*Article*

# DFSGraph: Data Flow Semantic Model for Intermediate Representation Programs Based on Graph Network

**Ke Tang [1,†], Zheng Shan [2,†*], Chunyan Zhang [3,†], Lianqiu Xu [4,†], Meng Qiao [5,†], Fudong Liu [6,†],**

‡ State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000
[1] tuck3r@foxmail.com; [2] zzzhengming@163.com; [3] iecyzhang@163.com; [4] lianqiuxu@163.com; [5] kopo.code@gmail.com; [6] lwfydy@126.com
* Correspondence: zzzhengming@163.com

**Abstract:** Obfuscation techniques have become complex and diverse, and while they play a crucial role in software copyright protection, they also help produce numerous malware variants that evade antivirus software. Automatic detection of malware is inseparable from binary code similarity detection technology. However, the existing code detection methods are difficult to cope with the increasingly complex obfuscation techniques. Therefore, in this paper, we propose a method combining data flow relationships and neural network to analyze obfuscated code for the first time. In our approach, we first construct the data transformation graph based on LLVM IR. Then, we design a novel intermediate language representation model based on graph neural network, named *DFSGraph*, to learn the data flow semantic from DTG. Through extensive experiments on obfuscated dataset, it is proved that our method can extract the semantic information of obfuscated code well. And it can achieve surprising results in binary code similarity detection task and obfuscation technique classification task. Our method provides an idea for further research on deobfuscation techniques.

**Keywords:** Obfuscation; Deobfuscation; LLVM IR; Graph Network

## 1. Introduction

Existing obfuscation tools mainly include Obfuscator-LLVM[1], VMProtect[2], Themida[3], Tigress[4], etc. The most commonly used obfuscation techniques in malware [1] cover encryption, dead code insertion, register reallocation, instruction reordering, instruction substitution, opaque predicate insertion, etc. As an exploratory work, we initially select two commonly used obfuscators Obfuscator-LLVM and Tigress for experimental research.

In Obfuscator-LLVM, there are mainly three obfuscation options: *sub, fla, bcf*. In Tigress C obfuscator, there are too many obfuscation options, we select five representative options: *addOpaque, EncodeLiterals, Virtualize, Flatten, EncodeArithmetic*. To be representative, we choose 4 and 16 as the number of opaque predicates for option *addOpaque*, and treat them as two obfuscation options *addOpaque4, addOpaque16* respectively. In one obfuscator, multiple obfuscation options can be superimposed, and it will result in a more complex and hard-to-understand binary program. Among them, there are 4 mixed obfuscation schemes in O-LLVM and 17 in Tigress. Together with 9 single obfuscation schemes, there are 30 different obfuscation techniques in total.

In recent years, code obfuscation technology has been developed rapidly to protect software copyright from plagiarism. It can protect critical code from being cracked by attackers in commercial software. But everything has two sides. Obfuscation techniques have penetrated into the realm of malware. Attackers have used them to obfuscate malicious

[1] https://github.com/obfuscator-llvm/obfuscator
[2] https://vmpsoft.com/
[3] https://www.oreans.com/themida.php
[4] https://tigress.wtf

 doi:10.20944/preprints202209.0046.v1



code for evading detection by antivirus software, leading to a proliferation of malware variants. To solve such problems, binary code similarity detection technology is essential. The existing binary code deobfuscation methods are mainly divided into two categories: traditional binary code analysis methods and neural network-based binary code analysis methods.

In traditional binary code analysis methods, some researchers use search-based methods to study deobfuscation techniques [2–4]. Specifically, Blazytko et al. [3] proposes a generic approach for automated code deobfuscation using program synthesis guided by Monte Carlo Tree Search. On this basis, Zhao et al. [4] uses a heuristic nested Monte Carlo Search algorithm to locate obfuscated code fragments for improving the search efficiency. Some researchers have also conducted research on specific obfuscators. For example, David et al. [5] mainly studied the virtualization-based commercial obfuscators VMProtect and Themida, There are also some scholars who only study the detection of obfuscated code and the classification of obfuscation techniques. For obfuscated code detection, Ming et al. [6] proposes a logic-oriented opaque predicate detection tool. It can construct general logical formulas representing the intrinsic characteristics of opaque predicates through the symbolic execution of trace, and use constraint solvers to solve these formulas to determine whether they contain opaque predicates. For obfuscated code classification, Kim et al. [7] directly uses the histogram of the opcode to classify the obfuscation technology according to the frequency of different opcodes.

Among the neural network based methods, many researchers have proposed different binary code analysis methods [8,9]. But most of them don't think about obfuscated binaries, and only a few use the proposed method to perform simple tests on obfuscated code. For instance, In *Asm2Vec* proposed by Ding et al. [10], the CFG of the assemble program is converted into instruction sequences by random walk algorithm, and the adapted PV-DM model is used for representation learning. The author uses this method to test the binary code compiled with the obfuscation options of O-LLVM, and the experimental results show that *Asm2Vec* has a certain anti-obfuscation ability. However, this method only tests OLLVM and does not involve other obfuscators. In addition, it does not design a special structure or algorithm to resist existing obfuscation techniques.

