Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 4095481, 11 pages
https://doi.org/10.1155/2022/4095481

Research Article

SROBR: Semantic Representation of Obfuscation-Resilient Binary Code

Ke Tang, Zheng Shan, Fudong Liu, Yizhao Huang, Rongbo Sun, Meng Qiao, Chunyan Zhang, Jue Wang, and Hairen Gui

State Key Laboratory of Mathematical Engineering and Advanced Computing, China

Correspondence should be addressed to Zheng Shan; zzzhengming@163.com

Received 12 April 2022; Accepted 25 May 2022; Published 24 June 2022

Academic Editor: Lei Zhang

With the rapid development of information technology, the scale of software has increased exponentially. Binary code similarity detection technology plays an important role in many fields, such as detecting software plagiarism, vulnerabilities discovery, and copyright solution issues. Nevertheless, what cannot be ignored is that a variety of approaches to binary code semantic representation have been introduced recently, but few can catch up with existing code obfuscation techniques due to their maturing and extensive development. In order to solve this problem, we propose a new neural network model, named SROBR, which is a deep integration of natural language processing model and graph neural network. In SROBR, BERT is applied to capture sequence information of the binary code at the first place, and then GAT is utilized to capture the structural information. It combines natural language processing and graph neural network, which can capture the semantic information of binary programs while resisting obfuscation options in a more efficient way. Through binary code similarity detection task and obfuscated option classification task, the experimental results demonstrate that SROBR outperforms existing binary similarity detection methods in resisting obfuscation techniques.

1. Introduction

In recent years, researchers have shown an increased interest in detecting binary code similarity [1], which plays a pivotal role in program analysis. Binary code similarity detection is widely applied in many areas, including software plagiarism detection, automated vulnerability discovery, and malware identification. Existing research approaches have been intensively studied, and many remarkable achievements have been attained in this field. For example, Genius [2], Gemini [3], VulSeeker [4], InnerEye [5], Asm2Vec [6], SAFE [7], Mirrors [8], Codee [9], etc. all of these refer to the natural language processing method. They use their proposed model to embed the binary program and judge whether the binary program is similar based on the similarity of the embedded vectors.

Specifically, SAFE [7] draws on the natural language processing method, uses Word2Vec [10] to generate the embedding of instructions, then regards the assembly instructions as sequences, and uses a self-attention-based neural network to generate the embedding of binary functions. However, it did not take the structural characteristics of the binary code into explicit consideration and could represent the semantic information of the binary function preferably. Order-Matter [11] uses different models to obtain the embeddings for binary functions at three levels, semantically sensitive, structural sensitive, and order sensitive, respectively. In addition, it also defines two graph-level tasks for evaluation. DeepSemantic [12] uses BERT [13] model, which is the best model in the field of natural language processing. It mainly consists of two stages: In the pretraining stage, a general model suitable for downstream tasks is created. In the fine-tuning stage, specific models for specific

tasks based on the pretrained model are generated. OSCAR [14] is based on a hierarchical transformer [15] and uses LLVM IR to capture the context information of long code sequences. InnerEYE [5] focuses on the use of neural machine translation models to solve the task of binary code similarity comparison across instruction set architectures (ISA).

However, with the development of information technology, obfuscation technology is increasingly applied to binary programs, making binary code similarity detection more and more challenging. Most research scholars have not conducted in-depth exploration on this issue. Some researchers have explored the effectiveness of the proposed method in resisting obfuscation techniques, but none of them can completely defeat it.

For instance, LoPD [16] uses a deviation-based program equivalence checking method to evaluate whether the programs are similar and have a certain degree of obfuscation-resilient ability. FOSSIL [17] recognizes malicious code functions on the basis of resisting various obfuscation techniques. In Asm2Vec [6], it shares the idea of PV-DM algorithm to detect the similarity of binary programs and manifests certain obfuscation-resilient capabilities. However, all these methods do not provide specific solutions to existing obfuscation techniques.

In order to resist obfuscation techniques better in binary code similarity detection, we find that it is a feasible scheme to combine natural language processing methods with graph processing algorithms. Inspired by DeepSemantic [12] and Order-Matters [11], we propose a new deep neural network architecture that learns the deep semantic information of binary functions by combining BERT (Bidirectional Encoder Representations from Transformers) and GAT (graph attention network). Experiments have proved that it can resist the existing binary code obfuscation technology better than the existing models.

Our contributions are concluded as follows:

(1) We adopt a new neural network architecture, named SROBR, which combines natural language processing and graph data processing. It performs well in capturing the semantic information of binary functions, while resisting the existing obfuscation methods based on the semantic information of the functions

(2) We utilize BERT model to obtain the semantic embedding of assembly instructions, so that each assembly instruction contains richer contextual information, which makes the semantic representation more accurate

(3) We apply GAT to embed the control flow graph of binary function. Through graph attention layer, we could get the attention weights and update the semantic information of the basic blocks

The remaining part of this paper proceeds as follows: Section 2 mainly introduces related work and background knowledge. Section 3 describes the structure of our proposed model in detail. Section 4 shows the results of our experiments and compares with the baselines. Section 5 summarizes the full paper.

2. Related Work

2.1. Obfuscator-LLVM Options. O-LLVM [18] is a C/C++ compiler based on the LLVM framework and Clang toolchain. It modifies the program logic at the intermediate representation level to increase the complexity of the binary while ensuring that the semantics remain unchanged. This feature can not only protect the copyright of the software and prevent it from being disassembled and analyzed by others but also hide the actual purport of the software itself, thereby to carry out malicious behaviors.

