Fine-Grained Obfuscation Scheme Recognition on Binary Code

Zhenzhou Tian[1,2](✉), Hengchao Mao[1,2], Yaqian Huang[1,2], Jie Tian[1,2], and Jinrui Li[1,2]

[1] School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China
tianzhenzhou@xupt.edu.cn

[2] Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an, China

Abstract. Code obfuscation is to change program characteristics through code transformation, so as to avoid detection by virus scanners or prevent security analysts from performing reverse analysis. This paper proposes a new method of extracting from functions their reduced shortest paths (RSP), through path search and abstraction, to identify functions in a more fine-grained manner. The method of deep representation learning is utilized to identify whether the binary code is obfuscated and the specific obfuscation algorithms used. In order to evaluate the performance of the model, a data set of 60,000 obfuscation samples is constructed. The extensive experimental evaluation results show that the model can successfully identify the characteristics of code obfuscation. The accuracy for the task of identifying whether the code is obfuscated reaches 98.6%, while the accuracy for the task of identifying the specific obfuscation algorithm performed reaches 97.6%.

Keywords: Code obfuscation recognition · Binary code · Neural network

1 Introduction

Code obfuscation is a widely used software protection technique that can mitigates the risks caused by reverse engineering. It helps to protect software intellectual property by hiding the logic and sensitive data implied in the code. The

This work was supported in part by the Science and Technology of Xi'an (2019218114GXRC017CG018-GXYD17.16), the National Natural Science Foundation of China (61702414), the Natural Science Basic Research Program of Shaanxi (2018JQ6078, 2020GY-010, 2020JM-582), the International Science and Technology Cooperation Program of Shaanxi (2018KW-049, 2019KW-008), the Key Research and Development Program of Shaanxi (2019ZDLGY07-08), and the Special Funds for Construction of Key Disciplines in Universities in Shaanxi.

use of code obfuscation depends on the sensitivity of the application. Its applications are mainly on digital rights management, software licensing and white box encryption. Malicious code also makes extensive use of code obfuscation to hide its intentions, so as to evade detection and hinder analysis. Therefore, the problem of de-obfuscation has attracted widespread attention in the academic community, many researchers have attempted to recover the original code from obfuscated programs, while identifying the specific obfuscation algorithm [22] applied facilitates to a large extent the de-obfuscation process.

At present, most de-obfuscation techniques focus on the topic of automatically processing the obfuscated code to restore the original code. Generally, they only work on certain obfuscation algorithms. Some of existing approaches, including layout de-obfuscation [2], opaque predicate de-obfuscation [13], control leveling obfuscation [17] and virtualization de-obfuscation [4,15,20], all work under the premise of knowing the specific obfuscation algorithms used. In reality, researchers often face completely unknown malware in the form of executable code, which leads to two closely related problems. The first question, from the perspective of de-obfuscation, is whether the target program is obfuscated? For example, if the existing de-obfuscation techniques are used to analyze the target program that does not contain obfuscated code, not only the internal logic of the original program will be broken, it also causes the analysts to spend a lot of time doing useless work. The second question is, from the perspective of reverse engineering, what is the specific kind of obfuscation algorithm used to produce to target obfuscated program? Especially, it is worth mentioning that in recent years, new obfuscation algorithms have emerged one after another. Obviously, it is very necessary for analysts to understand the characteristics of each code obfuscation and quickly identify the obfuscation algorithms from the target program, which once can be identified in an automated way, will greatly reduce the difficulty of reverse analysis and time cost for security analysts.

In recent years, tremendous successes have been witnessed of applying deep learning models to diverse program analysis tasks, such as binary code clone detection [21], compiler provenance analysis [16] and vulnerability detection [19]. They generally leverage the many layers of non-linearities to automatically boost learning a latent vector representation with semantic richness. In this regard, this paper proposes to take each individual function within a binary program as the basic analysis subject. After processing a function into a set of reduced shortest paths, our method operates directly on the corresponding normalized assembly instructions to achieve fine grained obfuscation detection. Also, from the perspective of sequence analysis and structural analysis, we design a supervised learning model based on deep neural networks to achieve code obfuscation detection and obfuscation algorithm identification. Our main contributions are summarized as following:

– We propose a new form of function representation as a set of reduced shortest paths (RSP), through path search and abstraction on its control flow graph (CFG), to get the function represented in a more fine-grained and semantics-aware manner.