In general, compared with the traditional binary code similarity analysis methods, although neural networks have a certain degree of inexplicability, the binary code similarity analysis technology based on neural network has considerable advantages, such as automatic analysis of massive codes. Moreover, due to the interference of obfuscation techniques, the existing binary code similarity analysis technology is difficult to deal with the obfuscated code effectively. Therefore, in this paper we propose a novel model approach to learn semantic representations of obfuscated codes automatically. The main contributions of this paper are as follows:

- We propose the Data Transformation Graph (DTG) for the first time, and it can express the data flow transformation relationship of the function completely and clearly.
- We propose a new graph network model that integrates edge feature in directed graph and redefines the message propagation algorithm, which can learn the semantic information from DTGs better.
- Experiments show that our proposed method can learn the semantic information of obfuscated code well and exhibits satisfactory performance in downstream tasks.

The remaining part of this paper proceeds as follows: Section 2 mainly introduces the related works based on LLVM IR and graph neural network. Section 3 describes our proposed method in detail. Section 4 shows the results of our experiments. Section 5 discusses the advantages and disadvantages of our method and look forward to the improvement direction. And we make a conclusion in section 6.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1



## 2. Related works

### 2.1. Intermediate Representation

Intermediate language is independent of the programming language and hardware platform, and is a compiler-based intermediate representation, LLVM IR is one of them. LLVM IR[5] is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations and flexibility.

At present, Some researchers analyze the semantic information of the code from the IR perspective. Among them, Ben-Nun et al. [11] provides a new concept XFG to describe the context flow information of IR, which leverages the underlying control flow and data flow information. They can match or even surpass advanced methods using only simple LSTM and pre-trained embeddings. Venkatakeerthy et al. [12] proposes *IR2Vec*, a Concise and Scalable encoding infrastructure to represent programs as a distributed embedding in continuous space. This method takes two embeddings, Symbolic and Flow-aware, constructs LLVM Entities and maps them to real-valued distributed embeddings.

There are also some researchers who study deobfuscation techniques based on LLVM IR. For instance, Garba et al. [13] proposes *SATURN*, which is an LLVM-based software deobfuscation framework. This method lifts binary code to LLVM IR, then uses optimization pass and super optimization to simplify intermediate representation. Experiments show that this method is very effectively for obfuscation techniques such as constant unfolding, certain arithmetic-based opaque expressions, dead code insertions, bogus control flow and integer encoding.

### 2.2. Graph Neural Network

With the rapid development of deep learning, graph representation learning has also made significant progress in recent years. The graph neural network models such as GCN [14] and GAT [15] have shown satisfactory results in the fields of node classification, graph classification and link prediction. As a variant of GCN, GraphSAGE [16] optimizes node-centered neighbor sampling instead of full-graph sampling, enabling it to perform inductive learning and improve the the efficiency of graph convolution greatly. With Transformer [17] showing great potential in the field of neural networks, the attention mechanism has gradually been applied to the field of graph neural networks [18–20].

In many real scenarios, the edges may contain very important information. But these models fail to take full advantage of the edge information in the graph. Therefore, many researchers attempt to incorporate edge information into graph neural networks. For example, the Relational-GCN proposed by [21] simply aggregates node information according to edge types, but this method cannot handle the multi-dimensional features on the edge well. In PNAConv [22] and Crystal Graph Conv [23], the edge information in the graph is also regarded as multi-dimensional features, and operations such as aggregation and update are performed together with node features. In EGNN [24], each dimension of the edge feature is processed separately, and the final output vector is concatenated to obtain a new edge feature. It will produce long redundant vectors when the dimension become larger, making it less scalabe. In addition, there are also methods to alternately learn the embeddings of nodes and edges in the graph [25**?** ,26]. These methods all achieve very good results for specific tasks, but are not suitable for our task.

## 3. Our Approach

### 3.1. Our Insights

We have conducted a detailed study of various existing obfuscation techniques and found that most of the obfuscation techniques can be divided into two categories: structural obfuscation and data obfuscation. In structural obfuscation, there are mainly two ways. One is to build opaque predicates to fake the control flow, and the other is to use branch instructions such as switch to flatten the control flow. The former will greatly change

[5] https://llvm.org/docs/LangRef.html

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1



the control flow structure of the program, while the latter will significantly change the instruction relationship of the program, which makes it difficult to detect the existing binary code similarity analysis technology.

In structural obfuscation, no matter how the control flow changes, the semantic information of the program remains unchanges, so the data flow relationship also remains the same. In data obfuscation, encryption algorithms are mainly used to encrypt the original code, or more complex instruction relations are used to replace simple expressions. Since the functionality of the program is completely the same, its overall data flow conversion relations should be equivalent. Therefore, we focus on the direction of data flow transmission to analyze the obfuscation code. But considering that the assembly language only contains only limited registers and varies in different hardware architectures, it is difficult to extract the data flow relationship. To simplify our work and make it more versatile, we perform data flow analysis at the intermediate representation level.