There are three obfuscation options for application at present. The *SUB* (Instruction Substitution) option will replace part of the assembly instructions with equivalent code fragments without changing the structural information of control flow graph. The *BCF* (bogus control flow) option will generate bogus control flow by adding invalid edges and nodes of the function control flow graph. The *FLA* (Control Flow Flattening) option will use a complex hierarchical structure to reconstruct the control flow graph and fuse it into a linear structure while ensuring that its semantics remain unchanged.

We use a simple example to intuitively exemplify the effect of these three obfuscation options. The source code is shown in Figure 1. Figure 2 shows the different program control flow graph (CFG) by compiling the same source code with different obfuscation options. Figure 1 is the CFG compiled without any obfuscation options. Figure 1 uses the *sub* option, and its CFG structure has basically not unchanged. Only some instructions are replaced with more complex equivalent instructions. Figure 1 uses the *bcf* option, of which the CFG is very different from (a), and introduces a lot of false instruction blocks. Figure 1 uses the *fla* option to completely disrupt the basic blocks in the CFG. It is difficult to understand the semantics of the program by using traditional reverse analysis.

2.2. Existing Approaches. LoPD [16] proposes a deviation-based program equivalence checking method, which searches for any dissimilarity between two programs by finding an input that will lead these two programs to be having differently, either with different output states or with semantically different execution paths. However, this method will consume a lot of time and cannot be applied on a large scale, and the results may be distorted.

FOSSIL [17] integrates a range of syntactical, semantic, and behavioral features by using Bayesian network model, which can recognize the functions in the open source software for the malicious code. Besides, it has the ability to resist obfuscation options and compiler optimization options. But it did not focus on the similarity of obfuscated code.

Asm2Vec [6] proposes to model the control flow graph as multiple sequences by using random walk algorithm. Each sequence corresponds to a potential execution trace that contains linearly laid-out assembly instructions. Then, these sequences are taken as input, and the PV-DM [19] model are used for training to learn the semantic representation of the function. Converting CFG to sequences will lose the structural information of the program, which causes this model not capable of representing the semantic information

```
int main(int argc, char ** argv)
{
    unsigned int n = argv[0];
    unsigned int mod = n % 4;
    unsigned int result = 0;
    if (mod == 0) result = (n | 0xBAAAD0BF) * (2 ^ n);
    else if (mod == 1) result = (n & 0xBAAAD0BF) * (3 + n);
    else if (mod == 2) result = (n ^ 0xBAAAD0BF) * (4 | n);
    else result = (n + 0xBAAAD0BF) * (5 & n);
    return result;
}
```

Figure 1: The source code of test function.

of the function completely. Although this method shows certain resistance to obfuscation options, it cannot completely defeat the code obfuscation.

2.3. BERT. BERT [13] is currently the best performing pretraining model in the field of natural language processing. It is different from the traditional monodirectional language model or the shallow splicing of two monodirectional language models. Instead, its main architecture is a stack of transformer's encoders [15], each of these layers utilizes the self-attention mechanism to learn the semantic representation of natural language. BERT is actually a two-stage framework, including pretraining and fine-tuning. First, it performs pretraining on a large corpus to obtain a generalized representation and then fine-tune it for specific tasks. A large number of experiments have proved that this method can achieve good results and the same in the field of assembly language analysis. For example, Order-Matter [11] and DeepSemantic [12] have used the BERT model and achieved good results. Therefore, we will also use the BERT model to learn deeper semantic information in the binary function to resist the obfuscation options of O-LLVM.

2.4. GAT. Although traditional deep learning algorithms have been applied to extract the features of Euclidean spatial data with great success, its performance on dealing with the data that generated from non-Euclidean spaces in many practical scenarios is not satisfactory. This is because the graph is irregular, each graph has unordered nodes of variable size, and each node in the graph has a different number of adjacent nodes, which makes it difficult for existing deep learning algorithms to deal with it. In addition, a core assumption of existing deep learning algorithms is that the data samples are independent of each other. However, each data sample (node) has edges related to other real data samples (nodes) in the graph, and this point can be used to scout the interdependence relations between nodes.

In recent years, people have become more and more interested in the expansion of deep learning algorithms on graphs. Successfully driven by many factors, the researchers integrated the ideas of convolutional networks, recurrent networks, and deep autoencoders to define and design the neural network structure for processing graph data, which brought up the graph neural network.

Graph neural networks mainly include graph convolution networks (GCN) [20, 21], graph attention networks (GAT) [22], graph autoencoders (GAE), [23] etc. In the field of semantic representation of binary codes, Qiao et al. [24] and Massarelli et al. [25] use GCN for semantic embedding of functions, but considering that the CFG of functions is a directed graph, which will cause a certain loss of the structural information. To avoid this problem, we adopt the GAT model instead.

