are replace with <bb> .
(3) The lcfi identifiers are replace with <lcfi>.
(4) The jump instruction identifiers are replace with <lbb>.
(5) The function names are replace with <func>.
(6) String constants are replace with <str>.
(7) Other symbol constants are replace with <symb>.
(8) All comments are deleted.

Take the code snippets in Figure 4 as an example, the code snippet on the left shows the original assembly code, and the right one is the result after preprocessing.

2) Slicing Basic Block: A basic block in the traditional sense is a piece of code that executes sequentially. When the program runs this code, if no control flow error occurs, the program can only enter the basic block from the first sentence and exit the basic block from the last sentence. That is, under normal circumstances, no control-flow branch should occur in the basic block except for the last sentence, and no control-flow entry should occur in the basic block except for the first sentence. In assembly code, it is usually to take the jump statements as the cutting points to divide basic blocks. Code obfuscation transformations such as control flow flatten and bogus control flow change the control flow structure in the function. It means that the original code in a function splits into multiple basic blocks by adding jump instructions to increase the complexity of code analysis.

Figure 5 is two different kinds of basic block slicing schemes. (a) takes jump instructions as cutting points to divide the target program into basic blocks. This kind of slicing scheme is not suitable for building a semantic neural network model. On the one hand, each basic block contains too few instructions, which results in less context semantic information available to the neural network. On the other hand, The dependencies between each basic block are lost. (b) treats a function as a basic block and keeps its internal instruction in sequence. This way can fully preserve the correlations between instructions in the function. Specifically, each function has a specific function, and its internal instructions also serve this common goal. Therefore, these instructions have a close contextual relationship. Thus, in our semantic neural network model for obfuscation detection, we take a function as a basic block.

In the specific implementation, we obtained the assembly code by gcc compiler on Linux, where $\#BB\#$ has been used to mark the beginning of each function in the generated $*.s$ file. Therefore, we cut the basic block by identifying $\#BB\#$ label and the ret instruction.

B. Semantic Model of OBFEYE

The approach models contextual semantics of assembly code as the following steps: First, it takes instruction as a word and uses word2vec in terms of skip-gram to produce instruction representations. Second, it treats basic blocks as sentences and uses CNN to generate basic block representations from instruction representations. Third, LSTM is employed to adaptively encode the semantics of basic blocks and their inherent relationships in program representations. These representations are naturally used as features to classify the assembling file with the obfuscated labels.

1) Instruction embedding generation: An instruction consists of an opcode and an operator. For example, *mov eax, dword ptr [rbp 4]* is an instruction, where *mov* is the opcode and *eax, dword ptr [rbp 4]* are operators. In NLP, a word is converted into a word embedding vector. Similarly, we treat an instruction as a word, so the instruction *mov eax, dword ptr [rbp 4]* will be converted to an instruction embedding vector.

There is a big "semantic gap" between a programming language and natural language. Usually, a word embedding generated by the deep learning model can fill this "semantic gap". In particular, a word embedding is to capture the contextual semantic meaning of this word. Thus, words with similar contexts are closer to each other in high-quality word embeddings. Recently, a series of neural network models [39], [40] have been proposed to learn high-quality word embeddings.

Among these models, Mikolov's skip-gram model [39], [40] is popular due to its efficiency and low memory usage. The training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document with the help of a sliding window. Figure 6 is an example of a sliding window working on instructions in the skip-gram model. The size of the window is 2, covering the first two and last two instructions of the current instruction.

Let us denote a sequence of instructions in a program as $\{i_1, i_2, ..., i_t, ..., i_T\}$. The objective of the Skip-gram model is to maximize the average log probability [40] as shown in Formula 1.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-C\leq j\leq C, j\neq 0}\log p(i_{t+j}|i_t) \tag{1}$$

Where i_t is the current instruction, C is the context of i_t covered by the sliding window. The larger the sizes of sliding window, the more training samples are generated, which leads to a higher accuracy, at the expense of the training time.

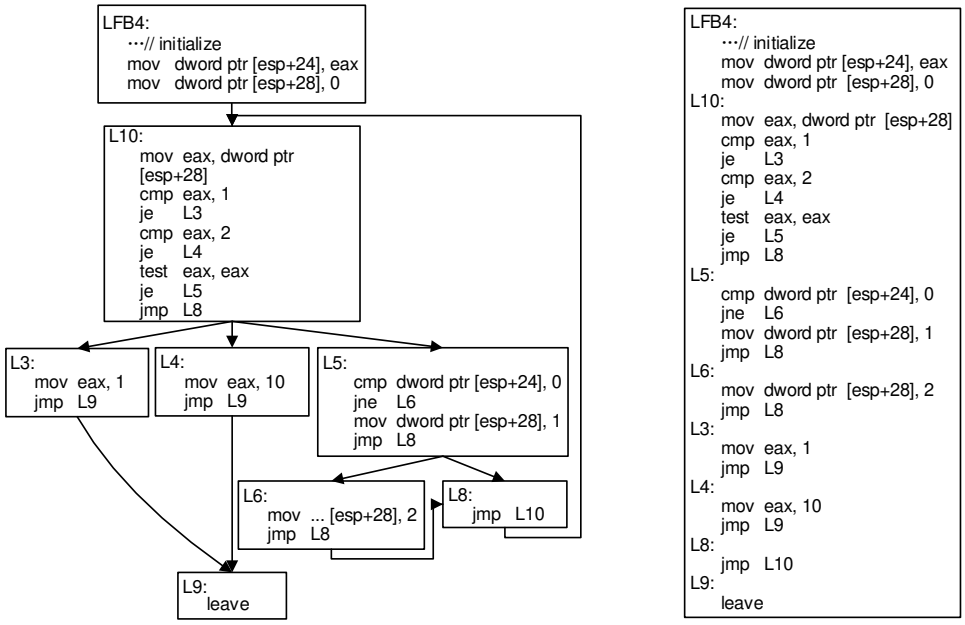


Fig. 5. Basic block slicing granularity. (a) takes the jmp as the cutting point to divide a program into Basic Blocks. (b) treats a function as a Basic Block and keeps its internal instruction in sequence. In contrast, scheme (b) provides more contextual semantic information for our model.

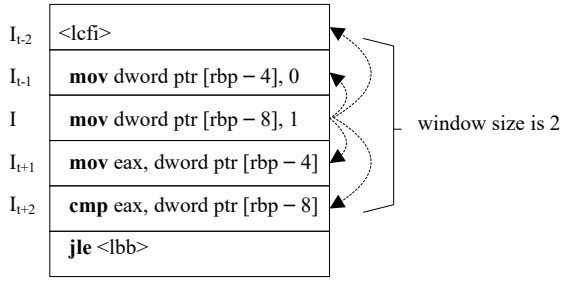


Fig. 6. A sliding window on instructions in the skip-gram model.

$$p\left(i_k \in C_t | i_t\right) = \frac{\exp\left(\mathbf{I}_t{}^{\mathsf{T}} \mathbf{I}_k\right)}{\sum_{i_j \in C_t} \exp\left(\mathbf{I}_t{}^{\mathsf{T}} \mathbf{I}_j\right)} \tag{2}$$

Skip-gram uses $softmax$ function to predict the probability of i_k appearing in the sliding window of i_t. The $softmax$ function is defined in Formula 2, where $\mathbf{I}_t$, $\mathbf{I}_k$, $\mathbf{I}_j$ are the embedding of instructions i_t, i_k, i_j, respectively.