- We suggest to perform fine-grained binary code obfuscation recognition for each individual function by designing a lightweight function abstraction strategy and a deep representation learning model based on RNN and CNN. It reduces the impact of task complexity and human bias by handing over the important feature extraction and selection process of the function to less human intervened neural networks.
- We have implemented a prototype tool called OBDB (OBfuscation scheme Detector on Binary code), which integrates our proposed method. A dataset consisting of more than 60,000 samples is constructed by processing 11,000 programs collected from Google Code Jam (GCJ), with two of the most well-known code obfuscators including Tigress [1] and OLLVM [8]. The experimental evaluation results conducted on the dataset show that, the proposed method can effectively capture the significant features of specific code obfuscations. OBDB achieves rather good performance, with the accuracy of identifying whether the code is obfuscated reaches 98.6% and the accuracy of identifying the specific obfuscation algorithm reaches 97.6%.

2 Background

Code obfuscation is a technique that enforce control and data transformations on the program's source code or even its binary code while retaining the functionality of the original program, with the aim of making it more difficult to analyze and tamper with. Collberg et al. [3] makes a general taxonomy of obfuscating transformations. According to underlying basic schemes, obfuscation transformations can generally be divided into four categories: layout obfuscation, data obfuscation, control obfuscation and prevention obfuscation. So far, the research of code obfuscation algorithms has been relatively mature, various algorithms have emerged in recent years. Table 1 shows six typical code obfuscation algorithms supported in two widely used obfuscation tools, on the basis of which many other obfuscation algorithms have been derived.

Table 1. Six typical obfuscation algorithms

Tool	Obfuscation algorithm	Description
OLLVM	Instructions substitution (sub)	Replace binary operators including addition, subtraction and boolean operators
	Bogus Control Flow (bcf)	Add opaque predicates to make a conditional jump to the original basic block
	Control Flow Flattening (fla)	Break down the program's control flow
Tigress	Virtualize (vir)	Replace code with virtualized instructions and execute them by an interpreter
	AddOpaque (opa)	Add opaque predicates to split up the control flow of the code
	EncodeLiterals (lit)	Replace literal integers and strings with less obvious expressions

3 Overview of Our Approach

In this section, we will introduce the proposed method in detail. The overview of OBDB is shown in Fig. 1, which consists of the training and the detection phases. In the training phase, firstly a set of reduced shortest paths RSP are extracted and normalized to get each function expressed. To be specific, IDA-Pro [5] is used to analyze each function in the training set to obtain its control flow graph (CFG), on the basis of which all the shortest paths are extracted from the entry basic block to each other basic block in the CFG to obtain a path set SP. By checking the inclusion relationship between the paths in SP, a set of reduced shortest paths RSP can be generated. That is, for each path $p \in SP$, as long as it is not completely contained by all the other paths in SP, it will be considered as a reduced shortest path and added to the set RSP. With a light-weighted assembly instruction normalization scheme, each path in RSP is further abstracted to get rid of inessential details.

The collection of abstracted paths form the basic representation of a function, which are fed as the inputs to the neural network module to extract latent feature vectors that are indicative of the obfuscation algorithms. The whole model is finally trained by appending a dense and a softmax layer to process the latent vectors together with their ground truth labels as supervision. The detection phase is much simpler, which reads in an individual function, processes it with the function abstraction and utilizes the trained model to produce predictions.

