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: With the improvement of software copyright protection awareness, code obfuscation technology plays a crucial role in protecting key code segments. As the obfuscation technology becomes more and more complex and diverse, it has spawned a large number of malware variants, which make it easy to evade the detection of anti-virus software. Malicious code detection mainly depends on binary code similarity analysis. However, the existing software analysis technologies are difficult to deal with the growing complex obfuscation technologies. To solve this problem, this paper proposes a new obfuscation-resilient program analysis method, which is based on the data flow transformation relationship of the intermediate representation and the graph network model. 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 networks, named *DFSGraph*, to learn the data flow semantics from DTG. *DFSGraph* can detect the similarity of obfuscated code by extracting the semantic information of program data flow without deobfuscation. Extensive experiments prove that our approach is more accurate than existing deobfuscation tools when searching for similar functions from obfuscated code.

Keywords: Obfuscation; Deobfuscation; LLVM IR; Graph Network

1. Introduction

Code obfuscation is the act of converting computer programs into functionally equivalent, but difficult to read and understand. 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 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, most researchers use symbolic execution to combat code deobfuscation. For instance, Tofighi-Shirazi et al. [1] proposed DoSE, which can improve the deobfuscation technique based on dynamic symbolic execution by statically eliminating obfuscation transformations, and remove two-way opaque constructs by semantic equivalence. While Xu et al. [2] adopted the multi-granularity symbolic execution method to simplify the trace snippets for partially virtualized binary code, and achieved encouraging experimental results at that time. In the meantime, few studies use search-based algorithms [3–5]. Specifically, Blazytko et al. [4] proposed a generic approach for automated code deobfuscation using program synthesis guided by Monte Carlo Tree Search. On this basis, Zhao et al. [5] used a heuristic nested Monte Carlo Search algorithm to locate obfuscated code fragments for improving the search efficiency.

There is also some research focusing on specific obfuscators. For example, David et al. [6] mainly studied virtualization-based commercial obfuscators, such as VMProtect and Themida, and achieved good results. Eyrolles et al. [7] proposed a simplified algorithm based on pattern matching for MBA obfuscation technology, and proved that at least a subset of MBA obfuscation lacks resistance to pattern matching analysis. Whereas some scholars only study the detection of obfuscated code and the classification of obfuscation techniques. For obfuscated code detection, Ming et al. [8] proposed 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. [9] directly used the histogram of the opcode to classify the obfuscation technology according to the frequency of different opcodes.

For binary code detection methods based on neural networks, many studies have been proposed one after another [10,11]. 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, Ding et al. [12] proposed *Asm2Vec*, the CFG of the assemble program is converted into instruction sequences by a random walk algorithm, and the adapted PV-DM model is used for representation learning. The authors used this method to test the binary code compiled with the obfuscation options in O-LLVM, and the experimental results showed that *Asm2Vec* has a certain anti-obfuscation ability. However, this method only tests O-LLVM and does not involve other obfuscators. And it does not design a special structure or algorithm to resist existing obfuscation techniques. In addition, FID [13] used machine learning to learn binary semantic information for function recognition, and tested it on obfuscated code, proving that it can resist commonly used obfuscation techniques. Tofighi-Shirazi et al. [14] proposed a static automatic deobfuscation tool through machine learning and binary analysis, which can remove opaque predicate confusion, and the accuracy rate can reach 98%.

In general, traditional methods require specific analysis of specific obfuscation techniques, and it is difficult to automatically apply to large-scale obfuscated code detection. In contrast, neural networks have the advantages of scalability and easy analysis. Moreover, the existing binary code similarity analysis technologies are difficult to deal with the obfuscated code effectively due to the interference of obfuscation techniques. Therefore, in this paper, we propose a novel approach for learning the semantic representation of obfuscated code 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 relationships of the function completely and clearly.
- We redesign the message aggregation algorithm and update algorithm on the basis of graph network, making it 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 better performance than existing state-of-the-art methods in downstream tasks.

The remaining part of this paper proceeds as follows: Section 2 mainly introduces existing obfuscation technologies, 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 and compares them with current state-of-the-art methods. And last, we make a conclusion in section 5.

2. Related works

2.1. *Obfuscation Techniques*

Existing obfuscation tools mainly include Obfuscator-LLVM[1], VMProtect[2], Themida[3], Tigress[4], etc. The most commonly used obfuscation techniques in malware [15] 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.