According to the index mapping, all the instruction are stacked in a word embedding matrix $\mathbf{I} \in \mathbb{R}^{d \times |V|}$, where d is the dimension of word vector and $|V|$ is vocabulary size. In this way, all instructions can be mapped onto the matrix $\mathbf{I}$, and each instruction corresponds to a certain column in the matrix. For example, the j_th column in the word embedding matrix $\mathbf{I}$ corresponds to the j_th instruction in the dictionary, so it is marked as $\mathbf{I}_j$.

2) Basic Block embedding generation: We use CNN to compute continuous representations of basic block with semantic composition. The core idea of CNN is to capture local features. For text, local features are sliding windows composed of several words, similar to N-gram. The advantage of a convolutional neural network is that it can automatically combine and filter N-gram features to obtain semantic information at different levels of abstraction.

Step 1: Input layer. Let us denote a sentence consisting of n instructions as $\{i_1, i_2, i_3 ... i_n\}$. The input layer is a matrix of $n \times d$, where n is the number of instructions in a basic block, and d is the dimension of the word embedding vector corresponding to each instruction. In other words, each line of the input layer is a d-dimensional word embedding vector corresponding to an instruction. In addition, in order to make the vector length consistent, the original basic block is padding. We use $\mathbf{I}_j \in \mathbb{R}^d$ to represent the d-dimensional word embedding of the j_th instruction in the basic block.

Step 2: Convolution operation. The correlation between adjacent instructions in the basic block is high, so we only use one convolution kernel $\mathbf{W}$. The width of the convolution kernel $\mathbf{W}$ is d and the height is h. The convolution operation formula is as follows:

$$\mathbf{O}_j = \mathbf{W} \times \mathbf{I}_j, (j = 1, 2, ..., n - h + 1) \tag{3}$$

Add the bias b and activate it with the activation function f to get the feature: $\mathbf{B}_j = f(\mathbf{O}_j + b)$. The output of this layer is $\mathbf{B} = [\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{B}_{n-h+1}]$.

Step 3: Polling and FullConnection. To capture global semantics of a basic block, we feed the outputs of linear layers to a 1-max pooling layer, resulting in an output vector with fixed-length. We further average the outputs of multiple filters to get basic block representation.

3) Program embedding generation: In this paper, we are inspired by NLP. When processing text data, due to the length of the document, traditional RNN or N-gram algorithms cannot capture the dependencies between words that are far apart

relationship. LSTM [22] solves the problem of the word's long-range dependence by designing the input gate, forget gate, and output gate. The input gate determines the amount of change in the information in the memory cell by the input vector of the LSTM unit at the current time. The forget gate determines the influence of historical information at the previous moment on the information in the memory cell at the current moment. The output gate controls the amount of information output in the memory cell.

The inputs of LSTM are m basic blocks, $\{\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{B}_m\}$, represented as a sequence of instruction embeddings, $\left(\mathbf{I}_1^1, \mathbf{I}_2^1, ..., \mathbf{I}_n^1\right)$, $\left(\mathbf{I}_1^2, \mathbf{I}_2^2, ..., \mathbf{I}_n^2\right)$, and $(\mathbf{I}_1^m, \mathbf{I}_2^m, ..., \mathbf{I}_n^m)$ respectively. An LSTM cell analyzes an input vector coming from either the input embeddings or the precedent step and updates its hidden state at each time step. Each cell contains four components which are real-valued vectors. They are an input gate $\mathbf{i}$, an output gate $\mathbf{o}$, a memory state $\mathbf{c}$, and a forget gate $\mathbf{f}$.

For example, an LSTM cell at the first layer in LSTM updates its hidden state at the time step t via Equations 4 – 9.

$$\mathbf{i}_t^1 = \text{sigmoid}(\mathbf{W}_i \mathbf{I}_t^1 + \mathbf{U}_i \mathbf{x}_{t-1}^1 + \mathbf{v}_i) \tag{4}$$

$$\mathbf{f}_t^1 = \text{sigmoid}(\mathbf{W}_f \mathbf{I}_t^1 + \mathbf{U}_f \mathbf{x}_{t-1}^1 + \mathbf{v}_f) \tag{5}$$

$$\widetilde{\mathbf{c}}_t^1 = \tanh(\mathbf{W}_c \mathbf{I}_t^1 + \mathbf{U}_c \mathbf{x}_{t-1}^1 + \mathbf{v}_c) \tag{6}$$

$$\mathbf{c}_t^1 = \mathbf{i}_t^1 \odot \widetilde{\mathbf{c}}_t^1 + \mathbf{f}_t^1 \odot \widetilde{\mathbf{c}}_t^1 \tag{7}$$

$$\mathbf{o}_t^1 = \text{sigmoid}(\mathbf{W}_o \mathbf{I}_t^1 + \mathbf{U}_o \mathbf{x}_{t-1}^1 + \mathbf{v}_o) \tag{8}$$

$$\mathbf{x}_t^1 = \mathbf{o}_t^1 \odot \tanh(\mathbf{c}_t^1) \tag{9}$$

4) Obfuscation Classification: The composed program representations can be naturally regarded as features of programs for obfuscation classification. In the classifier we add a linear layer and a softmax layer. The former converts program vectors to real-vectors and the latter converts the real-vector into a probability value. Specifically, we use $Softmax$ function in softmax layer to implement multi-classification. In simple terms, the $Softmax$ function maps some output neurons to real numbers between (0-1), so that the sum of the probabilities of multiple classifications is exactly 1. The $Softmax$ function is defined as follows:

$$P_i = \frac{e^{V_i}}{\sum_{i-1}^{C} e^{V_i}} \tag{10}$$

Where V_i is the output of the previous unit of the classifier. i represents the category index, and the total number of categories is C. P_i represents the ratio of the index of the current element to the sum of the indices of all elements. Through the $Softmax$ function, the output value of multiple classifications can be converted into relative probability.

In model training, we use cross entropy as a loss function. Cross-entropy describes the distance between the actual output probability and the expected output probability. The smaller the value of the cross-entropy, the closer the two probability distributions. The loss function is defied as the following:

$$H(p, q) = -\sum_{i=1}^{C} p(d_i) \log(q(d_i)) \tag{11}$$

Where, d is the representation of the document, C is the total number of categories, p is the expected probability distribution, and q is the actual probability distribution which is calculated using the data labeled in the data sets.

C. Annotation Dataset Construction

1) Construct Obfuscation Samples: The data source is the key to complete program understanding tasks when using deep learning. Most importantly, the quality of the data source determines the accuracy of the neural network model. Standardized and high-quality data is more conducive to the learning of deep neural networks. In image recognition and natural language processing, there are many open datasets for researchers to conduct research. However, in the field of program analysis, public, available, and high-quality datasets are very scarce. Many researchers have to construct their own data sets to complete the corresponding program analysis tasks. It leads to low data quality in the corpus and brings a lot of noise to the neural network model. In addition, the use of various data sets in the same task is very inconvenient for comparing and evaluating different methods. Therefore, how to obtain a unified and standardized high-quality obfuscated program corpus is a challenge.