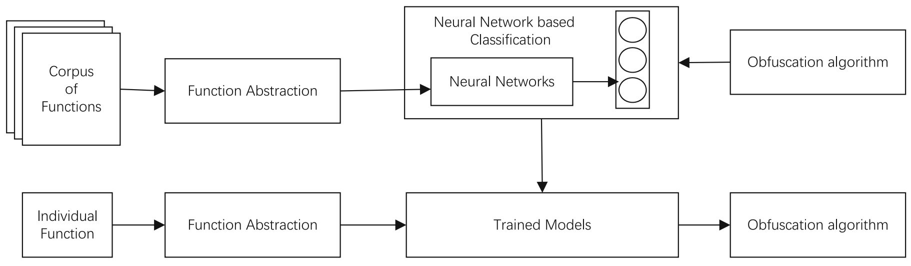

Fig. 1. The basic framework of the OBDB model

3.1 Function Abstraction

Reduced Shortest Path Extraction. A function must be expressed in a certain form such that it can be further analyzed. Typical methods include the use of raw byte sequences, assembly instruction sequences, or control flow graph to describe the function. As shown in many binary analysis tasks, adopting the original raw bytes has been proven to be an unwise choice [16]; while using the assembly code as a whole to describe the function, generally results in the loss of expressive structural information, which may be of significant importance to identify the features manifested in the obfuscated code by the obfuscation algorithms. To this end, we choose to use the CFG that implicitly encodes the possible execution paths as the basic representation form for each function. In particular, we retrieve and construct from each function's CFG all the reduced shortest paths, and use the assembly instructions that appear along these paths to get it represented. It ensures that, at the time of capturing instruction level obfuscation indicative features, the structural features can also be covered. Algorithm 1 gives the pseudo-code that converts the function into the assembly instruction sequences.

Algorithm 1. Reduced shortest path based function representation

```
Input:
    $G$: control flow graph of a function
    $I_B$: instruction set within each basic block
    $T$: the number of basic blocks
Output:
    $RSP$: the reduced shortest path representation of the input function
 1: $SP \leftarrow \langle\ \rangle$
 2: $path \leftarrow \langle\rangle$
 3: $G_{acyclic}$ = clear_garph($G_d$) ▷ clear the loops in $G$ to be a directed acyclic graph
 4: $E = get_entry(G)$ ▷ get the entry node of the CFG
 5: for each node $B$ in $G_{acyclic}$ do
 6:     if $E$ is exactly $B$ then
 7:         continue
 8:         $path$ =Dijkstra($G_{acyclic}, E, BB$) ▷ using Dijkstra to find the shortest path
 9:         $SP = SP \oplus path$ ▷ Add a new shortest path to the set
10:     end if
11: end for
12: for each $p$ in $SP$ do
13:     if containedBy($p, SP$) then
14:         $RSP = RSP \oplus p$ ▷ Add $p$ to the reduced shortest path set
15:     end if
16: end for
```

Instruction Normalization. After obtaining the instruction sequence of the function, the instruction consists of an opcode (i.e. the mnemonic) and a list of operands. It is usually unwise to process the original instructions directly. In our case, we want to capture the characteristics of the obfuscated algorithm, rather than the functionality of the function. In other words, we don't care whether the value 6 is assigned to the register eax or the value 10 is assigned to the register eax, but we are more concerned with the form and type of instructions chosen by the obfuscation algorithm. In this regard, by considering that these two instructions "mov eax, 6" and "mov eax, 10" are the same may be better choice to our problem. In addition, the memory addresses (such as the target address of the jmp instruction) are meaningless, they tend to be noises that distract the attention of successive neural network based training process. In addition, in order to avoid introducing too many human prejudices, we choose to use a lightweight abstraction strategy to process the original assembly instruction sequences in RSP. To be specific, we have formulated the following instruction abstraction rules:

– All the opcode of instructions remain unchanged.
– All registers in the operand remain unchanged.
– All base memory addresses in the operand are replaced with the symbol MEM.
– All immediate values in the operand are replaced with the symbol MEM.