2.2. *Intermediate Representation*

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

At present, a lot of research work is carried out based on IR, such as code search [16], code optimization [17], binary translation [18], binary recompilation [19], binary semantic understanding [20], etc. As a fundamental work, many studies analyze the semantic representation of binary code from different perspectives. Among them, Ben-Nun et al. [18] provided a new concept XFG to describe the context flow information of IR, which leveraged the underlying control flow and data flow information. They can match or even surpass advanced methods using only simple LSTM [21] and pre-trained embeddings. Venkatakeerthy et al. [22] provided *IR2Vec*, a concise and scalable encoding infrastructure to represent programs as distributed embeddings in continuous space. This method takes symbolic and flow-aware embeddings to construct LLVM Entities and map them to real-valued distributed embeddings.

There is also some achievements on deobfuscation techniques based on LLVM IR. For instance, Garba et al. [23] provided *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 the intermediate representation. Experiments show that *SATURN* is very effective for obfuscation techniques such as constant unfolding, certain arithmetic-based opaque expressions, dead code insertions, bogus control flow, and integer encoding. But it is difficult to deal with more sophisticated obfuscation technologies such as virtualization.

2.3. *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 [24] and GAT [25] have shown satisfactory results in the fields of node classification, graph classification, and link prediction. As a variant of GCN, GraphSAGE [26] optimizes node-centered neighbor sampling instead of full-graph sampling, enabling it

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

to perform inductive learning and improve the efficiency of graph convolution greatly. With transformer [27] showing great potential in the field of neural networks, the attention mechanism has gradually been applied to graph neural networks [28–30].

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, Schlichtkrull el al. [31] proposed Relational-GCN for aggregating node information based on edge types, but this method cannot handle the multi-dimensional features on the edge well. In PNAConv [32] and Crystal Graph Conv [33], the edge information in the graph is also regarded as multi-dimensional features, and can be aggregated and updated together with node features. In EGNN [34], 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 becomes larger, making it less scalable. In addition, there are also methods to alternately learn the embeddings of nodes and edges in the graph [35–37]. These methods all achieve very good results for specific tasks but are not suitable for our task. Battaglia et al. [38] made a more general summary of the graph network. The proposed graph network framework not only considers the node information and edge information of the graph at the same time, but also introduces the global state information of the graph. The initial value of the global state can be regarded as a certain inherent property of the graph or an encoding vector of prior knowledge.

3. Proposed Approach

We have conducted a detailed study of various existing obfuscation techniques and found that most of them can be divided into two categories: structural obfuscation and data obfuscation. In structural obfuscation, there are mainly two ways. One is to construct 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 the control flow structure of the program, while the latter will disrupt the order of basic blocks completely. These obfuscation techniques make it difficult for existing binary code analysis technologies to judge similarity.

In structural obfuscation, no matter how the control flow changes, the semantic information of the program remains unchanged, 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. Given that the assembly contains only limited registers and varies in different hardware architectures, it is difficult to extract the data flow relationships. To simplify our work and make it more versatile, we perform data flow analysis at the intermediate representation level.

In this paper, we use the intermediate representation programs compiled from the source code as the dataset. After preprocessing, we can extract data flow relationships from the intermediate representation programs and build data transformation graphs (DTG) as our training dataset. Then, we automatically learn the dataflow semantic information of the function from DTG via *DFSGraph*. Finally, we perform downstream tasks to test the performance of our model. The overall framework is shown in figure 1.

3.1. 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 and extract each operand and its corresponding data type. On the basis of the functionality of operands, we divide them into local variable, global variable, immediate value, basic block label, and use *LVAR*, *GVAR*, *IMM*, and *LABEL* to represent it respectively. At the same time, we divide the oprands into