To construct a dataset 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 a label of obfuscated type. First, it checks whether the source file contains obfuscation points. Obfuscation points are defined here as those codes that can apply for obfuscation transformations, such as character strings, arithmetic operators, jump statements, assignment statements, functions, etc. Second, it chooses the corresponding obfuscation algorithm for different obfuscation points. For example, if there is an arithmetic operation statement as $a = b + c$ in the original file, the code obfuscated sample generator selects the arithmetic obfuscation to implement transformation on this file. Third, if the code transformation is successful, the obfuscated code will be compiled into assembly code by gcc. Fourth, the corresponding obfuscation algorithm is used to name the obfuscated assembly file. That is, the file name is considered as an obfuscated label.

To establish the ground truth about the obfuscated assembly codes, we use two open-source code obfuscation tools, OLLVM [18] and Tigress [11]. OLLVM is a project initiated by the information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains. OLLVM aims to provide an open-source fork of the LLVM compilation suite able to provide increased software security through code obfuscation. Tigress is a project developed by

the Department of Computer Science at the University of Arizona [35], [18], [25]. Tigress is a diversifying virtualizer/obfuscator and supports to resist both static and dynamic reverse engineering and de-virtualization attacks. Both OLLVM and Tigress support the obfuscation for C language. There are two reasons why we chose open source obfuscation tools instead of commercial obfuscators [48], [46], [50], [49]. On one hand, compared to commercial obfuscators, it is easier to get their internal obfuscation principles. On the other hand, it is very flexible to configure obfuscation strategies 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 C language. So, to obtain obfuscated assembly samples, we use the gcc compiler to produce *.s" file. The details about the obfuscated datasets in Section V-A.

2) Verify the Effectiveness of Obfuscation Samples: It is a very tedious and subjective matter to verify whether the sample is obfuscated successfully manually. Therefore, we use three kinds of metrics to verify the effectiveness of obfuscation transformations from the perspective of complexity. The first kind is the number of instructions, which counts the number of all assembly instructions in the file after obfuscation. The second kind consists of basic statistics about the call graph and control flow graph (CFG), including the number of edges in both graphs and the number of basic blocks. The third metric is the cyclomatic number [38]. It is defined as $e - n + 2$, where e and n are the numbers of edges and vertices in the CFG. These metrics have been proved to be able to evaluate the complexity of the obfuscated code and have been widely used to evaluate obfuscation techniques in related work. [55], [8], [13].

For each assembly file after obfuscation transformation, we calculate their three metrics to quantify their complexity. Compare with the complexity of the original assembly code, the samples whose complexity has not changed are considered to be failed. Only samples that are successfully obfuscated can stay in our datasets.

Although, from the perspective of obfuscation evaluation, complexity is not an adequate metric. Because there is already a proof that there is no direct correlation between complexity and security [4], but this metric is still sufficient in our scenario.

V. Experimental Setup

In this section we describe our experimental samples, evaluation metrics and implementation platforms.

A. Data Preparation

In the experiment, we use two data sources for verifying the performance of OBFEYE. One is the source code coming from $gcc-7.4.0$ and $GNUtoolkit$. 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 $GNUtoolkit$ is widely used in the GNU operating system. Both of them contain a lot of source code in C language. So, their source codes are considered as original samples, then implemented by the method described in Section IV-C to produce obfuscated samples.

To maintain the experimental results' objectivity and avoid situations where the experimental results are biased due to our own constructed datasets, we find another data source. So, the other data source is *Obfuscation Benchmarks* provided by TUM [51]. *Obfuscation Benchmarks* is built (but not limited) to compare the strength of different obfuscation transformations against both human-assisted and automated attacks. It has been used in the field of code obfuscation for the training and testing of machine learning models [2], [3].

Speaking of obfuscation transformations, there are eight kinds of single obfuscation and six kinds of multiple obfuscation in our experiments. It should be emphasized that, in most cases, multiple obfuscation is composed of two types of obfuscation transformations. In this paper, we do not discuss the situation where there are more than two obfuscation transformations. Dataset1-3 are single obfuscation datasets as shown in Table II. Dataset4-6 are multiple obfuscation datasets as shown in Table III.

The details of each dataset are described as the following:
Dataset1: A single obfuscation dataset. Its source codes come from *gcc-7.4.0*. Eight kinds of obfuscation transformations are performed on all source codes, respectively. They are options of *-bcf, -fla, -sub* in OLLVM and options of *-Flatten, -AddOpaque, -Virtualize, -EncodeArithmetic, -EncodeLiterals* in Tigress. The details of these obfuscation transformations are described in Table I. Finally, there are a total of 100,287 samples in Dataset1.

Dataset2: A single obfuscation dataset. Its source codes come from *GNU Toolkit*. The obfuscated samples are produced in the same way as Dataset1. This dataset contains a total of 10,259 samples.

Dataset3: A single obfuscation dataset. Its source codes come from *Obfuscation Benchmarks*. The benchmarks supply plenty of $*.c$ files and obfuscation scripts to generate obfuscation samples. There are 50,370 samples in total.

Dataset4: A multiple obfuscations dataset. Its source codes come from *gcc-7.4.0*. We use *-bcf, -fla, -sub* option in OLLVM to superimpose each other to generate corresponding data. For example, we use the command like *-mllvm; -fla; -sub* to perform control flow flatten and instruction replacement transformation on the original code *"a.c"*, and save the generated code to a new file named *"a-fla_sub.s"*. There are a total of 73,536 samples in Dataset4, including six types of multiple obfuscation.

Dataset5: A multiple obfuscations dataset. Its source codes come from *GNU Toolkit*. The obfuscated samples are produced in the same way as Dataset4. This dataset contains a total of 7,420 samples.

Dataset6: A multiple obfuscations dataset. Its source code comes from *Obfuscation Benchmarks*. We use the supplied scripts to produce the multiple obfuscation samples. There are 35,259 samples in total.

TABLE II

SINGLE OBFUSCATION DATESETS. $Dataset1$'S SAMPLES COMING FROM $gcc-7.4.0$, $Dataset2$'S SAMPLES COMING FROM $GNUtoolset$, $Dataset3$'S SAMPLES COMING FROM $ObfuscationBenchmarks$. #PROS IS THE NUMBER OF PROGRAMS, #BBS IS THE NUMBER OF BASIC BLOCKS, #INS IS THE NUMBER OF INSTRUCTIONS.