For example, using the above normalization rules, the instruction "add eax, 6" will become "add eax, IMM", the instruction "mov ebx, add eax, 6" will become "mov ebx, MEM", and the instruction "mov ebx, [ebp−20]" will become "mov ebx,[ebp−IMM]".

3.2 Instruction Embedding

Before using the neural network based learning model to detect obfuscation algorithms, we must transform the normalized assembly instruction sequences into numerical vectors, such that they can be used as input to subsequent classifiers. As our designs choose to use advanced deep neural network to master the subtle features and patterns indicative of the applied obfuscation algorithms, we firstly utilize the widely adopted word embedding to assign a vector to each unique normalized instruction, based on which the whole instruction sequence can also be represented as a vector sequence.

There are several word embedding choices that can be leveraged for our application, of which the one-hot encoding has been widely deployed. It represents each unique word by a n-dimensional vector, with the i^{th} dimension being set to 1 and all other dimensions being set to 0, where i is the index of the word in the vocabulary of size n. This technique is computationally intractable as the generated vectors are too sparse (the same dimension as the size of the whole vocabulary) and generally needs to do joint-learning with subsequent neural networks, making the learnt word semantics significantly task-specific. In this respect, OBDB leverages the popular skip-gram model [12] to learn more compact vector representations that carry instruction co-occurrence relationships and lexical semantics in an independent and unsupervised manner, so as to make the learnt vectors reusable in other binary analysis tasks. Specifically, we treat each basic block as a sentence and each abstracted instruction within the basic block as a word, and feed all the basic blocks from our binary collection to the skip-gram model to learn for each unique normalized instruction a d-dimensional vector, by minimizing the loss of observing an instruction's neighborhood (within a window w) conditioned on its current embedding. The objective function of skip-gram can be defined as [14]:

$$arg\,\min_{\Phi} \sum_{-w \leq j \leq w, i \neq j} -log\ p\left(e_{i+j} | \Phi(e_i)\right) \tag{1}$$

3.3 The BiGRU-CNN Model

Based on the instruction embedding learned with skip-gram, different schemes such as maximum pooling, average pooling or cascading can be exploited to aggregate the embeddings of each abstracted instruction sequence, and then feed it to any classification model for obfuscation algorithm identification. However, it still faces the following two limitations: (1) skip-gram assigns a static embedding vector to each instruction, and it does not know the context of the different sequences it interacts with; it may not be able to learn obfuscation-related features; (2) as instruction sequences are abstracted from functions, they may not only enjoy local instructions. In this regard, it needs sequence learning model to better capture representative obfuscation algorithm specific patterns and features from instruction sequences, so as to recognize the specific obfuscation algorithms applied. As advanced neural network structures, either RNN or CNN based models have ever been applied for representation learning sequences in NLP tasks. Therefore, in this work, OBDB attempts to combine RNN and CNN structures to learn the syntactic and structural information implied in the instruction sequences, so as to use both their advances to identify obfuscation algorithms. Figure 2 depicts the basic structure of our BiGRU-CNN based model.

Specifically, it firstly use a RNN-based layer to iteratively process each normalized reduced shortest path p_i in RSP into a numerical vector. RNN is chosen in our design for its ability of capturing long range dependencies between the normalized opcodes in a sequence, so as to capture the obfuscation indicative features. Yet, the naive RNN structure exposes the vanishing/exploding gradient issue in handling long sequences, while two improved structures LSTM and GRU have been proposed to alleviate the problem. Also, considering that GRU has a simpler structure and is generally believed to be more efficient than LSTM, we choose to use GRU in the current design.

The GRU unit reads in the input instruction sequence p_i through the hidden layer function $\mathcal{H}$ to generate a hidden vector state $\mathbf{h}_i$ at each timestep i. To improve the learning ability, OBDB further devises the bidirectional GRU (BiRGU) structure to jointly capture both the forward and backward sequential dependency and increase the amount of information available to the network. The hidden state vector h_i at timestep i can then concatenated as:

$$h_i = \left[\overrightarrow{h_i}; \overleftarrow{h_i}\right]. \tag{2}$$

After bidirectionally reading the entire input sequence, the hidden states $\mathbf{h}_t$ corresponding to the last timestep will act as the latent vector representation of the input sequence. After processing all the $p_i \in RSP$, all the encoded numerical vectors will be formed as a numerical matrix A, which is to be fed into a CNN based layer to further learn a latent vector representation.