### *3.2. Overview*

In this paper, we use C source code from the Google Code Jam(2008-2020) as our original dataset. We compile the original dataset with all obfuscation options for both obfuscators O-LLVM and Tigress, respectively. Through this operation, we can get the corresponding obfuscated intermediate representation program. After preprocessing, we are able to extract data flow relationships from the intermediate representation program and build data transformation graph (DTG) as our training dataset. Finally, the DTG is embedded using the graph network model (DFSGraph) proposed in this paper, and the semantic information of the data flow is automatically learned through model training and used for downstream tasks. The overall framework is shown in the figure 1.

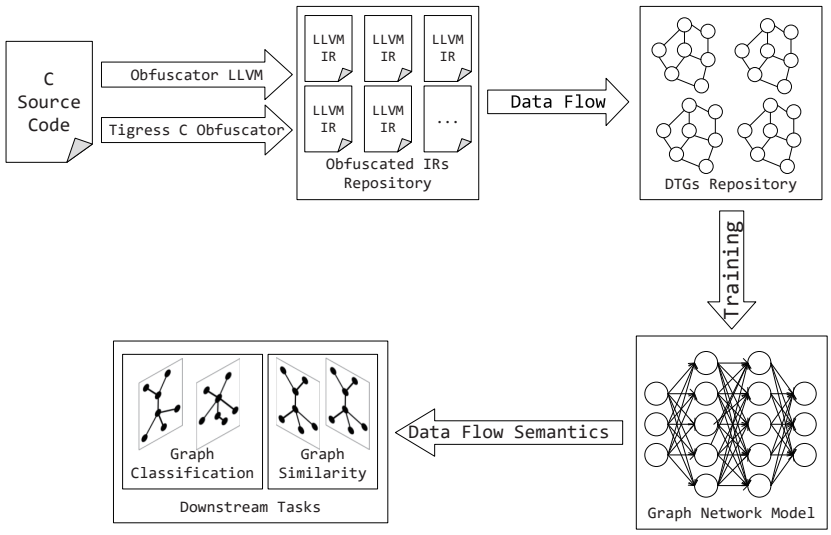



**Figure 1.** The overall framework of our approach.

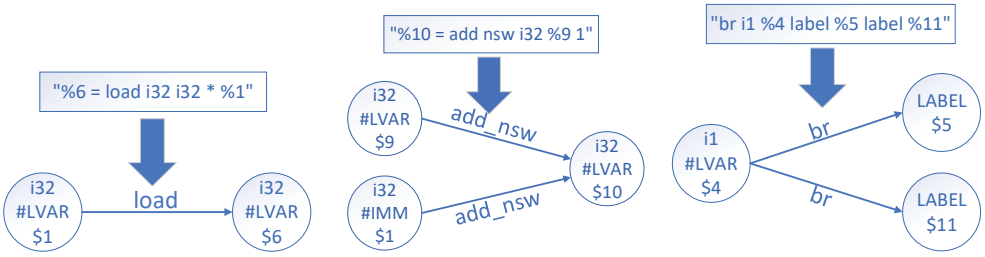
### *3.3. Data Transformation Graph*

According to the format specification of the LLVM IR instruction, we perform preliminary preprocessing. First, the useless characters in the IR instruction are removed. Then, we analyze the format of the instruction according to the opcode of the instruction, and extract each operand and its corresponding data type.

According to the functionality of operands, we divide them into local variables, global variables, immediate value, basic block label, and use *LVAR*, *GVAR*, *IMM*, *LABEL* to represent it respectively. At the same time, we divide the oprands into *i8*, *i16*, *i32*, *i64*, *i128*, *float* depending on the varibale types. Besides, we divide the instructions of LLVM IR into various types according to the number of operands, which are zero-element operators, unary operators, binary operators and multivariate operators.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1






(a) The unary operator. (b) The binary operator. (c) The multivariate operator.

**Figure 2.** Data flow conversion relationship for different types of instructions.

For zero-element operators, such as *alloca* and *ret*, since there is no data flow information transfer, they can be ignored directly. For the unary operator, such as *load*, we take "*%6 = load i32 i32 * %1*" as an example, and its data flow transformation is shown in Figure 2(a). For binary operators, such as *add*, *sub*, *mul*, we take "*%10 = add nsw i32 %9 1*" as an example, and its data flow transformation is shown in Figure 2(b). For multivariate operators, such as *br*, we take "*br i1 %4 label %5 label %11*" as an example, and its data flow transformation is shown in Figure 2(c). To simplify the parsing complexity, for nested instructions, such as "*call @func ( i8 * getelement ...)*", we use *%opcode* to denote it: "*call @func %opcode*". Although this will lose part of the data flow relationship, it is a trade-off between accuracy and complexity. Figure 3 is an example of a data transformation graph.