GAT is a spatial-based graph convolutional network. It uses the attention mechanism to determine the weights of neighbor nodes when aggregating feature information. The GAT [22] introduces a self-attention mechanism in the propagation process, and the hidden state of each node is calculated in consideration of its neighbor nodes. In the internal structure of the GAT network [22], it is a simple stack of graph attention layers. For the node pair (i, j) in each attention layer, the attention coefficient is calculated as

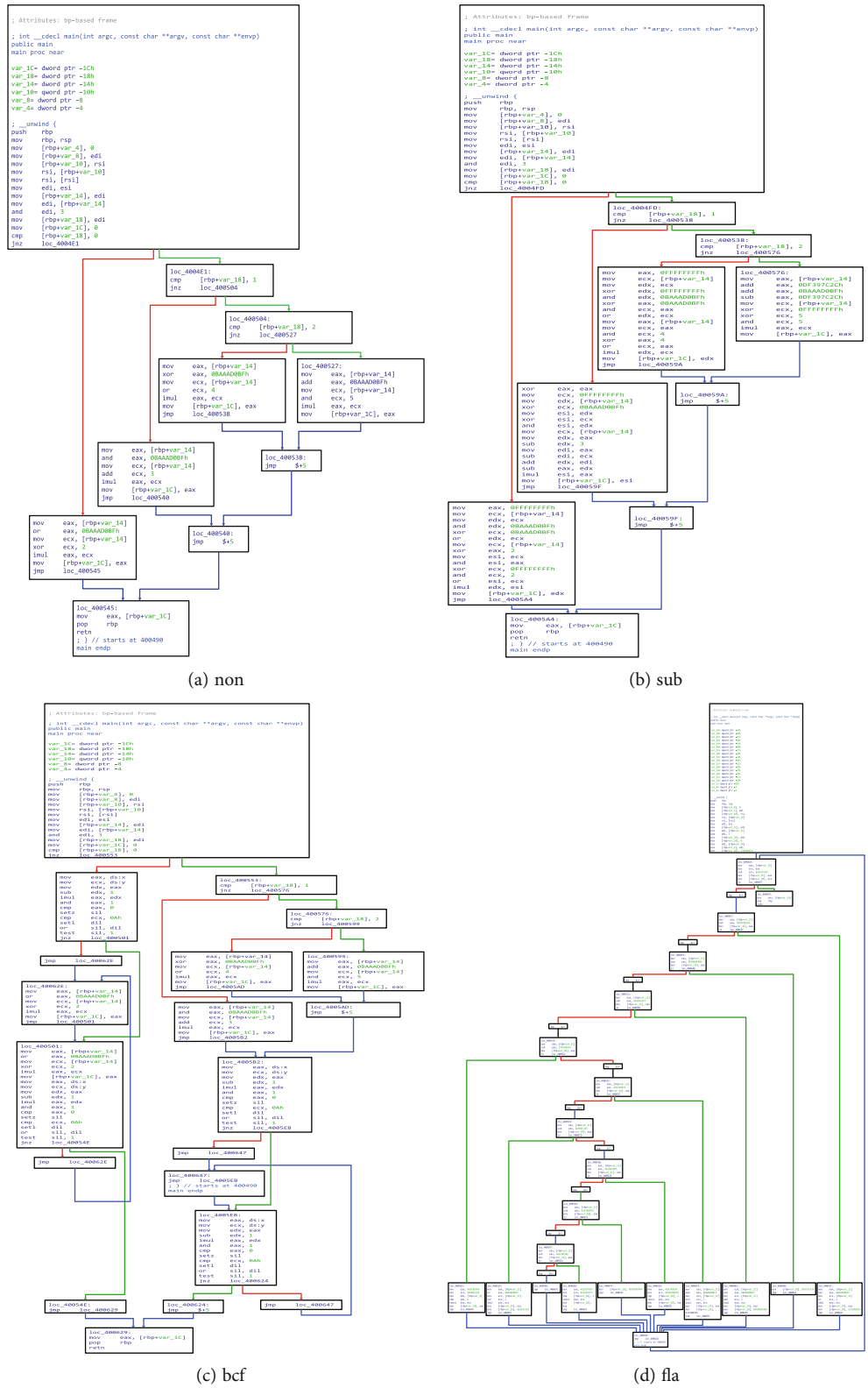
$$\alpha_{ij} = \frac{\exp\left(LeakyReLU\left(a^T\left[Wh_i \| Wh_j\right]\right)\right)}{\sum_{k\in N_i}\exp\left(LeakyReLU\left(a^T\left[Wh_i \| Wh_k\right]\right)\right)}, \quad (1)$$

where N_i represents the neighbors of node i, the input feature of the nodes is $h = \{h_1, h_2, h_3, \cdots, h_n\}$, $h_i \in \mathbb{R}^F$, and N, F represent the number of nodes and the feature dimension, respectively. The output feature of the nodes is $h' = \{h_1', h_2', h_3', \cdots, h_n'\}$, $h'_i \in \mathbb{R}^{F'}$, in which F' is the output feature dimension. $W \in R^{F'\times F}$ is the linear transformation weight matrix on each node, and $a \in R^{2F'}$ is the weight vector. Finally, the Softmax function is used for normalization, and Leaky*ReLU* is added to provide nonlinearity.

The finally output feature of the node i is

$$h_i' = \sigma\left(\sum_{j\in N_i}\alpha_{ij}Wh_j\right). \quad (2)$$

Multihead attention can be applied in GAT [22] to enhance the learning ability of the model. It applies k



(a) non (b) sub

(c) bcf (d) fla

Figure 2: CFGs compiled with different obfuscation options.