Dataset1													
	Dataset1:OLLVM					**Dataset2:Tigress**							
	code	**fla**	**sub**	**bcf**	**Sum**	**code**	**opa**	**fla**	**vir**	**ari**	**lit**	**Sum**
#Pros	12717	8301	6652	9524	37194	10909	10698	10404	10208	10407	10467	63093
#BBs	256006	123385	221311	217980	818682	126395	1258219	105891	93371	99974	129672	1813522
#Ins	5135561	7600786	5089106	12163194	29988647	2754745	19779676	2340494	7145576	9516787	2354060	43891338
Dataset2													
	Dataset2:OLLVM					**Dataset2:Tigress**							
#Pros	1088	1008	794	1057	3947	1087	952	1082	1025	1082	1084	6312
#BBs	101804	13262	98754	74497	288317	71603	200436	88112	33280	84821	128465	606717
#Ins	1779845	4313281	2150187	5440434	13683747	1235141	4161749	3314910	4462367	4981274	11382623	29538064
Dataset3													
	Dataset3:OLLVM					**Dataset3:Tigress**							
#Pros	5037	5037	5037	5037	20148	5037	5037	5037	5037	5037	5037	30222
#BBs	51444	17625	51444	40930	161443	51444	148266	61380	55086	56343	76491	449010
#Ins	867549	1856775	1272639	2629662	6626625	867549	2318976	1222089	3144100	1361724	2811249	11725687

TABLE III

MULTIPLE OBFUSCATION DATESETS. $Dataset4$'S SAMPLES COMING FROM $gcc-7.4.0$, $Dataset5$'S SAMPLES COMING FROM $GNUtoolset$, $Dataset6$'S SAMPLES COMING FROM $ObfuscationBenchmarks$. #PROS IS THE NUMBER OF PROGRAMS, #BBS IS THE NUMBER OF BASIC BLOCKS, #INS IS THE NUMBER OF INSTRUCTIONS.

Dataset4								
	code	**fla_sub**	**sub_fla**	**fla_bcf**	**bcf_fla**	**sub_bcf**	**bcf_sub**	**Sum**
#Pros	12717	9522	9524	10340	10356	10521	10556	73536
#BBs	256006	141616	141610	146379	146507	219626	219765	1271509
#Ins	5135561	8594533	8593280	19407393	19527354	16685215	16901642	94844978
Dataset5								
#Pros	1088	1026	1026	1069	1074	1068	1069	7420
#BBs	101804	13289	13289	13417	13456	74796	74787	304838
#Ins	1779845	4737920	4736953	9999077	10047732	7998178	8041944	47341649
Dataset6								
#Pros	5037	5037	5037	5037	5037	5037	5037	35259
#BBs	51444	17625	17625	17625	17625	40970	40839	203753
#Ins	867549	2260256	2261931	4608707	4578436	4159234	4142780	22878893

B. Evaluation metrics

We use the widely used metrics of Accuracy(ACC) [37], [26] and True Positive Rate(TPR) [43] to evaluate the code obfuscation detection system, where both of them are standard metrics to measure the overall classification performance in NLP. Here is a brief introduction:

$$ACC = \frac{(TP + TN)}{(TP + FP + TN + FN)}$$

$$TPR = \frac{TP}{(TP + FN)}$$

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.

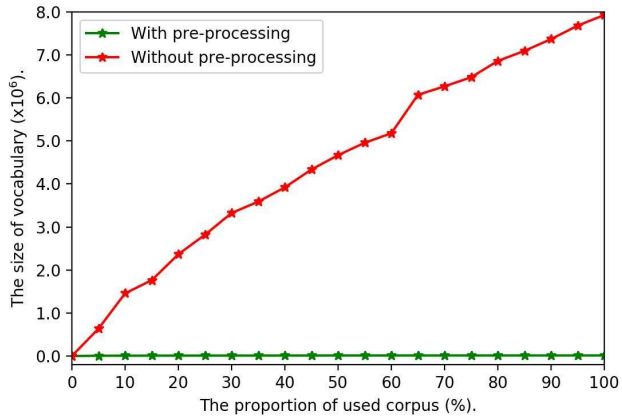
C. Implementation and Evaluation Platforms

Our prototype system is implemented using Python v.3.7. Specifically, the Word2Vec model is coded using gensim v.3.8.3, and the classifier model is built upon the Keras v.2.0.0 framework with TensorFlow v.1.13.1 as its backend. Keras provides a large number of high-level neural network APIs and can run on top of TensorFlow.

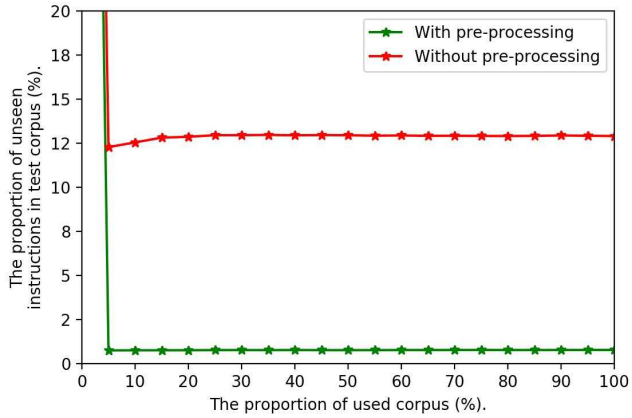
The experiments are performed on a computer running the Windows 7 operating system with two 64-bit 3.4 GHz Intel(R) Core(TM) i7-3770 CPU and 32 GB RAM without GPUs. The training and testing are expected to be significantly accelerated if GPUs are used.

VI. EXPERIMENTAL RESULTS

In this section, we first evaluate the effect of preprocessing on instruction embedding and discuss hyperparameter selection. We then present the ACC and TPR of our approach on single obfuscation and multiple obfuscations, showing that our approach is highly effective in identifying the obfuscation scheme. Finally, we compare our approach against other detection models, demonstrating that our approach significantly outperforms all other detection models.



(a) The growth of the vocabulary size. The vocabulary size in terms of the percentage of the corpus analyzed.



(b) The proportion of used corpus. The percentage of unseen instructions that do not exist in the vocabulary.

Fig. 7. Evaluation on Out-Of-Vocabulary Instructions.

A. Evaluation on Out-Of-Vocabulary Instructions

As pre-processing described in Section IV-A1 is applied to addressing the OOV issue, we evaluate its impact by studying the following two issues: 1) how the vocabulary size grows with or without pre-processing. 2) The relationship between the number of unseen words and the size of corpus with or without pre-processing.

The experiment is conducted using Dataset2:OLLVM as a corpus. The size of the samples extends from 1KB to 18MB. All samples are in the form of assembly files. When implementing instruction embedding, not only the opcode but also the operand are all taken into account as a word. In general, there are a total of 3,947 samples and 13,683,747 instructions for training the model.

Figure 7(a) shows the growth of the vocabulary size with and without pre-processing. The vertical axis represents the size of the vocabulary, and the horizontal axis represents the percentage of the corpus. The red and green lines represent the growth of the vocabulary size without or with pre-processing, respectively. From the figure we can see that if there is no pre-processing, The size of the pre-processed vocabulary is almost unchanged and much smaller than the unprocessed one.

As to the unseen words issue, we take the vocabulary generated from Dataset2:OLLVM as the training dataset and Dataset2:Tigress as the testing dataset. Then, we try to check how many instructions in Dataset2:Tigress don't appear in the trained vocabulary. Specifically, each time we take in 1k instructions from Dataset2:Tigress, and then calculate the proportion of words unseen in the vocabulary to all the words taken in.

Figure 7(b) shows the relationship between the number of unseen words and the size of corpus. The vertical axis indicates the proportion of unseen instructions in testing corpus, and the horizontal axis indicates the percentage of the corpus read. The red and green lines represent the propotion of unseen instructions in the corpus without or with preprocessing, respectively. From the figure we can see that after pre-processing, the rate of unseen instructions drops to 0.7%, compare to the 12% without pre-processing. This shows that the instruction embedding model with pre-processing has a good coverage of instructions.

TABLE IV
STATISTICS ON THE NUMBER OF BASIC BLOCKS IN A PROGRAM. #BB IS THE NUMBER OF BASIC BLOCKS IN EACH PROGRAM.