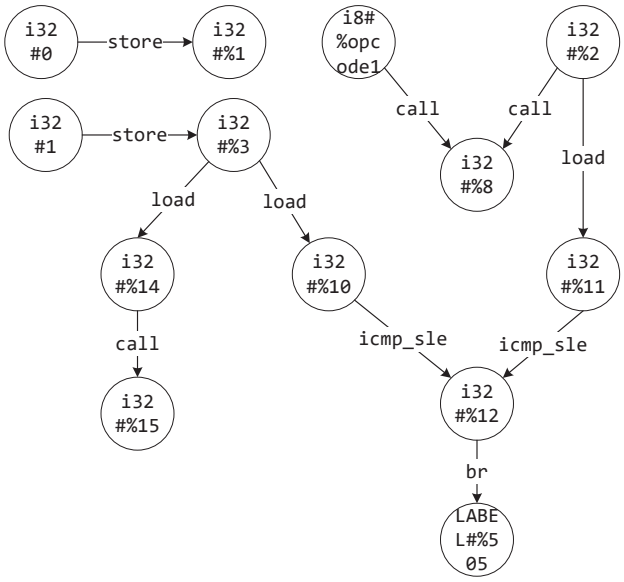



**Figure 3.** An example of DTG.

### *3.4. Graph Network*

Inspired by the Graph Network [27], we redesigned the message passing function and aggregation function to make it more suitable for DTGs, which can efficiently extract data flow semantic information of intermediate representation programs while resisting existing obfuscation techniques.

In our proposed graph network model, the main update steps are shown in Figure 4. In the whole update process, it is mainly divided into three parts.

First, we update the edge features based on the features of the nodes at both ends of the edge and the current state information of the entire graph, which can be formally described as follows:

$$e_{ij}^{l+1} = \Phi^e(e_{ij}^l, h_i^l, h_j^l, u^l)$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1



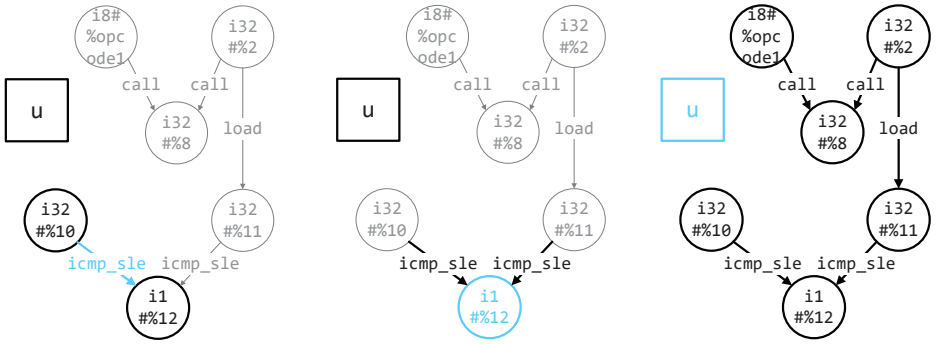



**Figure 4.** The update process of the DFSGraph.

where $e_{ij}^l$ represents the feature of directed edge $e_{ij}$ from node $i$ to node $j$ in the $l$th layer. $h_i^l$ represents the current feature of node $i$. $u_i^l$ represents the global state information of the graph. $\Phi^e$ is the update funciton of the edge. In DFSGraph, $\Phi^e$ is defined as:

$$\Phi^e = e_{ij}^l * (h_i \| h_j) * W_h + u^l * W_u$$

in which, $\|$ represents the concatenation operation of the vector, $W_h \in \mathbb{R}^{2D*M}$ and $W_u \in \mathbb{R}^{(D+M)*M}$ are the parameter matrixes.

Next, the state update process of the node is performed. We update the state of the current node by aggregating the feature vectors of all incoming edges of the current node, combined with the current global state information of the graph. Its formal description is as follows:

$$\overline{e}_i^{l+1} = \rho^{e\to h}([e_{ij}^{l+1}, \forall v_j \in \mathcal{N}(V_i)])$$

$$h_i^{l+1} = \Phi^h(\overline{e}_i^{l+1}, h_i^l, u_l)$$

where, $\mathcal{N}(V_i)$ represents all neighbors of node $i$. $\rho^{e\to h}$ is the aggregation function of the edge. In this paper, we find that max pooling works best through repeated experiemntal verification, which is defined as follows:

$$\rho^{e\to h} = maxpooling(e_{ij}^{l+1})$$

$\Phi^h$ is the update function of the node, which is defined as follows:

$$\Phi^h = Dropout(\overline{e}_i^{l+1} * W_e * h_i^l) + u^l * W_u$$

in which, $W_e \in \mathbb{R}^{M*D}$ is the parameter matrix that maps edge features to node features.