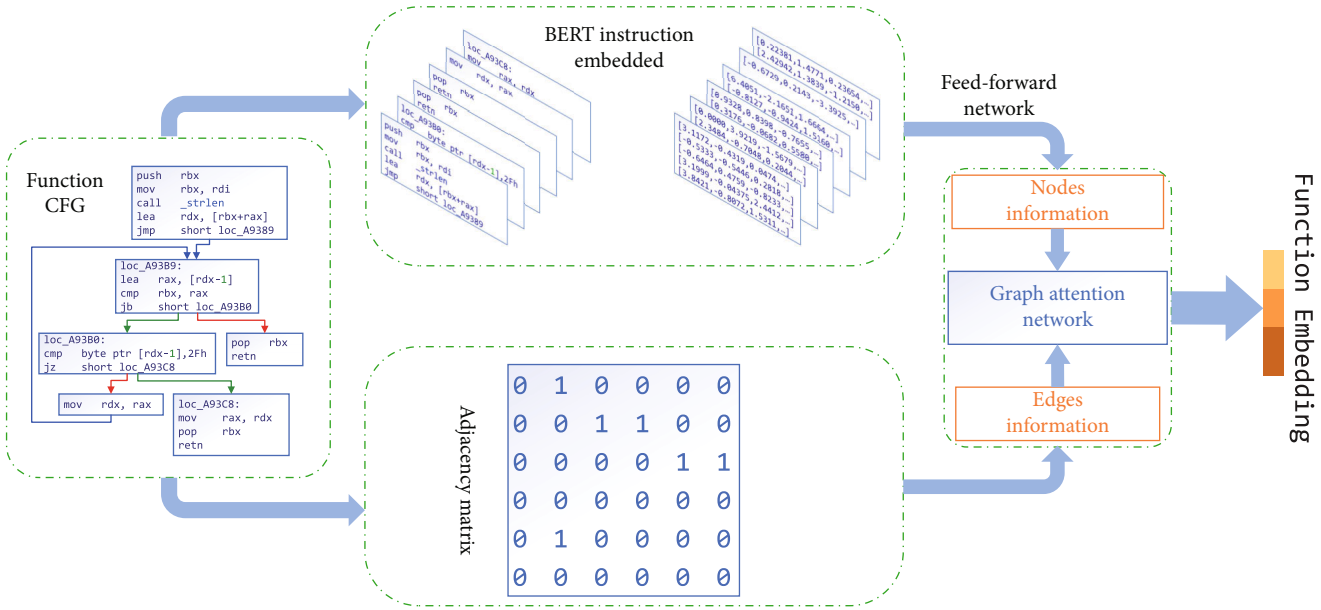


Figure 3: The overall architecture of SROBR.

independent attention mechanisms to calculate the hidden state and then stitches or averages the features, such as

$$h_i' = \Big\|_{k=1}^{K} \sigma\left(\sum_{j \in N_i} \alpha_{ij}^k W^k h_j\right), \tag{3}$$

or

$$h_i' = \sigma\left(\frac{1}{k}\sum_{k=1}^{K}\sum_{j \in N_i} \alpha_{ij}^k W^k h_j\right), \tag{4}$$

where α_{ij}^k is the attention coefficient of the kth attention head.

3. Model Design

3.1. Overview. The overall framework of SROBR is shown in Figure 3. The model architecture SROBR is mainly composed of three parts: The first part draws on the method of natural language processing, takes basic blocks as input, and uses the BERT model to obtain the instruction embedding according to the context information of the instruction. The second part uses feed-forward network to aggregate the embeddings of the instructions in the basic block and uses nonlinear mapping as the semantic representation of the basic block. The third part uses the graph attention neural network, takes the embeddings of the basic block as vertices, and uses the adjacency matrix of the control flow graph as the edges to obtain the high-dimensional vector containing the semantic information of the entire function. SROBR can be formally described as

$$\text{embdding}_f = \text{LayerNorm}\left(\text{GAT}\left(\text{FFN}\left(\sum_{i \in b}\text{BERT}(i)\right), \text{adj}_f\right)\right), \tag{5}$$

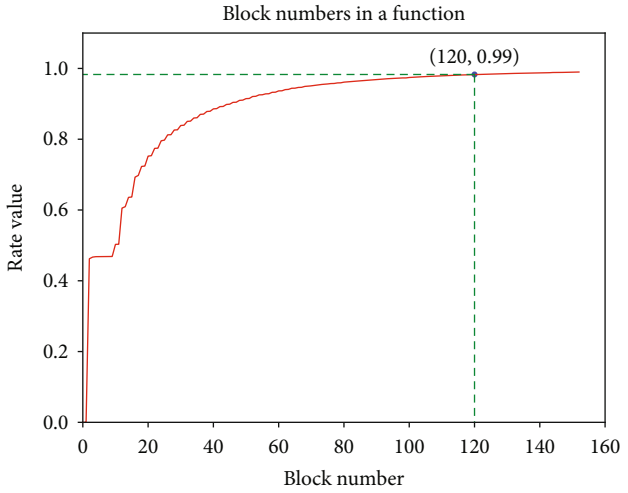


Figure 4: Basic block numbers.

where $i \in b$ represents all the instructions i in the basic block b, and adj_f represents the adjacency matrix of the basic block in the function f.

In general, SROBR takes the CFG of the binary function as input and obtains a high-dimensional vector to represent its semantic information. This digitized vector is proved to be resistant to obfuscation in subsequent experiments.