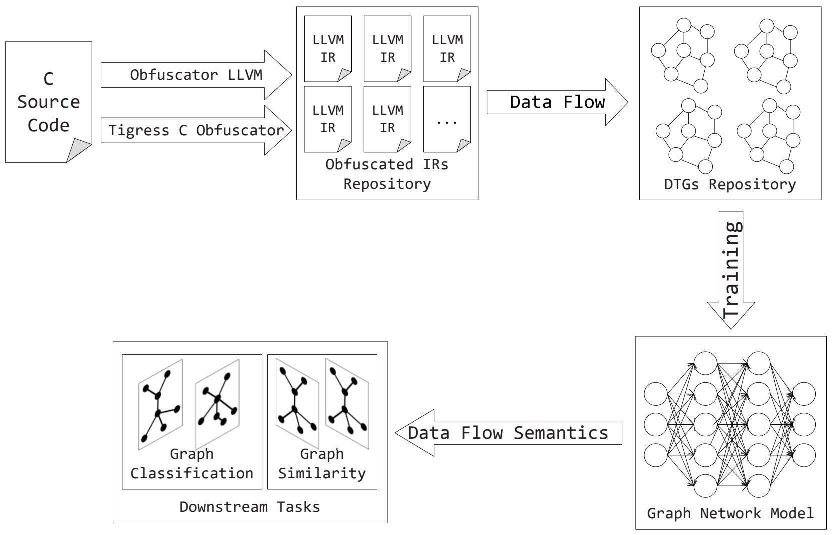


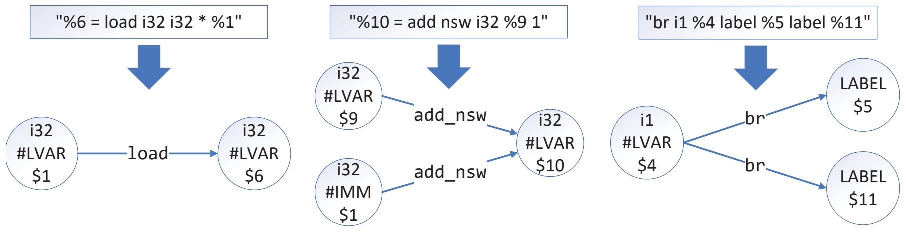
Figure 1. The overall framework of our approach.

i8, *i16*, *i32*, *i64*, *i128*, *float* depending on the varibale types. Since there are a large number of anonymous variables in the LLVM IR instruction, most of them are numbered and do not contain semantic information. Some variable names also contain functionality types, such as global variables, strings or arrays, etc. In the preprocessing stage, these variable names are standardized to keep away from the OOV issue, wipe out the obstruction brought about by irregular variable names, and simultaneously feature the functional type and data type of every variable. Therefore, each variable can be formally represented as:

$$\text{Type}\,\#\,\text{Class}\,\$\,\text{Num}$$

Where # and $ are delimiters, Type represents the data type, such as *i8*, *i16*, *i32*, *i64*, *i128*, *float*. Class refers to the functional category, such as *LVAR*, *GVAR*, *IMM*, *LABEL*. The value of *Num* is determined by Class. When the Class is *LVAR*, *GVAR*, and *LABEL* respectively, the value of *Num* is an integer greater than 0 as the variable number, and the initial variable number of each function starts from 1. and when the Class category is *LABEL*, the Type field can be omitted. When the Class category is *IMM*, the value of *Num* is the specific immediate value.

Besides, we divide the instructions of LLVM IR into various types according to the number of operands. These oprands can be divided into zero-element operators, unary operators, binary operators, and multivariate operators.



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

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

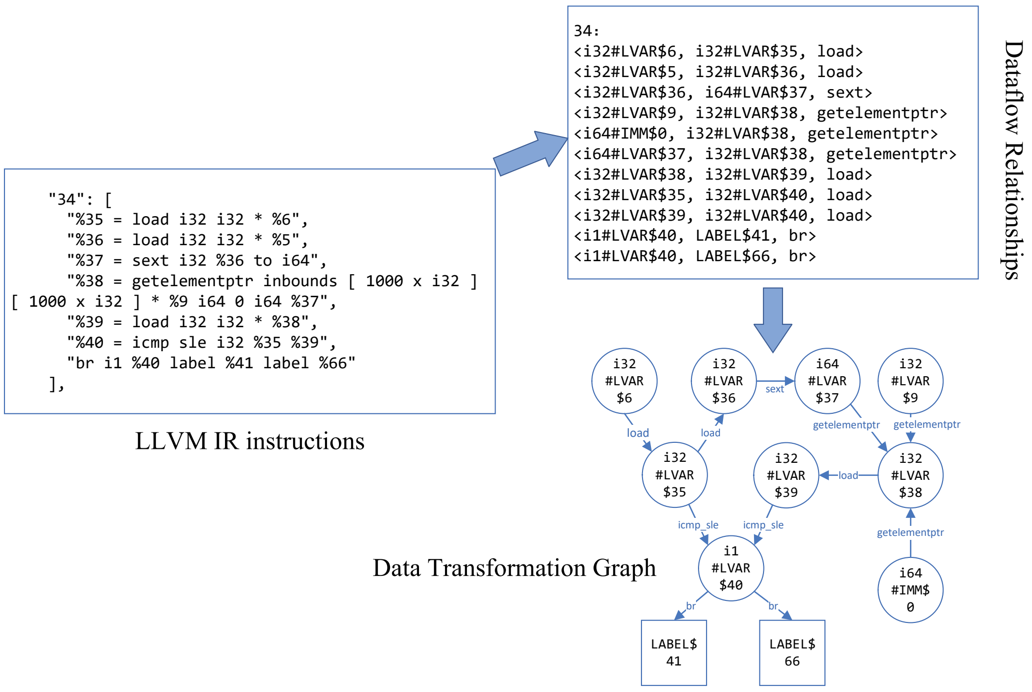
For zero-element operators, such as *alloca* and *ret*, since there is no data flow information transfer, they can be ignored directly. For the unary operators, we take "*%6 = load i32 i32 * %1*" as an example, and its data flow transformation is shown in Figure 2(a). For binary operators, we take "*%10 = add nsw i32 %9 1*" as an example, and its data flow transformation is shown in Figure 2(b). For multivariate operators, we take "*br i1 %4 label*