Programs	#BB					
	25%	30%	50%	60%	75%	90%
origin	2	2	3	4	6	12
fla	1	1	2	2	3	5
bcf	2	2	3	4	6	13
sub	3	3	4	5	8	16
multi	1	1	2	2	4	7
all	1	1	2	3	4	9

TABLE V
STATISTICS ON THE NUMBER OF INSTRUCTIONS IN A BASIC BLOCK. #INS IS THE NUMBER OF INSTRUCTIONS IN EACH BASIC BLOCK.

Programs	#Ins					
	25%	30%	50%	60%	75%	90%
origin	9	9	13	15	17	2
fla	9	9	15	17	18	76
bcf	10	14	18	43	67	101
sub	9	9	15	17	19	29
multi	9	15	17	23	93	139
all	10	16	26	38	64	146

B. Hyperparameter Selection

In this section, we investigate the impact of different hyperparameters on OBFEYE. In particular, we discuss the number of basic blocks in a program and the number of instructions in a basic block.

1) Numbers of Basic Blocks: In order to determine the hyperparameter of the number of basic blocks in the semantic neural network model, we perform a statistical analysis of Dataset1 with a total of 100,287 assembly code files and 73,880,035 instructions. As shown in Table IV, 86% of the programs contain less than or equal to 8 basic blocks. Therefore, we set the number of basic blocks of a program to 8.

2) Numbers of Instructions: In this experiment, we still perform statistical analysis on Dataset1 with a total of 100,287 assembly code files. According to statistics, all documents contain 2,632,204 basic blocks in total. Then we count the number of instructions contained in each basic block. The statistics result is shown in Table V, nearly 91.39% of the basic blocks contain less than or equal to 128 instructions. Therefore, we set the number of instructions in a basic block to 128.

It should be emphasized that when the number of instructions is less than 128, we use the character "0" to fill in. Otherwise, only 128 instructions are kept, and redundant instructions are deleted. Similarly, basic blocks are treated in the same way.

C. Accuracy of OBFEYE

In this section, we evaluate the accuracy of OBFEYE. The experimental results of single obfuscation are recorded in Table VI and the multiple obfuscations' in Table VII.

1) Model Training: In the learning phase, we first merge Dataset1(shown in Table II) and Dataset4(shown in Table III), then use 80% of them for training and the other 20% for validation. On the one hand, the samples in both Dataset1 and Dataset4 come from $gcc-7.4.0$. On the other hand, Dataset1 is a single obfuscation dataset, and Dataset4 is a multiple obfuscation dataset. In general, there are 139,000 samples in the training dataset, covering two types of obfuscation tools, eight kinds of single obfuscation, and six kinds of multiple obfuscations.

We use the training datasets to train OBFEYE individually for 100 epochs. After each epoch, we measure the ACC and TPR on the corresponding validation datasets, and save the model achieving the best ACC as the base model.

2) Accuracy of Single Obfuscation: In the detecting phase, we have two testing datasets for single obfuscation detection, and they are Dataset2 and Dataset3(both of them are shown in Table II). We evaluate the accuracy of the base model using the corresponding testing datasets.

From the experimental results in Table VI, when testing on our own constructed Dataset2, the average accuracy of OBFEYE for single obfuscation detection reached 89.4%. As to TPR, each obfuscation method behaves slightly differently due to their different semantic features. Through further analysis, we find that options of -bcf, -fla in OLLVM and options of -fla, -opa, -vir in Tigress perform excellent. All the TPRs of them are above 95%, even as high as 99.62%. The reason is that these types of code obfuscation transformations usually require more instructions and basic blocks to achieve. They all meet the two typical characteristics of obfuscation algorithms of strong structure and long dependence. That is in line with the design of OBFEYE, which tries to capture the semantic information of such long sentences or even super-long sentences.

These transformations can usually be done with a few instructions. For example, as shown in the Figrue 8, the $addition$ transformation replaces the original code "$a = b+c$;" with "$r = rand();\ a = b + r;\ a = a + c;\ a = a - r$;".

```
mov  edx,  dword ptr[esp+8]
mov  eax,  dword ptr[esp+4]
add  eax,  edx
mov  dword ptr [esp+12], eax
mov  eax,  dword ptr[esp+12]
```

(a) Original

```
call  _rand
mov   dword ptr[esp],  eax
mov  edx,  dword ptr[esp+8]
mov  eax,  dword ptr[esp]
add  eax,  edx
mov  dword ptr[esp+12],  eax
mov  eax,  dword ptr[esp+4]
add  dword ptr [esp+12],  eax
mov  eax,   dword ptr[esp]
sub   dword ptr [esp+12],  eax
mov  eax,  dword ptr [esp+12]
```

(b) -sub-addition

Fig. 8. Example of $-sub-addition$. (a) is the original code of "$a = b+c$;", (b) is the transformed code of "$r = rand();\ a = b + r;\ a = a + c;\ a = a - r$;".

```
extern double pow(double,double);
extern void abort(void);

double foo (double x)
{
 return pow (x, 6);
}

double bar (double x)
{
 return pow (x, -4);
}

int main()
{
 if (foo (2.0) != 64.0)
   abort ();

 if (bar (2.0) != 0.0625)
   abort ();

 return 0;
}
```

(a) from Dataset2

```
#include<stdio.h>
int main(int argc, char* argv[]){

 int temp,i,j,a[10];

 for(i=1;i<argc;i++){
    a[i-1] = argv[i][0];
}

 for(i=argc-3;i>=0;i--){
    for(j=0;j<=i;j++){
       if(a[j]>a[j+1]){
          temp=a[j];
          a[j]=a[j+1];
          a[j+1]=temp;
       }
    }
 }

 printf("After sorting: ");
 for(i=0;i<argc-1;i++)
    printf(" %d",a[i]);

 return 0;
}
```

(b) from Dataset3

Fig. 9. Example of obfuscation points comparison. (a) is an original code in Dataset2(coming from $GNUtoolkit$). (b) is an original code in Dataset3(coming from $ObfusctionBenchmark$). There are only 2 obfuscation points in (a), but 5 in (b).

The number of instructions increases from 5 to 11. However, this weak context change is ignored when OBFEYE does feature embedding. The reason is that CNN uses Max Pooling operation when implementing the basic block embedding. Maximum pooling means to extract multiple eigenvalues from a specific filter and only retain the one with the largest value. In other words, it only retains the strongest feature and discards other weak features.

To avoid bias, we also test OBFEYE on Dataset3 which is a *Obfuscation Benchmarks* provided by TUM. The average accuracy of OBFEYE for single obfuscation detection reached 91.81%. From Table VI, we can see that the testing results on this dataset are better than Dataset2, since there are many obfuscation algorithms with TPR as high as 1.

TABLE VI
ACCURACY FOR SINGLE OBFUSCATION DETECTION. DATASET 2 IS SINGLE OBFUSCATION DATA SET BUILT BY US. DATASET 3 IS SINGLE *Obfuscation Benchmarks* FROM TUM.