3.2. Data Preprocess. Through statistics on the basic information of our dataset, from Figure 4, we find that the number of basic blocks does not exceed 120 in more than 99% of functions. Since SROBR uses a graph neural network, too many basic blocks will lead to a sharp increase in memory usage. Limited by our hardware environment, we discard functions with more than 120 basic blocks. In addition, from Figure 5, we find that more than 99% of the basic blocks have no more than 40 instructions, so we define the maximum number of

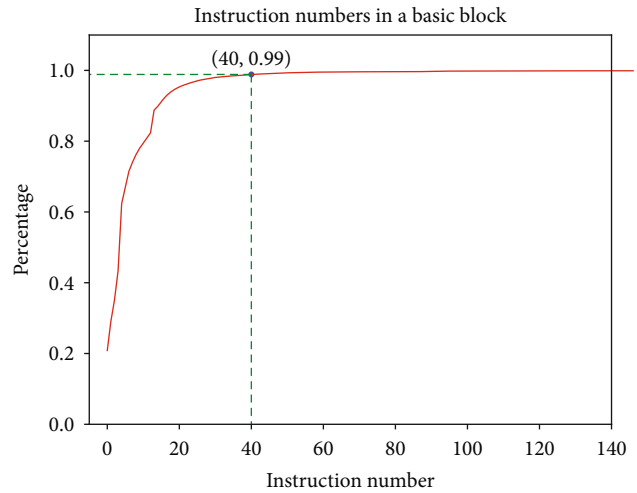


Figure 5: Instruction numbers.

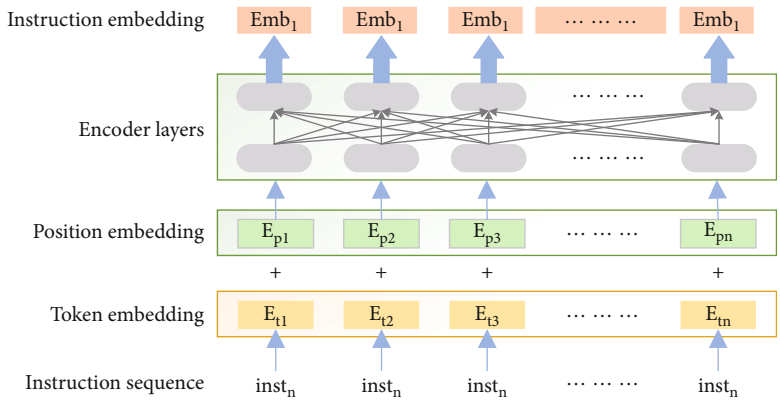


Figure 6: The structure of the instruction embedded module.

instructions in the basic block as 40. If the number of instructions in the basic blocks is more than 40, it will be truncated. Through processing, the amount of parameters can be significantly reduced, so that we can learn as much semantic information as possible while training.

3.3. Instruction Standardization. We learn from the natural language processing method to process the assemblers, in order to avoid OOV problems, we need to standardize the assembly instructions. In the standardization process, we first preprocess the assembly files to remove all comments, invalid characters, useless strings, and other unwanted content. For every instruction, it may contain various registers, memory addresses, variable names, immediate numbers, and other auxiliary information. We need to make a trade-off between the vocabulary size and the information retained.

Inspired by the normalization process in DeepSemantic [12], we propose the following regularization rules: we standardize the registers according to their categories and numbers of bits. For example, we replace "rax" with "reg_data_64," replace "edi" with "reg_addr_32," and replace "rbp" with "reg_pointer." In addition, the names of the variables in the instruction do not convey too much semantic information, so we use "var" to replace them. For immediate numbers, we use "imm" to replace them with. Through our instruction standardization process, although small amount of the information will be lost, it can significantly reduce the noise data,

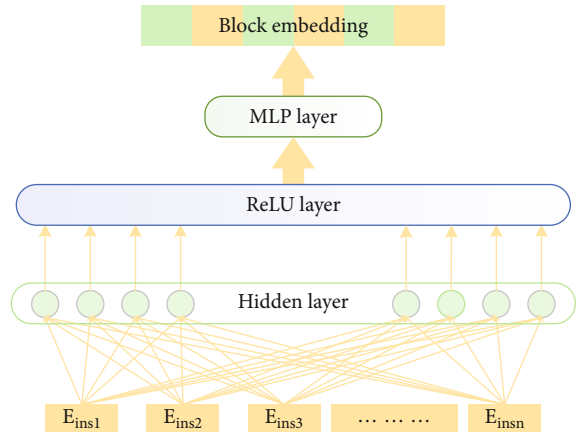


Figure 7: The structure of the feed-forward layer.