%5 label %11" as an example, and its data flow transformation is shown in Figure 2(c). In addition, table 1 lists the corresponding categories of all operators in LLVM IR.

Table 1. Categories of LLVM IR operators

Categories	Operators
Zero-element	ret, unreachable, fence, call, landingpad, catchpad, cleanuppad
Unary	resume, fneg, alloca, load, freeze
Binary	catchret, cleanupret, add, fadd, sub, fsub, mul, fmul, udiv, sdiv, fdiv, urem, srem, frem, shl, lshr, ashr, and, or, xor, extractelement, extractvalue, store, trunc..to, zext..to, sext..to, fptrunc..to, fpext..to, fptoui..to, fptosi..to, uitofp..to, sitofp..to, ptrtoint..to, inttoptr..to, bitcast..to, addrspacecast..to, icmp, fcmp, select
Multivariate	br, switch, indirectbr, invoke, callbr, catchswitch, insertelement, shufflevector, insertvalue, cmpxchg, atomicrmw, getelementptr, phi, va_arg

To simplify the parsing complexity, we use *%opcode* to replace nested instructions. For example, for nested instructions "*call @func (i8 * getelement ...)*", we denote it as: "*call @func %opcode*". Although this process will lose part of the data flow relationship, it is a trade-off between accuracy and complexity. Figure 3 is an example of the construction of a data transformation graph.



```
    "34": [
      "%35 = load i32 i32 * %6",
      "%36 = load i32 i32 * %5",
      "%37 = sext i32 %36 to i64",
      "%38 = getelementptr inbounds [ 1000 x i32 ]
[ 1000 x i32 ] * %9 i64 0 i64 %37",
      "%39 = load i32 i32 * %38",
      "%40 = icmp sle i32 %35 %39",
      "br i1 %40 label %41 label %66"
    ],
```

```
34:
<i32#LVAR$6, i32#LVAR$35, load>
<i32#LVAR$5, i32#LVAR$36, load>
<i32#LVAR$36, i64#LVAR$37, sext>
<i32#LVAR$9, i32#LVAR$38, getelementptr>
<i64#IMM$0, i32#LVAR$38, getelementptr>
<i64#LVAR$37, i32#LVAR$38, getelementptr>
<i32#LVAR$38, i32#LVAR$39, load>
<i32#LVAR$35, i32#LVAR$40, load>
<i32#LVAR$39, i32#LVAR$40, load>
<i1#LVAR$40, LABEL$41, br>
<i1#LVAR$40, LABEL$66, br>
```

Figure 3. An example of DTG.