Dataset		ACC	TPR								
			code	fla-o	sub	bcf	opa	fla-t	vir	ari	lit
2	OLLVM	0.8935	0.9420	0.9960	0.7604	0.9962	-	-	-	-	-
	Tigress	0.9038	0.9273	-	-	-	0.9737	0.9519	0.9434	0.8456	0.7915
	O+T	0.8846	0.8271	0.9960	0.7591	0.9962	0.9737	0.9519	0.9570	0.8401	0.8044
	Mean	0.8940	0.8988	0.9960	0.7596	0.9962	0.9737	0.9519	0.9502	0.8429	0.7980
3	OLLVM	0.9966	0.9886	1	0.9988	0.9992	-	-	-	-	-
	Tigress	0.8762	0.7546	-	-	-	1	1	1	1	0.8030
	O+T	0.8815	0.6037	1	0.9733	0.9990	1	1	1	1	0.8352
	Mean	0.9181	0.7823	1	0.9861	0.9991	1	1	1	1	0.8191

TABLE VII
ACCURACY FOR MULTIPLE OBFUSCATION DETECTION. DATASET5 IS A MULTIPLE OBFUSCATION DATASET BUILT BY US. DATASET6 IS MULTIPLE $ObfuscationBenchmarks$ FROM TUM.

Dataset	ACC	TPR		
		sub&bcf	sub&fla	bcf&fla
5	0.8279	0.9766	0.9156	0.9519
6	0.9784	0.9972	0.9991	0.9984

TABLE VIII
TRAINING TIME OF THE SEMANTIC-AWARE CONTEXT EMBEDDING MODEL IN OBFEYE WITH RESPECT TO DIFFERENT SIZE OF DATASETS.

Dataset		Training time(s)
1	OLLVM	294
	Tigress	485
	O+T	705
1+4	All	765

However, this result seems so beautiful that we have to find the real reason. We study the data sources of Dataset2 and Dataset3, respectively. The original code of Dataset3 comes from $ObfuscationBenchmark$. There are a total of 100 original programs in this benchmark, which are mainly classify into sorting algorithms, hashing algorithms, and arithmetic operation algorithms. These algorithms have a common feature, that is, they all contain a large number of obfuscation points, as shown in Figure 9(b). The original code of Dataset2 comes from $GNUtoolkit$, and there are a total of 2483 original programs in $GNUtoolkit$. But 80% of its original programs have only two or fewer obfuscation points as shown in Figure 9(a). The more obfuscation points mean that there are richer semantic information exists in the obfuscated programs, especially the obfuscation characteristics. Therefore, the TPR of the obfuscation detection on Dataset3 is higher.

In additional, the average TPR of $-lit$ is only 81.91%, which is almost the same reason as the $-sub$. In reality, $-sub$ and $-lit$ are weak code obfuscation transforms which usually used in conjunction with other code obfuscation algorithms. Therefore, we believe that even if OBFEYE has a weak ability to recognize such kind of obfuscation algorithm, it does not affect its use.

In general, Dataset2 and Dataset3 represent two different application scenarios of code obfuscation, namely universality and typicality. At the same time, They also cover a variety of obfuscation algorithms such as strong obfuscation and weak obfuscation. The experimental results show that OBFEYE has good performance on both of them.

3) Accuracy of Multiple Obfuscation: In reality, most obfuscated application scenarios will use multiple obfuscations. So, to verify the accuracy of OBFEYE in this scenario, we also perform a set of experiments on Dataset5 and Dataset6 where the programs are obfuscated with combinations of two obfuscation transformations with OLLVM. The experimental results are shown in the Table VII. The average accuracy is 82.79%, and the TPR of each type of multiple obfuscation detection also exceeded 90% when the testing on Dataset5. They are even higher to 97.84% and 99.82% when the testing on Dataset6. The reason for the high recognition rate of multiple obfuscations is that the implementation of multiple obfuscations requires more instructions and can provide more contextual semantic for the semantic neural network.

D. Efficiency of OBFEYE

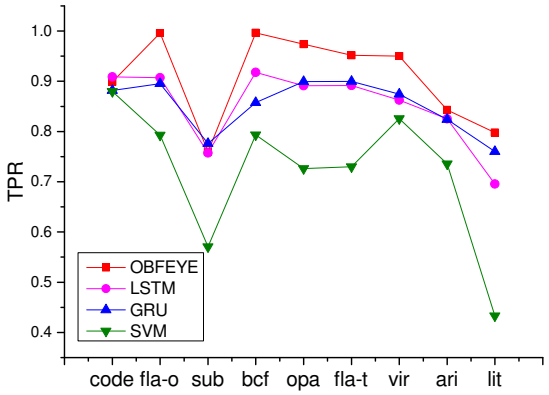
Training Time: We evaluate the time used for training the semantic-aware context embedding model in the OBFEYE, and we do four groups of training experiments. Table VIII is the training time of OBFEYE with respect to different sizes of datasets. It is found that the training time is linear to the sample size. Take Dataset1 as an example, when the data scales are 37194, 63063, and 100257, respectively, the training time spent is 294s, 485s, and 705s, respectively. In other words, it takes about 7ms to process a program during the training phase.

Testing Time: Although we do four groups of training experiments, we finally employ the fourth training model for the testing experiment. As shown in Table IX, we take multiple obfuscations on Dataset5 as an example. In every millisecond, there are 0.25 (7420/30000) programs, 10.16 (304838/30000) basic blocks, and 1578.06 (47341649/30000) instructions are deal with.

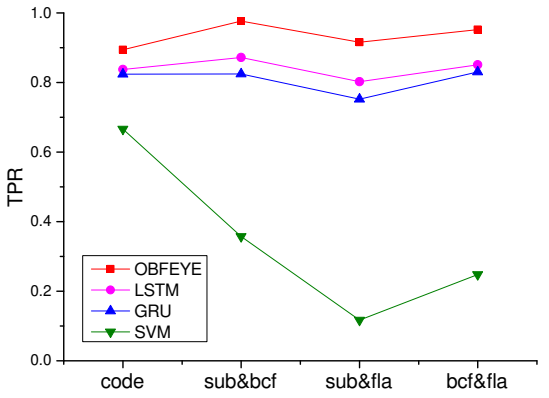
The timing measurements reported in the table highlight that the majority of execution time spent on training the OBFEYE model. Testing is high-speed, taking only a few seconds per

TABLE IX
TESTING TIME FOR BOTH SINGLE AND MULTIPLE OBFUSCATIONS.

Obfuscation	Dataset		Testing time(s)
Single	**2**	**OLLVM**	12
		Tigress	19
		O+T	27
	3	**OLLVM**	58
		Tigress	89
		O+T	146
Multiple	**5**	**All**	30
	6	**All**	153



(a) Single obfuscation.



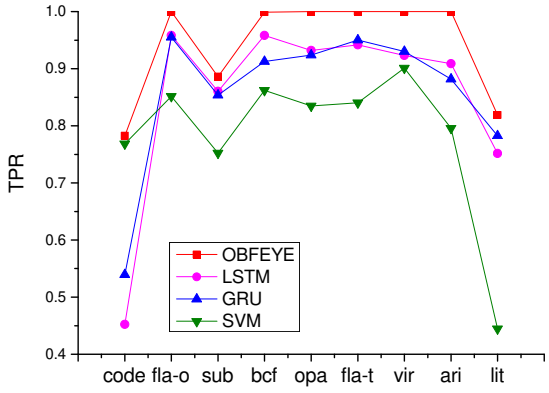
(b) Multiple obfuscation.