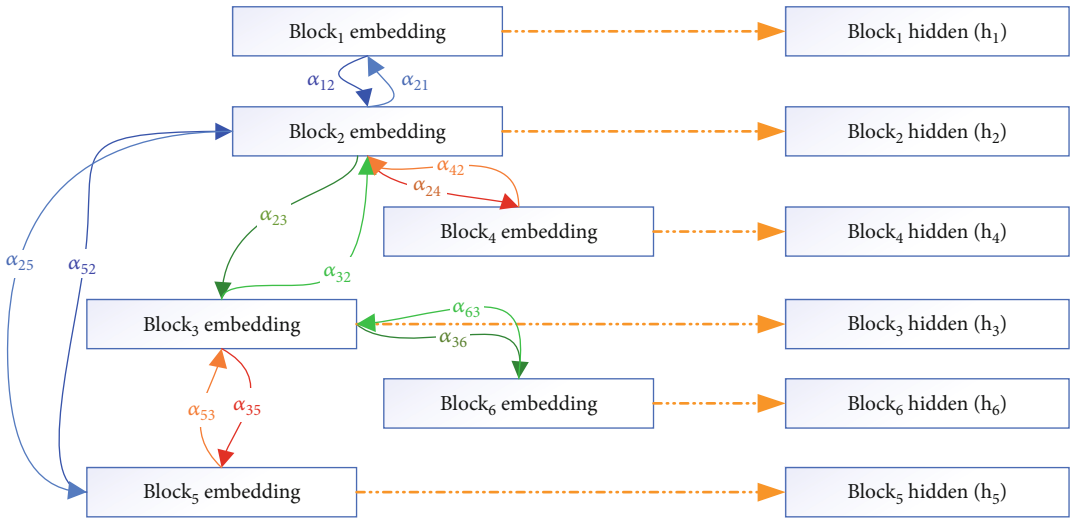


Figure 8: Graph attention weights propagation process.

which will make SROBR learn the semantic information of the function preferably.

3.4. Instruction Semantic Embedding. In the first step, we use BERT to embed the instructions in the basic block to obtain semantic information. In most papers that study the semantic representation of assembly language, such as "Asm2Vec," they generate assembly instruction embedding through word2vec. In this way, regardless of the CBOW or skip-gram method, the generated embedding is fixed and will not be affected by its contextual information. But in a basic block, the execution of the current instruction may be affected by the execution result of the previous instruction and may also affect the execution of the subsequent instruction. Therefore, we use the BERT pretraining model to embed the current instruction according to the context information of the instruction.

The structure of the assembly instruction embedded module is shown in Figure 6. In this module, each instruction in the basic block is taken as input, token embedding and positional embedding are added to educe the initial instruction vector, and the final output is attained through several encoder layers.

3.5. Block Semantic Embedding. Through the BERT model in the previous section, we get the semantic embedding of the instruction, and then we will get the semantic embedding of the basic block. Applying the feed-forward Neural network, with all the instruction embeddings in a basic block as input, we use the fully connected layer and the nonlinear mapping to semantically aggregate the instruction embeddings, so as to get the semantic information of the basic block. The basic principle is shown in Figure 7.

In this module, we first use the fully connected layer to map the embeddings to a higher-dimensional vector space and then use the RELU activation function as a nonlinear layer. Then, we use the fully connected layer to map to the original vector space and add the vectors as the embedding of the basic block.

3.6. Function Semantic Embedding. After getting the semantic embedding of the basic blocks, we are ready to obtain the basic block embedding at the function level. In this section, we will use graph attention neural network for training to obtain a vector representation containing the semantics of the entire binary function.

Graph attention neural network (GAT) introduces a self-attention mechanism in the propagation process, and the hidden state of each node is calculated by paying attention to its neighbor nodes. The control flow graph of a binary function is a directed graph, in which each node is a basic block. These nodes are linked by jump instructions. The execution of instructions in each basic block may be affected by neighbor blocks.

Therefore, we propose to use the attention mechanism to simulate the effect between blocks through attention weights, so as to restore the semantic information of the function better, even when the obfuscation technology is applied. Figure 8 visually shows the working principle of graph attention weights.

4. Experimental Evaluation

4.1. Dataset Collection. Like Asm2Vec [6], we also use commonly open source projects on github as our dataset, including 8 projects such as OpenSSL (https://github.com/openssl/openssl), libGmp (https://github.com/libtom/libtomcrypt), libTomCrypt (https://github.com/mirror/busybox), SQLite (https://github.com/curl/curl), Busybox (https://rada.re/n/radare2.html), Diffutils (http://angr.io/), Libcurl (https://rada.re/n/radare2.html), and Zlib (http://angr.io/). Specific details about the dataset are shown in Table 1.

There are two ways to generate assembler files from source code. One is to compile the source code into binary files and then use a disassembly tool (such as Radare2 (https://rada.re/n/radare2.html) or Angr (http://angr.io/)) to analyze the binaries to get the assembler files. The another one is to directly compile the source code into assemblers. The assembler files generated by the two methods are almost the same, but the latter is more convenient, so we choose the second method.

Table 1: Detailed description of our dataset.

	Function numbers	Block numbers	Instruction numbers
OpenSSL	11384	221564	1477384
LibGmp	8760	187600	1087912
LibTomCrypt	2088	54972	436844
SQLite	1464	29732	174616
Busybox	200	4264	21108
Diffutils	632	12252	63820
LibCurl	180	3936	23968
Zlib	728	16488	97784
Total	25436	530808	3383436

Then, we use the four obfuscation options of O-LLVM to compile, including non, *sub*, *fla*, *bcf* (non means that the obfuscation option is not used, and the source code is compiled normally). In this way, we can get the initial dataset, in which there are four binary functions with different obfuscation options for each source code.

When we conduct an in-depth research on the obtained dataset, we find that because some functions are too simple, regardless of whether it is obfuscated or not, the corresponding assemblies are exactly the same. Given that these functions may interfere with our subsequent model training, we just filter them. At last, we get more than 11000 functions for each obfuscation option.

4.2. Model Training. In this section, we apply our dataset to train the model. In SROBR, we use triple loss and stochastic gradient descent to pretrain.

In order to explore the effect of hyperparameters on the training effect of the model, we train the proposed model with different dimensions and compare the training results, as shown in Figures 9 and 10. From Figures 9 and 10, we can see that the training effect of the model gradually gets better with the increase of the dimension, but when the dimension reaches 768, the effect of the test loss is worse than that of the dimension 512. Therefore, we choose 512 as the dimension of the model.