Last, we update the global state information of the graph. The update of the global state information of the graph needs to depend on the features of all edges and nodes. All edges and nodes are aggregated separately, then the global feature are updated using the state update function of the graph. Its formal description is as follows:

$$\overline{e}^{l+1} = \rho^{e\to u}([e_{ij}^{l+1}, \forall e_{ij} \in E])$$

$$\overline{h}^{l+1} = \rho^{h\to u}([h_i^{l+1}, \forall v_i \in V])$$

$$u^{l+1} = \Phi^u(\overline{e}^{l+1}, \overline{h}^{l+1}, u^l)$$

In this paper, we define the edge aggregation function $\rho^{e\to u}$, node aggregation function $\rho^{h\to u}$, and global state update $\Phi^u$ as follows:

$$\rho^{e\to u} = meanpooling(e_{ij}^{l+1})$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1



$$\rho^{h \to u} = meanpooling(h_{ij}^{l+1})$$

$$\Phi^{u} = \alpha * Dropout(\bar{e}^{l+1} \| \bar{h}^{l+1}) + (1 - \alpha) * u^{l}$$

where, $\alpha$ is the forgetting coefficient. We set the initial value of the graph state $u$ to zero, that is: $u^0 = \vec{0}$. Then, we can embed DTGs using the current graph network model.

### 3.5. Model Training

During model training, we employ triplet loss for training. The input form of each triple is $< \mathbf{a}, \mathbf{p}, \mathbf{n} >$, where **a** is the anchor sample refering to the unonbfuscated function, **p** is the positive sample refering to the homologous function after specified obfuscation technique, and **n** is the negative sample, which is the non-homologous function that has undergone the same obfuscation technique. Assuming that the obtained semantic vector of **a** through DFSGraph model is $x = (x_1, x_2, x_3, ..., x_n)$, and the semantic vector corresponding to **p** is $y = (y_1, y_2, y_3, ..., y_n)$. The the Euclidean distance between sample **a** and **p** is as follows:

$$d(a, p) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

For the triplet $< \mathbf{a}, \mathbf{p}, \mathbf{n} >$, the triplet loss can be formally described as:

$$L = \max\{d(a, p) - d(a, n) + margin, 0\}$$

In our implementation, we split the dataset into training set and testing set in a rate of 4:1. Since we are using a graph network, our batch size can only be set to 1. In DFSGraph, we set the dimension of the edges and nodes to be 128, and the output graph global state vector is also 128. The graph network is set to 3 layers. During the training phase, *AdamW* is used as optimizer. The margin in triplet loss is set to 50, and the learning rate is set to 0.00005.

## 4. Experiments

In this section, we apply two tasks to evaluate the effectiveness of our proposed method.

### 4.1. Dataset

At present, we use the C source dataset from Google Code Jam, with a total of 25,000 functions. We use a single obfuscation technology or a composite obfuscation technology to compile these source codes separately. All obfuscation technologies are shown in table 1.

**Table 1.** All obfuscation options in O-LLVM and Tigress. (Note: *A* is short for *addOpaque16*, *V* is short for *Virtualize*, *EA* is short for *EncodeArithmetic*, *EL* is short for *EncodeLiterals* and *F* is short for *Flatten*. *A+V* represents a composite option of *addOpaque16* and *Virtualize*, and others are the same.)

| Obfuscator | Options | Composite Options |
|---|---|---|
| O-LLVM | sub, fla, bcf | sub+fla, sub+bcf, fla+bcf, sub+fla+bcf |
| Tigress | addOpaque4, *A*, *V*, *EA*, *EL*, *F* | *A+V*, *A+EA*, *A+EL*, *A+F*, *A+V*, *EA+A*, *EA+V*, *EA+F*, *EL+A*, *EL+EA*, *EL+F*, *EL+V*, *F+A*, *F+EA*, *F+EL*, *F+V*, *V+A*, *V+EA*, *V+EL*, *V+F* |

We should note that in the compound obfuscation options, **A+V** means that the addOpaque16 option is used first, and then the Virtualize option is used at compile time, which is different from **V+A**.

### 4.2. Similarity Analysis of Obfuscated Code

In this experiment, we use P@N(precision at N) to evaluate the accuracy of the model. The P@N is often used in information retrieval systems, which reflects the probability of

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1



correct sample ranking at top N. So it can reflect the ability of our model to resist obfuscation techniques in the task of detecting binary code similarity. For each obfuscation technique, We first randomly select 100 functions as the search set $\Gamma$. Then, we compile each function in $\Gamma$ using a specific obfuscation technique. At the same time, we randomly select one function from $\Gamma$, compile it normally, and treat it as the function $\gamma$ to be retrieved. Last, we calculate the Euclidean distance between function $\gamma$ and each function in $\Gamma$ one by one, and sort them. Then we are able to get the rank of the obfuscation function corresponding to the function $\gamma$. Each experiment is repeated 100 times and we are able to get the probability of P@N.