Fig. 10. The TPR results compared with other methods on our constructed datasets.

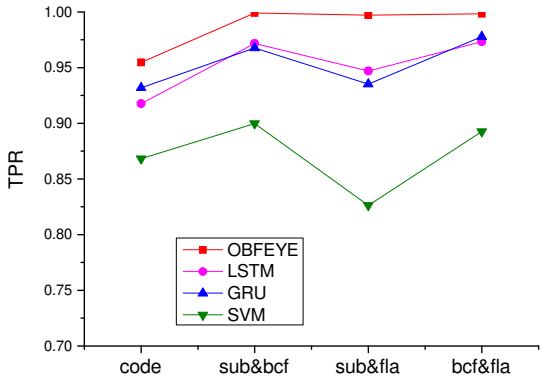
input. The execution time of our obfuscation detection is extremely low, taking only a couple of minutes for the entire set of datasets.

E. Comparison with other models

Since there is currently no deobfuscation detection for binary code, in this section, we only select several neural network models and a machine learning model commonly used in the field of binary code analysis for comparison. We are interested in LSTM [22], GRU(Gated Recurrent Unit) [9] and SVM(Support Vector Machine) [23]. In the LSTM and GRU models, we use word2vec to implement instruction embedding, which is taken as a classification feature. In the SVM



(a) Single obfuscation.



(b) Multiple obfuscation.

Fig. 11. The TPR results compared with other methods on $ObfuscationBenchmarks$.

model, we use TF-IDF(term Frequency - Inverse Document Frequency) as a feature to perform classification learning. In the same way, we still select 80% of Dataset1(shown in Table II) and Dataset4(shown in Table III) as the training datasets for LSTM, GRU, SVM to learn, and the other 20% for validation.

In the testing phase, we select Dataset2 and Dataset5 as the detecting data set, which includes single obfuscation and multiple obfuscations. The experimental results show that OBFEYE has the highest obfuscation detection accuracy, with 92.81%, followed by LSTM with 82.43%. And SVM is the worst with 45.32%, as expected.

For further investigation, we put the TPR of each model on single obfuscation and multiple obfuscations in Figure 10. From Figure 10(a), we can see that in addition to the two code obfuscation algorithms of $-sub$ and $-lit$, OBFEYE's TPR is significantly better than other models. However, as the figure shown, all models are performing poorly for this type of weak code obfuscation. That is because their features are not obvious, resulting in little adequate information available to the detection model. To some extent, this also proves that the quality of feature selection directly determines the detection result.

From Figure 10(b), it can be seen that the semantic neural network model in OBFEYE is more powerful than other models on multiple obfuscations. The reason is that it takes

into account the advantages of CNN and LSTM, and obtains much more semantic features of obfuscation algorithm. It also proves that the neural network model based on full program semantics is suitable for binary code obfuscation detection. Because this approach provides more semantic information for the neural network model. Therefore, OBFEYE is a feasible code obfuscation detection scheme.

To avoid bias, we still test OBFEYE, LSTM, GRU, and SVM on $ObfuscationBenchmarks$ provided by TUM. Figure 11 is the experimental results. It can be seen from the figure that the TPR of OBFEYE is still the best among the four models regardless of single obfuscation or multiple obfuscations. Compared with testing on Dataset2 and Dataset5, the TPRs of all models on the $ObfuscationBenchmarks$ datasets have improved. The reason is the same as we analyze in Section VI-C. That is because there are many more obfuscation points in the source codes of $ObfuscationBenchmarks$. These obfuscation points require more assembly instructions to implement, which provide more contextual semantic information for the classification model.

In addition, from the figure, it is not difficult to find that all the TPRs of each model for the samples without adding obfuscated code decreased significantly. That is because the source codes of $ObfuscationBenchmarks$ contain a large number of if statements and for statements, which can easily be mistaken for obfuscated code fragments by various classification models, resulting in high TPRs.

In a word, all the experimental results show that OBFEYE is highly effective in identifying the obfuscation scheme, with a prediction accuracy of at least 83% (up to 98%). Besides, the experiments successfully demonstrate that it is promising to approach binary analysis from the angle of language processing by adapting methodologies, ideas, and techniques in NLP.

VII. Limitations

Code obfuscation has proved to be a stumbling block for various virus detections and similarity detections. The work of this paper not only promotes the development of deobfuscation itself but also facilitates the development of the following area, such as virus detection [42], [7], clone code detection [45], binary code similarity detection [65], third-party libraries detection [31], etc.

Besides, there are some limitations to our work. The present design of OBFEYE is limited to dealing with obfuscation detection on assembly language generated by reverse analysis tools such as IDA. That is because the training samples with assembly language of the Intel style used in the neural network learning process are constructed by the compiler gcc. In future work, we will increase the compatibility of OBFEYE to meet the needs of obfuscation detection on different kinds of coding languages.

The current design of OBFEYE is not suitable for detecting data flow obfuscation, since it only analyzes the semantic information of the target program, without considering the data dependencies. In the future, we intend to also take the data flow as a classification feature of a neural network model.

At present, the data set we constructed only includes two obfuscation tools of OLLVM and Tigress, with a total of eight single obfuscations and six multiple obfuscations. However, in reality, there are many kinds of obfuscation tools that have not been considered, and we will add them in subsequent work. Of course, with the continuous development of deep learning, we can also try to use other technologies to solve training and testing on small-scale data sets.

VIII. Related Work

Deep learning and language modeling: In recent years, deep learning has flourished in different research fields, especially in the tasks of image recognition and natural language processing. Relying on the similarity between program code and natural language, deep models in natural language processing have been gradually applied to program understanding and code analysis. For example, DREBIN [1] performs a comprehensive static analysis, embedding many features from an applications code and manifest in a joint vector space to detect Android Malware. White et al. [58] proposed a deep learning-based detection method for source code clone detection using RNN. Zuo et al. [65] proposed a deep learning-based detection for code similarity detection by using LSTM. EKLAVYA [10] trains an RNN model to recover function type signatures from disassembled binary code. As far as we know, OBFEYE is the first to apply deep learning models to code obfuscation detection, and it is also the first to build a code obfuscation detection model on binary code with taking advantage of contextual semantic information.

Obfuscation detector with Machine learning: Currently, a large number of code obfuscation detections focus on scripting languages, such as Javascript [34], [24], [47] and Powershell [32], [17]. They leveraged machine learning and data mining to detect JavaScript or Powershell code obfuscation. NOFUS[27] takes a context feature from the abstract syntax tree of the script and uses a Bayesian-based detector to detect confusion. JSObfusDetector [24] takes the number of string variables and the number of dynamic functions in JavaScript scripts as features, and uses a One-class SVM algorithm to identify malicious JavaScript obfuscated scripts. But there is almost no obfuscation detection for binary code. That is because, compared with source code and scripting languages, binary code lacks semantic information such as symbol information, variable type information, software structure descriptions, function libraries, etc. Therefore, the existing token-based detection methods are difficult to transplant to binary detection. OBFEYE takes advantage of the character that deep learning uses supervised learning for feature extraction and embedding. It uses massive training data to replace people's observation of the code so that the algorithm can summarize the nature and rules of the obfuscated code, and then improve the accuracy and efficiency of the detection model.