The convolution layer can extract local contextual features with varying convolution kernels. In OBDB, k different filters with shape $n \times d$ are adopted for convolution operations on matrix A to obtain a new feature matrix $C \in \mathbb{R}^{(l-n+1)\times k}$, where n denotes the kernel size. To capture different aspects of features, we convolute A by varying kernels of size 2, 3 and 4 respectively. To reduce the dimensionality of learnt vectors and get OBDB focus on significant features, a pooling layer is applied that performs pooling operations on the produced convolutional features. There are usually two types, max-pooling and average-pooling, that are widely used. Maximum pooling is to select the maximum value in the vector after convolution operation as the local feature, while average pooling is to use the average value. In this paper, we choose maximum pooling. To prevent the neural networks from overfitting during the training phase, a dropout layer is also appended right after the CNN layer.

Finally, the output of the dropout layer is fed as inputs to subsequent dense and classification layers for predication. The classification layer is basically sigmod or softmax layer. Sigmoid is mainly used for binary classifications, and softmax can be used for multi-classifications. Specifically, we use the softmax function to achieve multi-classification. Simply put, the softmax function maps some output neurons to real numbers between 0 and 1, such that the sum of the probabilities of multi-classifications is exactly 1. The softmax function is defined as follows:

$$P_i = \frac{e^{V_i}}{\sum_{i-1}^{C} e^{V_i}} \tag{3}$$

where V_i is the output of the previous unit of the classifier, i denotes the class index, and the total number of classes is C.

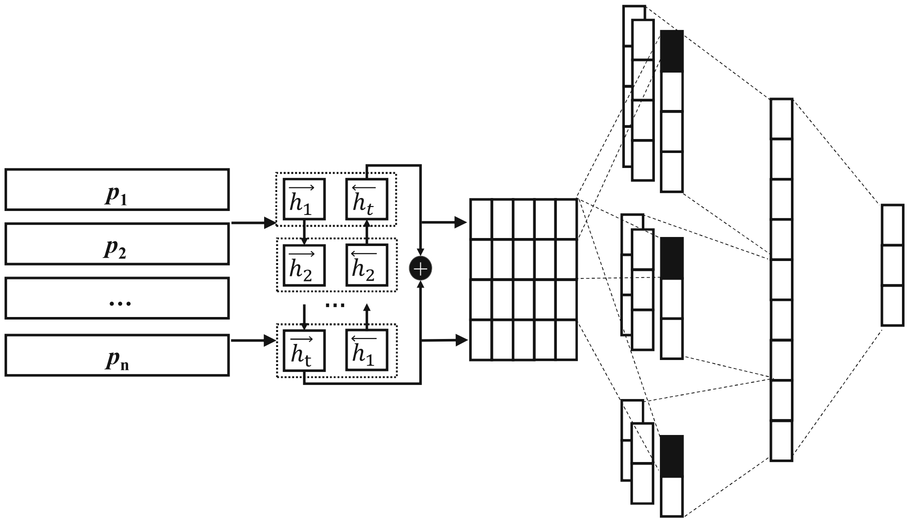

Fig. 2. BiGRU-CNN model structure diagram

4 Experiments and Evaluation

4.1 Dataset Construction

In the field of code analysis, publicly available labeled datasets for binary code are scarce, let alone datasets with obfuscation labels. In order to evaluate the performance of OBDB, we collected a large number of open source programs written in C programming language from the GCJ as a basis to build a data set. Specifically, we deal with these collected programs with the following steps:

(1) Firstly, we utilize the six typical obfuscation algorithms implemented in two widely use obfuscation tools to enforce obfuscation on the source code of each program. The specific obfuscation algorithms applied include the -bcf, -fla and -sub options in OLLVM, as well as the -AddOpaque, -Virtualize and -EncodeLiterals options in Tigress. For programs that are successfully obfuscated, the gcc compiler is then used to compile their obfuscated source code into binaries.
(2) Secondly, IDA Pro is used to identify and extract functions from each binary file. In addition, we removed some insignificant functions (functions that contain only several instructions, such as the stub functions), which are meaningless to analysis. In the current setting, we consider functions that contain less than 10 instructions to be trivial, which will not be considered by the dataset.
(3) Further, to prevent the neural network model from seeing in the training phase functions that are very similar to those in the testing phase, only distinct function are retained in the final dataset. Specifically, if a function has exactly the same normalized assembly instructions as any other function in the raw dataset, it will be considered as redundant. With these settings, we finally constructed a dataset consisting of 60,000 distinct and non-trivial obfuscated functions.

4.2 Implementation and Experimental Setup

We have implemented OBDB as a prototype tool. It uses ida pro to parse binary files to obtain functions and their original assembly instructions. The function abstraction module is implemented in Java programming language, while the neural network based representation learning and classification module is implemented with Python and the Tensorflow framework.

In the experimental settings, we randomly divide the entire dataset into training set, validation set and test set at a ratio of 80%, 10%, and 10%. The neural network model is trained using a Tesla V100 GPU with the Adam optimizer. The batch size is set to 128, and the initial learning rate is set to 0.001 (when the loss on the validation set stops improving for at least 5 epochs, the learning rate will be further divided by 10). In each epoch, the training samples are shuffled, the accuracy on the validation set is calculated. Besides, to avoid problems such as over-fitting and non-convergence, the early stopping mechanism is enforced to stop the training right after the epoch that the validation accuracy no longer improves. Finally, the model with the highest accuracy is considered as the finally trained obfuscation classifier, with which frequently used performance metrics including accuracy, precision, recall, and f1-score are evaluated and reported on the test set.

4.3 Experimental Results

In the following, we evaluate the performance of OBDB in identifying the existence of obfuscation and the specific obfuscation algorithms respectively. In addition, several other widely used deep neural network models have also been implemented and get compared with OBDB. Note that the precision, recall and f1-score in Table 2 all refer to the weighted average precision, recall and f1-score, respectively.

(1) Performance evaluation on identifying the existence of obfuscation
In this experiment, we take the obfuscated/non-obfuscated label of each function as the ground-truth. That is, all the functions obfuscated with either of the six obfuscation algorithms are marked with the obfuscated label. Then, OBDB is trained and evaluated according to the experimental settings in Sect, 4.2. As shown in Table 2 the evaluation results, OBDB outperforms all the other comparison models with respect to the performance metrics, exhibiting a rather good accuracy of 98.6% and a high f1-score of 0.986.

Table 2. Detection results for the existence of obfuscation

Model	Accuracy	Precision	Recall	F1
CNN	95.78%	0.9576	0.9678	0.9598
BiLSTM	93.65%	0.9395	0.9368	0.9365
BiGRU	94.98%	0.9458	0.9378	0.9488
OBDB	**98.62%**	**0.9794**	**0.9889**	**0.9861**

(2) Performance evaluation on identifying the specific obfuscation algorithms
In this experiment, the specific obfuscation algorithms applied on the functions are taken as the ground-truth labels to get OBDB trained and evaluated. That is, OBDB attempts to assign to each obfuscated function a certain label that indicates one of six obfuscation algorithms. According to the detection results, the total accuracy of OBDB reaches 97.6%, which is a bit lower than its accuracy reported on the task of identifying the existence of obfuscation. Besides, we compare it with Zhao's [23] method, which also designs a deep neural network based model to train classifiers for obfuscation scheme recognition. As reported in their work, the detection accuracy on recognizing the specific obfuscation algorithms reaches 89.4%, which is much lower than ours'. The about 9% performance gains achieved with OBDB than Zhao's

method may lie in that, OBDB adopts a more carefully designed function representation (a set of reduced shortest paths) that captures both syntactic and structural changes enforced by the obfuscation algorithms, while their method simply represent the function as a set of basic blocks. It indicates the superiority of our proposed novel function representation method in this task.