3.2. Graph Network

Inspired by the Graph Network [38], we redesign 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. The whole update process is mainly divided into three parts.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 September 2022 doi:10.20944/preprints202209.0046.v2

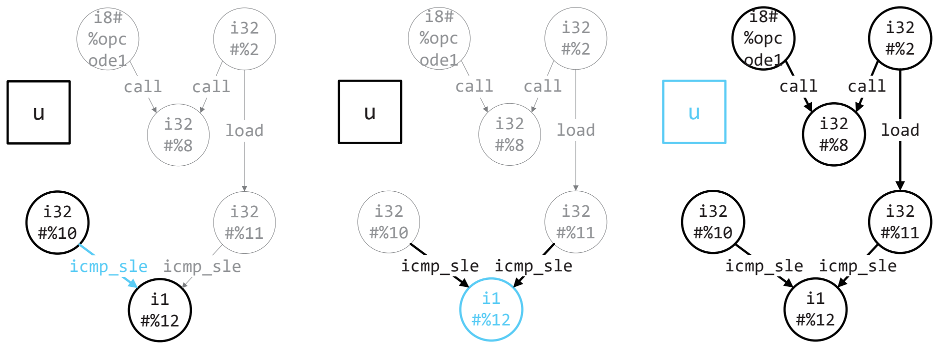


Figure 4. The update process of the *DFSGraph*.

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

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

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

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

where, $\|$ represents the concatenation operation of the vector, $W_h \in \mathbb{R}^{2D*M}$ and $W_u \in$ 206
$\mathbb{R}^{(D+M)*M}$ are the parameter matrixes. 207

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. The state update process of a node is formally described as:

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

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

where $\mathcal{N}(V_i)$ represents all predecessors 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 experimental verification, which is defined as follows:

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

Φ^h is the update function of the node, which is defined as follows:

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

where $W_e \in \mathbb{R}^{M*D}$ is the parameter matrix that maps edge features to node features. 208

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

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

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

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

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

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

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

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

where α is the forgetting coefficient. We set the initial value of the global state u to zero, that is: $u^0 = \vec{0}$. Then, we can embed DTGs using the current graph network model. 209 210

3.3. Model Training 211

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 through 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:

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

The margin represents the difference between the distance between $d(a,p)$ and $d(a,n)$. The larger the difference is, the more clearly the model can distinguish the positive and the negative samples. However, if the margin is too large, the model training process may be unstable. In the experiments in this section, it has been verified by experiments that when the value of margin is set to 50, it can increase the stability of the model training process while ensuring the accuracy of the model, so that the loss value of the model continues to decrease steadily. 212 213 214 215 216 217 218

In our implementation, we split the dataset into the training set and testing set at 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 256. The graph network is set to 3 layers. During the training phase, *AdamW* is used as an optimizer. The learning rate is set to 0.00005, making the step size of gradient descent smaller to prevent model oscillation or gradient explosion. 219 220 221 222 223 224

4. Experiments 225

In this section, we apply the obfuscated code similarity comparison task and the obfuscated technique classification task to evaluate the effectiveness of our proposed method respectively. 226 227 228

4.1. Dataset 229

At present, we take C programs collected from Google Code Jam from 2008 to 2020 as the original dataset, which contains 25,000 functions. And then we compiled them separately using the obfuscation options and their combinations in obfuscators O-LLVM and Tigress. All obfuscation technologies are shown in table 2. 230 231 232 233

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**. 234 235 236

Table 2. 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*

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 correct sample ranking at top N. So we use this metric to reflect the anti-obfuscation ability of the proposed model in the obfuscated code similarity comparison task.

For each obfuscation technique, we first randomly select 100 functions as the search set Γ. Then, we compile each function in Γ using a specific obfuscation technique. At the same time, we randomly select one function from Γ, compile it normally, and treat it as the function γ to be retrieved. At last, we calculate the Euclidean distance between function γ and each function in Γ one by one and sort them. Then we are able to get the rank of the obfuscation function corresponding to the function γ. Each experiment is repeated 100 times and we are able to get the probability of P@N.

The final results are shown in table 3. 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%. 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 transformations, such as *Virtualize*, *addOpaque16*, *EncodeArithmetic* in Tigress, the accuracy is relatively low. This may be because we are analyzing from the perspective of data flow relationship, so the proposed method is more sensitive to data flow confusion. Furthermore, it may be difficult to detect the equivalence of complex data flow transformation relationships only by relying on neural networks, so the accuracy drops slightly.

For the O-LLVM obfuscator, we conducted a comparative experiment with *Asm2Vec* [12], and the results are shown in Table 4. We can see that our method outperforms *Asm2Vec* in most cases. However, for the sub option, it may be because this option adds some complex data flow relationships, and *DFSGraph* does not perform well in detecting them, so the accuracy is slightly lower. The above results show that our method is partially resistant to data obfuscation and still needs to be further improved.