**Table 2.** Experimental results of obfuscated code similarity detection.

| Obfuscator | Options | p@1 | p@2 | p@3 | p@5 | p@10 |
|---|---|---|---|---|---|---|
| O-LLVM | sub | 0.909 | 0.970 | 0.993 | 0.997 | 1.000 |
| | fla | 0.910 | 0.975 | 0.992 | 0.997 | 1.000 |
| | bcf | 0.903 | 0.974 | 0.986 | 0.995 | 0.999 |
| | sub+bcf | 0.899 | 0.972 | 0.991 | 0.995 | 0.999 |
| | sub+fla | 0.913 | 0.978 | 0.989 | 0.994 | 0.999 |
| | bcf+fla | 0.872 | 0.962 | 0.985 | 0.993 | 0.997 |
| | sub+bcf+fla | 0.870 | 0.948 | 0.977 | 0.992 | 0.998 |
| Tigress | addOpaque4 | 0.818 | 0.921 | 0.964 | 0.989 | 0.998 |
| | *A* | 0.809 | 0.928 | 0.963 | 0.992 | 1.000 |
| | *EA* | 0.797 | 0.935 | 0.966 | 0.989 | 0.997 |
| | *EL* | 0.892 | 0.972 | 0.987 | 0.995 | 0.998 |
| | *F* | 0.893 | 0.965 | 0.988 | 0.993 | 0.999 |
| | *V* | 0.761 | 0.901 | 0.945 | 0.983 | 0.996 |
| | *A+EL* | 0.735 | 0.878 | 0.927 | 0.966 | 0.991 |
| | *A+V* | 0.528 | 0.703 | 0.793 | 0.892 | 0.974 |
| | *EA+A* | 0.722 | 0.876 | 0.932 | 0.974 | 0.997 |
| | *EA+F* | 0.824 | 0.944 | 0.980 | 0.994 | 1.000 |
| | *EA+V* | 0.641 | 0.809 | 0.892 | 0.957 | 0.993 |
| | *EL+EA* | 0.821 | 0.937 | 0.978 | 0.993 | 0.999 |
| | *EL+F* | 0.897 | 0.978 | 0.995 | 0.998 | 1.000 |
| | *EL+V* | 0.718 | 0.862 | 0.935 | 0.978 | 0.997 |
| | *F+A* | 0.817 | 0.917 | 0.963 | 0.987 | 0.997 |
| | *F+EA* | 0.813 | 0.924 | 0.969 | 0.994 | 0.999 |
| | *F+EL* | 0.903 | 0.980 | 0.993 | 0.999 | 1.000 |
| | *F+V* | 0.749 | 0.885 | 0.938 | 0.982 | 0.998 |
| | *V+A* | 0.630 | 0.795 | 0.885 | 0.953 | 0.991 |
| | *V+EL* | 0.760 | 0.894 | 0.951 | 0.986 | 0.996 |
| | *V+F* | 0.725 | 0.885 | 0.947 | 0.989 | 0.987 |
| | *V+V* | 0.654 | 0.800 | 0.877 | 0.948 | 0.987 |
| | *V+EA* | 0.591 | 0.771 | 0.862 | 0.952 | 0.995 |

The final results are shown in table 2. From the experimental results, we can find that, for all obfuscation options, P@10 can achieve an accuracy of more than 98% in the test results, and for most obfuscation options, P@1 can also reach 80%. For the O-LLVM obfuscator, our method outperforms *Asm2Vec* [10]. For the Tigress obfuscator, the accuracy and efficiency of our method far exceed the existing state-of-the-art deobfuscation tool *Xyntia* [2]. In addition, we can clearly see that our model is able to achieve quite high accuracy for structural obfuscation, such as *bcf*, *fla* in O-LLVM and *Flatten* in Tigress. This proves our dataflow-based method is robust to structural obfuscation. For slight data obfuscation techniques such as *sub* in O-LLVM and *EncodeLiterals* in Tigress, our model is still able to maintain high accuracy. However, for obfuscation techniques with heavy data flow changes, such as *Virtualize*, *addOpaque16*, *EncodeArithmetic* in Tigress, the accuracy is relatively low. This shows that our method is partially resistant to data obfuscation and still need to be further improved.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1



### 4.3. Identification of Obfuscated Techniques

In this section, we add a simple linear layer to the original model for making it suitable for the obfuscation technique classification task. For O-LLVM and Tigress, multiple obfuscation options within the same obfuscator can be arbitrarily matched. In addition, it is only necessary to judge whether the specific option is used, no matter the combination order of the options. Therefore, we adopt the idea of multi-label classification, and treat each label as a binary classification task. So we replace triplet loss with binary cross-entropy loss, whose formula is as follows:

$$loss = \frac{1}{N}\sum_{n=1}^{N} l_n$$

where, the loss corresponding to the nth label is $l_n$, which can be described as:

$$l_n = -w[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]$$

Next, we can retrain the parameters of our model. We evaluate our model using common performance metrics: accuracy, precision, recall and F1-score.

Since we are a multiple binary classification problem, we average the individual metrics across all classes for simplicity. Figure 5 shows the changes of each indicator during the training process of the classification model.