In addition, in BERT module, we use 12 encoder layers, 8 attention heads, and dropout set 0.1. In GAT module, we use 8 graph attention layers, with dropout set 0.1 and alpha set 0.2, then we perform *LayerNorm* on the output. The main hyperparameters we set are shown in Table 2.

4.3. Evaluation. We use two tasks to evaluate our model. The first one is the binary similarity comparison task. In a round of comparison, we randomly sample 100 functions from all datasets as the current test set and randomly select a function from the current test set as the objective function. Then, we compile each function in the current test set without the obfuscation option to obtain the corresponding unobfuscated assembly functions as our set of search functions. At the same time, we compile the objective function with the specific obfuscation option to obtain the corresponding assembly function, as the target function. We use our pretrained model

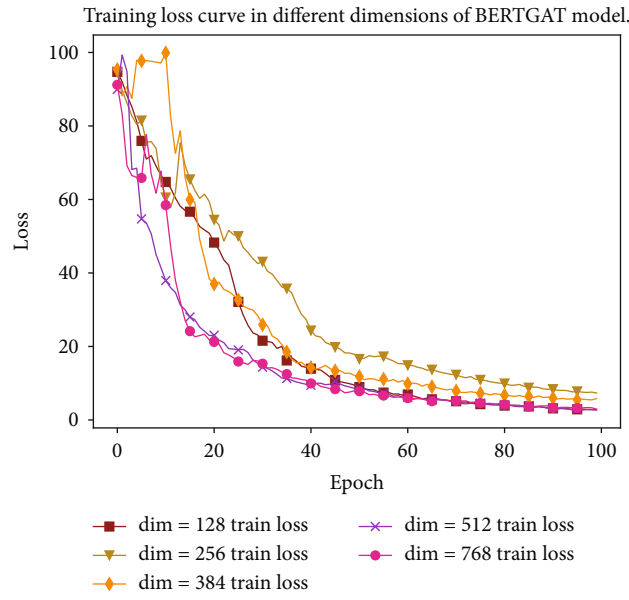


Figure 9: Training loss in different dimensions of SROBR model.

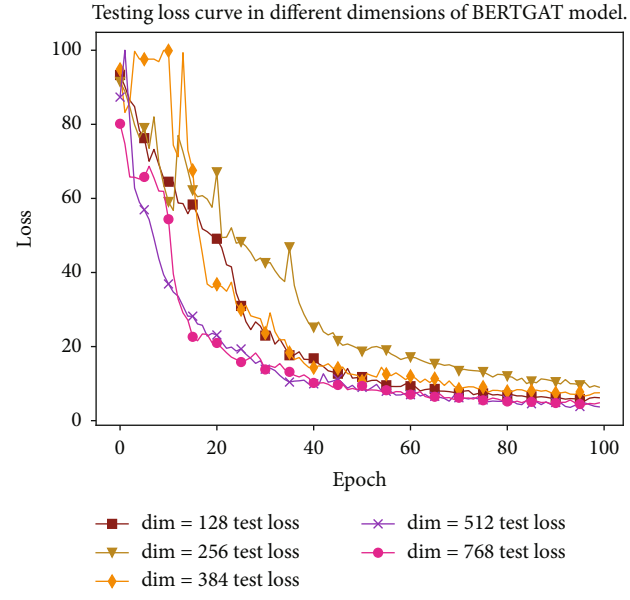


Figure 10: Testing loss in different dimensions of SROBR model.

to obtain the semantic embedding vectors of these functions, and compare the similarity between the target function and each function in the search function set according to the vector, and sort according to the similarity. Here, we use Euclidean distance as the similarity criterion. We assume that the vectors of the two functions are $f_1 = \{x_1, x_2, x_3, \cdots, x_n\}$ and $f_2 = \{y_1, y_2, y_3, \cdots, y_n\}$ respectively. Then, their Euclidean distance can be expressed as follows:

$$d = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{6}$$

Table 2: Hyperparameter settings.

Parameter	Value	Parameter	Value
Dimension of embeddings	512	Feed-forward network hidden size	1024
Number of encoder layers	12	Learning rate	0.0001
Number of attention heads	8	Dropout	0.1
Number of graph attention layers	8	Alpha	0.2
Number of graph attention layers	8	Epochs	100

The second one is the task of obfuscating options classification. We add a linear mapping on the basis of the original pretraining model and map the semantic embedding vector obtained previously to the obfuscating option label category. After fine-tuning the model, we use this model to classify the obfuscation options of the binary code.

4.3.1. Binary Function Similarity Comparison Task. In the binary code similarity comparison task, we evaluate SROBR according to the similarity ranking.

We conduct experiments with three obfuscation options (sub, fla, bcf) separately. In each experiment, we perform 1000 rounds of the previously mentioned comparison experiments to ensure the stability and robustness of the model. Through the model, we are able to gain function vectors representing its semantic information. By comparing the vector of the target function with each function vector in the search set, we can rank the functions in the search set according to the similarity. Here, we use $p@n$ to measure the accuracy of the model. The $p@n$ represents the probability that the target function ranks n in the search function set. In particular, we take n as 1, 3, and 5 to measure the pros and cons of the model.