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 (BCELoss). The formula for BCELoss is as follows:

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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 September 2022 doi:10.20944/preprints202209.0046.v2

Table 3. 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

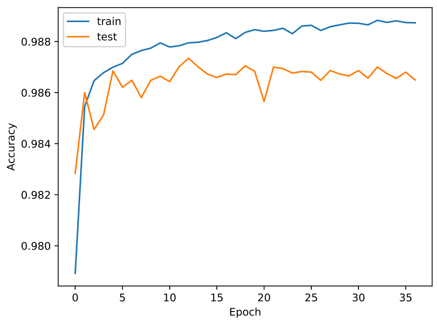
Table 4. Comparative experimental results between Asm2Vec and DFSGraph.

	sub	fla	bcf	sub+bcf	sub+fla	bcf+fla	sub+fla+bcf
Asm2Vec	**0.921**	0.871	0.856	0.820	0.782	0.729	0.653
DFSGraph	0.909	**0.910**	**0.903**	**0.899**	**0.913**	**0.872**	**0.870**

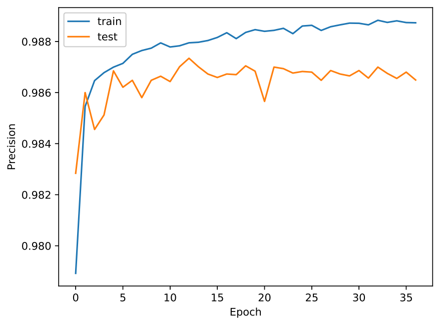
where l_n represents the loss value for the nth label, which can be described as:

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

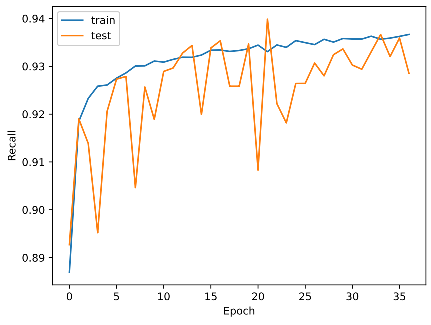
Next, we can retrain the parameters of our model. We evaluate our model using common performance metrics: accuracy, precision, recall, and F1-score.



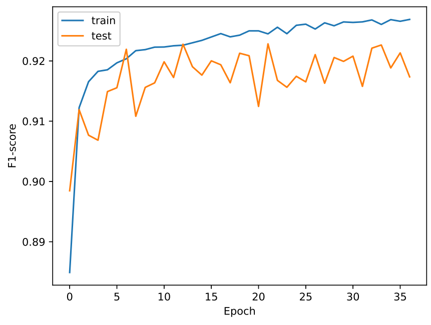
(a) Accuracy.



(b) Precision.



(c) Recall.



(d) F1-score.

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

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 can be stabilized at about 98.6%, indicating that our model can predict the obfuscation technology correctly with high probability. Meanwhile, we see that the recall and f1 values, although not very stable, are still able to stay above 90% after 10 epochs. 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 method.

For carrying out a comparative experiment with the method described in the literature [14] (denoted by *DefeatOP*), we first build the test dataset, which mainly consists of two parts. One half is unobfuscated functions and the others are generated by the obfuscation techniques that contain opaque predicates in Tigress. Then we use *DefeatOP* and *DFSGraph* to detect whether one function contains opaque predicates. The comparison experiment results are shown in Table 5. It can be seen from the results that the accuracy of our model for most obfuscation techniques is higher than *DefeatOP*. But for the addOpaque16 option, the accuracy of our model is slightly lower, which may also be due to too many opaque predicates, resulting in too complex data flow transformation relationships, which are difficult for our model to identify. Overall, this comparative experiment proves that *DFSGraph* can better identify obfuscated code containing opaque predicates in most cases.

Table 5. Comparative experimental results between DefeatOP and DFSGraph.

	addOpaque4	addOpaque16	EncodeArithmetic	EncodeLiterals
DefeatOP	98.2%	**96.3%**	95.8%	94.7%
DFSGraph	**99.2%**	95.8%	**97.2%**	**98.1%**

The above experiments show that our model can achieve satisfactory results, but it is still far from practical application. In our experiments, the 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 recover the complete data flow information of the binaries completely.

5. Conclusion

In this paper, we propose the data transformation graph based on LLVM IR for the first time. Then, we redesign the message passing algorithm and update algorithm based on the graph network model, and define it as *DFSGraph*. We build the obfuscated code dataset with the obfuscation options in the O-LLVM and Tigress, and use it to train our model. The experimental results prove that our model is quite resistant to most obfuscation techniques. Especially for structural obfuscation, our model has higher accuracy than the existing state-of-the-art methods 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.