From figure 5, we can see that the accuracy and precision of the trained classification model can be stabilized at about %98.6, indicating that our model can indeed predict the obfuscation technology with high probability. Meanwhile, we see that the recall and f1 values, although not very stable, are able to stay above 90% after 10 epoches. These four indicators show that our proposed model can achieve a good classification effect after only a small amount of training (about 10 rounds), which reflects the effectiveness of our proposed method.

## 5. Discussion

Although our model has achieved satisfactory results in experiments, it is still far from practical application. In our experiments, our intermediate representation program is compiled directly from the source code. However, it is almost impossible for us to obtain the source code corresponding to the binaries in the wild code. We have to disassemble the binaries into intermediate representation programs through tools like *RetDec*[6]. It will cause some information loss inevitably, making it difficult to completely recover the complete data flow information of the binaries. In the future, we can deeply study how to use *RetDec* to disassemble binaries into intermediate representation programs, preserving data flow information as much as possible. We can also look for other alternatives to make up for the missing information and minimize the impact of the information loss.

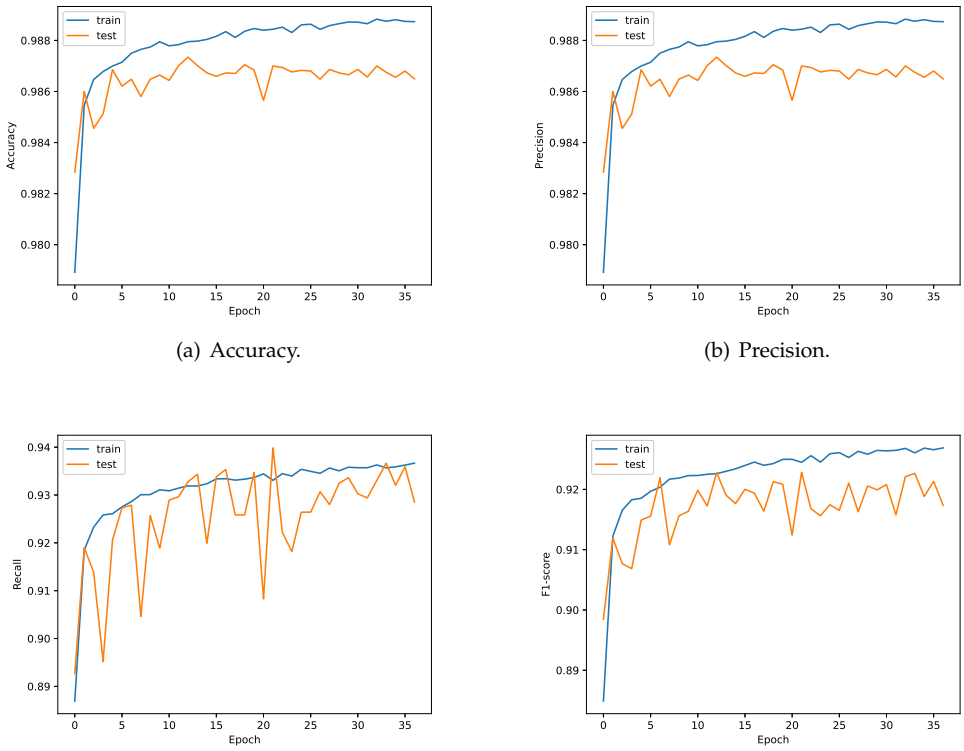
## 6. Conclusion

In this paper, we propose the data transformation graph based on LLVM IR for the first time. Then, we design a new graph network model *DFSGraph* to learn the data flow semantic from DTG. We constructed the obfuscated code dataset with the obfuscation options in the O-LLVM and Tigress, and use it training *DFSGraph* model. Experiments prove that our model is quite resistant to most obfuscation techniques. Especially for structural obfuscation, our model has a higher accuracy in the binary code similarity comparison task. For the obfuscation technologies with obvious data flow changes such as virtualization and encryption, the accuracy is relatively low, which requires further research in the future.

[6] https://github.com/avast/retdec

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 September 2022 doi:10.20944/preprints202209.0046.v1






(a) Accuracy. (b) Precision.

(c) Recall. (d) F1-score.