Figure 3 summarizes the true positive rates (TPRs) with respect to each obfuscation algorithm. As it shows, the TPRs vary across different obfuscation algorithms, indicating the different impacts enforced on the produced obfuscated code by different obfuscation algorithms. To be specific, the FPR for the -fla obfuscation in OLLVM reaches the highest, which is about 99.89%; while the FPRs for -lit and -sub obfuscations are the lowest, which are 94.63% and 94.65% respectively. As introduced in Table 1, the -fat option performs the control flow flattening obfuscation, which makes great changes to the control flow of the program. Our normalized reduced shortest path representation of functions that well captured the structural features just boosts the FPR on this obfuscation algorithm. Similarly, the reasons for the relatively lower FPRs of -lit and -sub lie in that, the changes introduced by these two kind obfuscation schemes which replace literals or strings are not that obvious as other obfuscation algorithms. In spite of that, the fairish FPRs indicate that OBDB still captures the subtle features indicative of these obfuscations. The TPRs for the other neural networks models are also evaluated and depicted in Fig. 4. As the data shows, OBDB still performs the best among all the models.

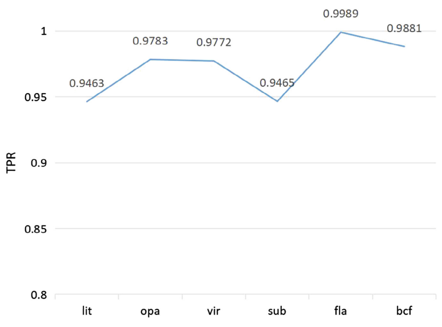

Fig. 3. TPRs in terms of diffent obfuscation algorithms

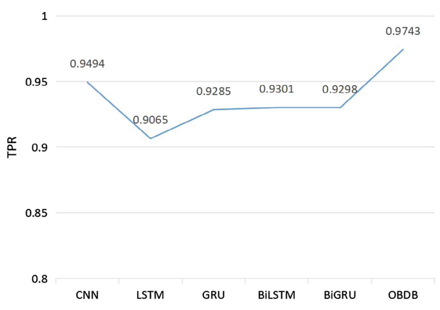

Fig. 4. TPRs in terms of different neural network models

5 Related Work

In recent years, the application of deep learning technology in the field of binary program analysis has achieved remarkable success [11,16]. Of the few works that implement obfuscation recognition, Zhao et al. [23] also adopts neural network based models. Different from their method that uses the set of basic blocks within a function to form the basic representation to be fed into subsequent neural networks, we propose to represent each function as a set of normalized reduced shortest paths extracted from its control flow graph, such that structural information can also be captured besides the syntactic information manifested by the assembly instructions appearing along the paths. It enables the subsequent neural network layers preferably capture subtle features that are indicative of the structural changes enforced by the obfuscation algorithms, so as to improve the whole detection performance.

In the literature, there are many code obfuscation detection approaches that focus on the scripting languages, such as JavaScript [10] and PowerShell [9], as well as on Android Apps [18], which also adopt machine learning and data mining based ways. JSObfusDetector detector [7] uses the SVM algorithm to get malicious JavaScript obfuscation scripts detected, after characterizing the number of string variables and the dynamic functions in JavaScript scripts. Wang et al. [18] extracts feature vectors from the Dalvik bytecode and uses it to identify the obfuscator provenance information within an App. On one hand, these methods works generally on the whole program level, and can not be easily adapted to the detection of obfuscation provenance on binary code, due to the missing of certain features that present in the source code but get lost during the compilation phase. Besides, rather than extracting and selecting features with manually defined templates, OBDB resorts to the powerful learning ability of neural network models and the largest number of training data to improve the accuracy and efficiency of the detection methods.