IX. Conclusion

In this paper, we introduced a semantic-aware obfuscation detection approach that achieves a detection rate of 89.4% for single obfuscation and 82.79% for multiple obfuscations

over an experimental dataset of 277,131 assembly documents. Further, we also test OBFEYE on Obfuscation Benchmarks provided by Technische Universität München(TUM, Germany). The accuracy rate of the single obfuscation test is 91.81%, and the accuracy of multiple obfuscations is as high as 97.84%. Additionally, we have shown that our approach requires modest computation to perform feature extraction and that it can achieve good accuracy over our corpus on a single CPU within modest timeframes. In particular, we constructed datasets containing 277,131 obfuscation file with assembly codes, eight single obfuscation algorithms, and six multiple obfuscation algorithms, which is valuable for other researchers who are dedicated to studying code obfuscation.

REFERENCES

[1] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., RIECK, K., AND SIEMENS, C. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss* (2014), vol. 14, pp. 23–26.

[2] BANESCU, S., COLLBERG, C., AND PRETSCHNER, A. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 661–678.

[3] BANESCU, S.-E. *Characterizing the Strength of Software Obfuscation Against Automated Attacks*. PhD thesis, Technische Universität München, 2017.

[4] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference* (2001), Springer, pp. 1–18.

[5] BAZZI, I. *Modelling out-of-vocabulary words for robust speech recognition*. PhD thesis, Massachusetts Institute of Technology, 2002.

[6] BICHSEL, B., RAYCHEV, V., TSANKOV, P., AND VECHEV, M. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 343–355.

[7] CAI, H., MENG, N., RYDER, B., AND YAO, D. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security 14*, 6 (2018), 1455–1470.

[8] CHEN, H., YUAN, L., WU, X., ZANG, B., HUANG, B., AND YEW, P.-C. Control flow obfuscation with information flow tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), pp. 391–400.

[9] CHO, K., VAN MERRIËNBOER, B., BAHDANAU, D., AND BENGIO, Y. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation* (Doha, Qatar, Oct. 2014), Association for Computational Linguistics, pp. 103–111.

[10] CHUA, Z. L., SHEN, S., SAXENA, P., AND LIANG, Z. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 99–116.

[11] COLLBERG, C., MARTIN, S., MYERS, J., ZIMMERMAN, B., KRAJCA, P., KERNEIS, G., DEBRAY, S., AND YADEGARI, B. The tigress c diversifier/obfuscator. *Retrieved August 14* (2015), 2015.

[12] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations, 1997.

[13] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1998), pp. 184–196.

[14] COLLBERG C. S., MARTIN S., M. J. E. A. The tigress diversifying c virtualizer, 2018. https://tigress.wtf/.

[15] CONTINELLA, A., FRATANTONIO, Y., LINDORFER, M., PUCCETTI, A., ZAND, A., KRUEGEL, C., AND VIGNA, G. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *NDSS* (2017).

[16] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 275–284.

[17] DANIEL BOHANNON, L. H. Powershell obfuscation frameworks, 2017. https://github.com/danielbohannon/Revoke-Obfuscation.

[18] GITHUB. obfuscator-llvm, 2017. https://github.com/obfuscator-llvm/obfuscator/wiki.

[19] GLANZ, L., AMANN, S., EICHBERG, M., REIF, M., HERMANN, B., LERCH, J., AND MEZINI, M. Codematch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting*

compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.

[41] MING, J., XU, D., WANG, L., AND WU, D. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 757–768.

[42] MURAD, K., SHIRAZI, S. N.-U.-H., ZIKRIA, Y. B., AND IKRAM, N. Evading virus detection using code obfuscation. In *International Conference on Future Generation Information Technology* (2010), Springer, pp. 394–401.

[43] PENDLETON, M., GARCIA-LEBRON, R., CHO, J.-H., AND XU, S. A survey on systems security metrics. *ACM Computing Surveys (CSUR) 49*, 4 (2016), 1–35.

[44] SALWAN, J., BARDIN, S., AND POTET, M.-L. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2018), Springer, pp. 372–392.

[45] SCHULZE, S., AND MEYER, D. On the robustness of clone detection to code obfuscation. In *Proceedings of the 7th international workshop on software clones* (2013), IEEE Press, pp. 62–68.

[46] SOFTWARE, V. Vmprotect software protection. http://vmpsoft.com, year = 2017,.

[47] SU, J., YOSHIOKA, K., SHIKATA, J., AND MATSUMOTO, T. An efficient method for detecting obfuscated suspicious javascript based on text pattern analysis. In *Proceedings of the 2016 ACM International on Workshop on Traffic Measurements for Cybersecurity* (2016), pp. 3–11.

[48] TECHNOLOGIES, O. Code virtualizer: Total obfuscation against reverse engineering, 2017. http://oreans.com/codevirtualizer.php.

[49] TECHNOLOGIES, O. Themida: Advanced windows software protection system, 2017. https://www.oreans.com/themida.php.

[50] TECHNOLOGY, S. Execryptor: Bulletproof software protection. http://www.strongbit.com/execryptor.asp, year = 2017,.

[51] TUM, I. . obfuscation-benchmarks, 2018. https://github.com/tum-i22/obfuscation-benchmarks.

[52] UDUPA, S. K., DEBRAY, S. K., AND MADOU, M. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)* (2005), IEEE, pp. 10–pp.

[53] WANG, C., DAVIDSON, J., HILL, J., AND KNIGHT, J. Protection of software-based survivability mechanisms. In *2001 International Conference on Dependable Systems and Networks* (2001), IEEE, pp. 193–202.

[54] WANG, C., HILL, J., KNIGHT, J., AND DAVIDSON, J. Software tamper resistance: Obstructing static analysis of programs. Tech. rep., Technical Report CS-2000-12, University of Virginia, 12 2000, 2000.

[55] WANG, P., WANG, S., MING, J., JIANG, Y., AND WU, D. Translingual obfuscation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 128–144.

[56] WANG, W., WANG, Z., LI, M., TANG, Z., AND FANG, D. Invalidating analysis knowledge for code virtualization protection through partition diversity. *IEEE Access PP*, 99 (2019), 1–1.

[57] WANG, Y., WU, Z., WEI, Q., AND WANG, Q. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access 7* (2019), 36340–36352.

[58] WHITE, M., TUFANO, M., VENDOME, C., AND POSHYVANYK, D. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2016), IEEE, pp. 87–98.

[59] XU, D., MING, J., FU, Y., AND WU, D. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 442–458.

[60] XUE, H., VENKATARAMANI, G., AND LAN, T. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation* (2018), pp. 27–33.

[61] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 674–691.

[62] YOUNG, S. R. Detecting misrecognitions and out-of-vocabulary words. In *Proceedings of ICASSP'94. IEEE International Conference on Acoustics, Speech and Signal Processing* (1994), vol. 2, IEEE, pp. II–21.

[63] ZHAO, Y., TANG, Z., YE, G., PENG, D., FANG, D., CHEN, X., AND WANG, Z. Compile-time code virtualization for android applications. *Computers & Security* (2020), 101821.

[64] ZHOU, J., ZHANG, H., AND LO, D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 14–24.

[65] ZUO, F., LI, X., YOUNG, P., LUO, L., ZENG, Q., AND ZHANG, Z. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).