**Figure 5.** Metric changes during the training phase of the classification model.

## References


1. Landage, J.; Wankhade, M. Malware and Malware Detection Techniques: A Survey. *International Journal of Engineering Research & Technology* **2013**, *2*, 61–68.
2. Menguy, G.; Bardin, S.; Bonichon, R.; Lima, C.D.S. *Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate*; Vol. 1, Association for Computing Machinery, 2021; pp. 2513–2525. https://doi.org/10.1145/3460120.3485250.
3. Blazytko, T.; Contag, M.; Aschermann, C.; Holz, T. Syntia: Synthesizing the semantics of obfuscated code. *Proceedings of the 26th USENIX Security Symposium* **2017**, pp. 643–659.
4. Zhao, Y.; Tang, Z.; Ye, G.; Gong, X.; Fang, D. Input-Output Example-Guided Data Deobfuscation on Binary. *Security and Communication Networks* **2021**, *2021*. https://doi.org/10.1155/2021/4646048.
5. David, R.; Coniglio, L.; Ceccato, M. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation **2020**. https://doi.org/10.14722/bar.2020.23009.
6. Ming, J.; Xu, D.; Wang, L.; Wu, D. LOOP: Logic-oriented opaque predicate detection in obfuscated binary code. *Proceedings of the ACM Conference on Computer and Communications Security* **2015**, *2015-Octob*, 757–768. https://doi.org/10.1145/2810103.2813617.
7. Kim, J.; Kang, S.; Cho, E.S.; Paik, J.Y. *LOM: Lightweight Classifier for Obfuscation Methods*; Vol. 13009 LNCS, Springer International Publishing, 2021; pp. 3–15. https://doi.org/10.1007/978-3-030-89432-0_1.
8. Peng, D.; Zheng, S.; Li, Y.; Ke, G.; He, D.; Liu, T.Y. How could Neural Networks understand Programs? **2021**.
9. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order matters: Semantic-aware neural networks for binary code similarity detection. *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence* **2020**, pp. 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466.
10. Ding, S.H.; Fung, B.C.; Charland, P. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *Proceedings - IEEE Symposium on Security and Privacy* **2019**, *2019-May*, 472–489. https://doi.org/10.1109/SP.2019.00003.
11. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. 2018, Vol. 2018-Decem, pp. 3585–3597.
12. Venkatakeerthy, S.; Aggarwal, R.; Jain, S.; Desarkar, M.S.; Upadrasta, R.; Srikant, Y.N. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Transactions on Architecture and Code Optimization* **2020**, *17*. https://doi.org/10.1145/3418463.
13. Garba, P.; Favaro, M. SATURN - Software Deobfuscation Framework Based on LLVM. *SPRO 2019 - Proceedings of the 3rd ACM Workshop on Software Protection* **2019**, pp. 27–38. https://doi.org/10.1145/3338503.3357721.

 doi:10.20944/preprints202209.0046.v1




14. Kipf, T.N.; Welling, M. Semi-Supervised Classification with Graph Convolutional Networks **2016**.
15. Veličković, P.; Casanova, A.; Liò, P.; Cucurull, G.; Romero, A.; Bengio, Y. Graph attention networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* **2018**, pp. 1–12.
16. Hamilton, W.L.; Ying, R.; Leskovec, J. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems* **2017**, *2017-Decem*, 1025–1035.
17. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Łukasz Kaiser.; Polosukhin, I. Attention is all you need. *Advances in Neural Information Processing Systems* **2017**, *2017-Decem*, 5999–6009. https://doi.org/10.48550/arXiv.1706.03762.
18. Yun, S.; Jeong, M.; Kim, R.; Kang, J.; Kim, H.J. Graph Transformer Networks **2019**. [1911.06455].
19. Rong, Y.; Bian, Y.; Xu, T.; Xie, W.; Wei, Y.; Huang, W.; Huang, J. Self-Supervised Graph Transformer on Large-Scale Molecular Data **2020**. [2007.02835].
20. Ying, C.; Cai, T.; Luo, S.; Zheng, S.; Ke, G.; He, D.; Shen, Y.; Liu, T.Y. Do Transformers Really Perform Bad for Graph Representation? **2021**. [2106.05234].
21. Schlichtkrull, M.; Kipf, T.N.; Bloem, P.; van den Berg, R.; Titov, I.; Welling, M. Modeling Relational Data with Graph Convolutional Networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **2017**, *10843 LNCS*, 593–607.
22. Corso, G.; Cavalleri, L.; Beaini, D.; Liò, P.; Veličković, P. Principal Neighbourhood Aggregation for Graph Nets **2020**. [2004.05718].
23. Xie, T.; Grossman, J.C. Crystal Graph Convolutional Neural Networks for an Accurate and Interpretable Prediction of Material Properties. *Physical Review Letters* **2018**, *120*, 145301, [1710.10324]. https://doi.org/10.1103/PhysRevLett.120.145301.
24. Gong, L.; Cheng, Q. Exploiting edge features for graph neural networks. IEEE, 2019, Vol. 2019-June, pp. 9203–9211. https://doi.org/10.1109/CVPR.2019.00943.
25. Jiang, X.; Ji, P.; Li, S. CensNet: Convolution with Edge-Node Switching in Graph Neural Networks. In Proceedings of the Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence; International Joint Conferences on Artificial Intelligence Organization: California, 2019; pp. 2656–2662. https://doi.org/10.24963/ijcai.2019/369.
26. Chen, J.; Chen, H. Edge-Featured Graph Attention Network **2021**.
27. Battaglia, P.W.; Hamrick, J.B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. Relational inductive biases, deep learning, and graph networks **2018**.