6 Conclusion

In this article, we propose OBDB to perform fine-grained binary code obfuscation recognition on isolated function, based on a novel reduced shortest path based function representation scheme and a deep neural network model that combines the typical CNN and RNN structures. Due to the lack of publicly accessible dataset, we constructed a relatively large dataset comprised of more than 60,000 obfuscated functions, by processing 11,000 C programs with 6 different obfuscation algorithms supported in OLLVM and Tigress. The experimental evaluation results conducted on the dataset indicate that, OBDB can effectively capture the significant features indicative of the code obfuscation algorithms. It can efficiently identify the existence of obfuscation with an accuracy of 98.6%, the accuracy of identifying the specific obfuscation algorithm also reaches a high score of 97.6%. Future work includes enriching the dataset with more samples and more kind obfuscation algorithms, as well as designing other neural representation learning models, such as the graph neural network (GNN), to comprehensively consider the CFG Node attributes and structural characteristics.

References

1. The tigress diversifying c virtualizer (2021). https://tigress.wtf/
2. Bichsel, B., Raychev, V., Tsankov, P., Vechev, M.T.: Statistical deobfuscation of android applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016)
3. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations (1997)
4. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: CCS 2011 (2011)
5. Ferguson, J.: Reverse Engineering Code with IDA Pro. O'Reilly Media, San Francisco (2008)
6. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.T.: On the naturalness of software. In: ICSE 2012 (2012)
7. Jodavi, M., Abadi, M., Parhizkar, E.: JSObfusdetector: a binary PSO-based one-class classifier ensemble to detect obfuscated javascript code. In: 2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP), pp. 322–327 (2015)
8. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM - software protection for the masses. In: 2015 IEEE/ACM 1st International Workshop on Software Protection, pp. 3–9 (2015)
9. Li, Z., Chen, Q., Xiong, C., Chen, Y., Zhu, T., Yang, H.: 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 (2019)
10. Likarish, P., Jung, E., Jo, I.: Obfuscated malicious javascript detection using classification techniques. In: 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), pp. 47–54 (2009)
11. Massarelli, L., Luna, G.A.D., Petroni, F., Querzoni, L., Baldoni, R.: Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. Internet Society (2019)
12. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: ICLR (2013)
13. Ming, J., Xu, D., Wang, L., 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)
14. Perozzi, B., Al-Rfou, R., Skiena, S.: DeepWalk: online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2014)
15. Salwan, J., Bardin, S., Potet, M.: Symbolic deobfuscation: from virtualized code back to the original. In: DIMVA (2018)
16. Tian, Z., Huang, Y., Xie, B., Chen, Y., Chen, L., Wu, D.: Fine-grained compiler identification with sequence-oriented neural modeling. IEEE Access **9**, 49160–49175 (2021). https://doi.org/10.1109/ACCESS.2021.3069227
17. Udupa, S.K., Debray, S., Madou, M.: Deobfuscation: reverse engineering obfuscated code. In: 12th Working Conference on Reverse Engineering (WCRE'2005), pp. 10–54 (2005)
18. Wang, Y., Rountev, A.: Who changed you? Obfuscator identification for android. In: The 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 154–164 (2017)

19. Wang, Y., Wu, Z., Wei, Q., Wang, Q.: NeuFuzz: efficient fuzzing with deep neural network. IEEE Access **7**, 36340–36352 (2019)
20. Xu, D., Ming, J., Fu, Y., 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)
21. Xue, H., Venkataramani, G., 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)
22. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: 2015 IEEE Symposium on Security and Privacy, pp. 674–691 (2015)
23. Zhao, Y., Tang, Z., Ye, G., Peng, D., Fang, D., Chen, X., Wang, Z.: Semantics-aware obfuscation scheme prediction for binary. Comput. Secur. **99**, 102072 (2020)