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.

References

1. Tofighi-Shirazi, R.; Elbaz-Vincent, P.; Oppida, M.C.; ha Le, T. Dose: Deobfuscation based on semantic equivalence. *ACM International Conference Proceeding Series* **2018**, pp. 1–12. https://doi.org/10.1145/3289239.3289243.
2. Xu, D.; Ming, J.; Fu, Y.; Wu, D. VMhunt: A verifiable approach to partially-virtualized binary code simplification. *Proceedings of the ACM Conference on Computer and Communications Security* **2018**, pp. 442–458. https://doi.org/10.1145/3243734.3243827.
3. 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.
4. 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.
5. 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.
6. 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.
7. Eyrolles, N.; Goubin, L.; Videau, M. Defeating MBA-based Obfuscation **2016**. pp. 27–37.
8. 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.
9. 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.
10. Peng, D.; Zheng, S.; Li, Y.; Ke, G.; He, D.; Liu, T.Y. How could Neural Networks understand Programs? **2021**.
11. 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.
12. 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.

13. Wang, S.; Wang, P.; Wu, D. Semantics-aware machine learning for function recognition in binary code. *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017* **2017**, pp. 388–398. https://doi.org/10.1109/ICSME.2017.59.
14. Tofighi-Shirazi, R.; Asavoae, I.M.; Elbaz-Vincent, P.; Le, T.H. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. *SPRO 2019 - Proceedings of the 3rd ACM Workshop on Software Protection* **2019**, pp. 3–14. https://doi.org/10.1145/3338503.3357719.
15. You, I.; Yim, K. Malware Obfuscation Techniques: A Brief Survey. 2010, pp. 297–300. https://doi.org/10.1109/BWCCA.2010.85.
16. Yu, Z.; Zheng, W.; Wang, J.; Tang, Q.; Nie, S.; Wu, S. CodeCMR: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* **2020**, *2020-Decem*, 1–12.
17. Cummins, C.; Petoumenos, P.; Wang, Z.; Leather, H. End-to-End Deep Learning of Optimization Heuristics. In Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017, pp. 219–232. https://doi.org/10.1109/PACT.2017.24.
18. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. 2018, Vol. 2018-Decem, pp. 3585–3597.
19. Altinay, A.; Nash, J.; Kroes, T.; Rajasekaran, P.; Zhou, D.; Dabrowski, A.; Gens, D.; Na, Y.; Volckaert, S.; Giuffrida, C.; et al. BinRec: Dynamic binary lifting and recompilation. *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020* **2020**. BinRec, https://doi.org/10.1145/3342195.3387550.
20. Jain, P.; Jain, A.; Zhang, T.; Abbeel, P.; Gonzalez, J.; Stoica, I. Contrastive Code Representation Learning. Association for Computational Linguistics, 2021, pp. 5954–5971. https://doi.org/10.18653/v1/2021.emnlp-main.482.
21. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Computation* **1997**, *9*, 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.
22. 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.
23. 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.
24. Kipf, T.N.; Welling, M. Semi-Supervised Classification with Graph Convolutional Networks **2016**.
25. 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.
26. Hamilton, W.L.; Ying, R.; Leskovec, J. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems* **2017**, *2017-Decem*, 1025–1035.
27. 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.
28. Yun, S.; Jeong, M.; Kim, R.; Kang, J.; Kim, H.J. Graph Transformer Networks **2019**. [1911.06455].
29. 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].
30. 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].
31. 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.
32. Corso, G.; Cavalleri, L.; Beaini, D.; Liò, P.; Veličković, P. Principal Neighbourhood Aggregation for Graph Nets **2020**. [2004.05718].
33. 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.
34. 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.
35. 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.
36. Wang, Z.; Chen, J.; Chen, H. EGAT: Edge-Featured Graph Attention Network. In Proceedings of the Artificial Neural Networks and Machine Learning – ICANN 2021; Farkaš, I.; Masulli, P.; Otte, S.; Wermter, S., Eds.; Springer International Publishing: Cham, 2021; pp. 253–264.
37. Yang, Y.; Li, D. NENN: Incorporate Node and Edge Features in Graph Neural Networks. In Proceedings of the ACML, 2020.
38. 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**.