In our experiments, we choose Asm2Vec [6] and SAFE [7] as our benchmark models. Tables 3, 4, and 5, respectively, correspond to the similarity comparison results of the obfuscated binary functions and normal binary functions. From the results, we can find that SROBR performs significantly better than SAFE and Asm2Vec in most cases. Howerver, in the *bcf* option, the Asm2Vec model is slightly better than SROBR, which may be due to the introduction of false basic blocks in the fake control flow confusion technology, which has an impact on the GAT module and causes a slight decrease in accuracy, the random walk algorithm in Asm2Vec just has a certain resistance to it.

4.3.2. Obfuscation Option Classification Task. In the task of obfuscation options classification, we use the previously trained model as a pretraining model and add a linear mapping on this basis, as our classification model:

$$\text{Classify} = \text{MLP}\left(\text{embedding}_f\right), \tag{7}$$

where embedding_f represents the embedding vector obtained by the pretraining model of the function f.

Then, we can fine-tune it by using our labeled dataset. Experiments have proved that high accuracy can be achieved after slight training.

Table 3: sub obfuscation option.

Model	p@1	p@3	p@5
SAFE	0.255	0.427	0.540
Asm2Vec	0.824	0.950	0.977
SROBR	0.903	0.980	0.993

Table 4: bcf obfuscation option.

Model	p@1	p@3	s p@5
SAFE	0.117	0.241	0.337
Asm2Vec	0.802	0.912	0.949
SROBR	0.701	0.878	0.982

Table 5: fla obfuscation option.

Model	p@1	p@3	p@5
SAFE	0.105	0.240	0.341
Asm2Vec	0.165	0.279	0.357
SROBR	0.690	0.881	0.940

We perform four classification tasks on four options. The training effect of the classification model is shown in Figure 11. From this figure, we can see that after 12 epochs of training, the classification accuracy has reached more than 95%. When the training reaches 20 rounds, the accuracy gradually stabilizes at around 98.7%. From Table 6, we can observe the accuracy, recall, and f1-score for each classification option.

From Table 6, we can find that the classification model achieves satisfactory results, which has been pretrained previously. From the perspectives of accuracy, recall, and f1-score, our model is able to capture the internal features of different obfuscation techniques well for accurate identification.

4.4. Ablation Experiments. For the sake of exploring the contribution of each part in SROBR to the overall model framework, we perform extensive ablation experiments, replacing the BERT or GAT module with other layers in the model framework, respectively. Then, we conduct the same training for each model variant and compare the model effects based on the results to analyze the role of each module.

Specifically, we will replace BERT module with linear layer or RNN models and replace GAT module with other

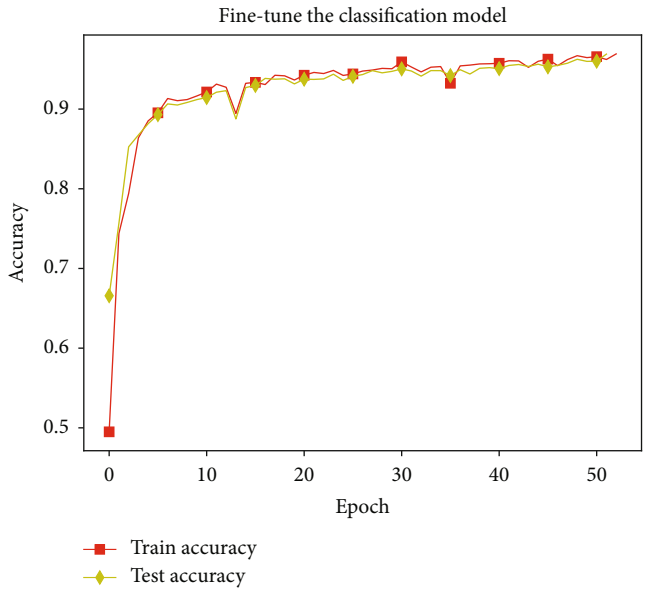


Figure 11: Fine-tune the classification model.

Table 6: Classification results for three obfuscation options.

	Precision	Recall	f1-score
Non	0.97	0.98	0.95
Sub	0.98	0.95	0.96
Fla	1.00	1.00	1.00
Bcf	1.00	0.99	1.00

Table 7: Experimental results of model variants.

options accuracy Variants	*sub*	*bcf*	*fla*
Linear+GAT	0.785	0.675	0.651
LSTM+GAT	0.759	0.577	0.531
BiLSTM+GAT	0.838	0.676	0.682
BERT+GCN	0.852	0.608	0.648
SROBR	0.888	0.701	0.694

Linear+GAT removes the BERT module, and the instruction vector is embedded only by random initialization. It is used to explore the role of the BERT module.
LSTM+GAT uses LSTM instead of BERT to generate the instruction embeddings in the basic block, which is used to compare the effects of BERT and RNN.
BiLSTM+GAT uses BiLSTM to further explore the pros and cons of recurrent neural networks and BERT.
BERT+GCN replaces the GAT module with GCN without using attention weights.

graph neural networks. Since LSTM excels in multiple domains [26–28], we choose it as a comparative experiment.

For all variant models, we train them in the same way. Then, we use the trained model to evaluate on the test set. For each variant, we use three obfuscation options, with the same method in Section 4.3.1 to compare its accuracy according to p@1. The results are shown in Table 7. From the results, we can clearly see that in terms of antialiasing ability, the BERT model is significantly better than the linear model and the recurrent neural network, and the GAT module is better than GCN.

5. Conclusion

In this paper, we propose a novel neural network structure SROBR for obfuscated code to obtain the semantic embedding representation of binary functions. It mainly contains three submodules. The first submodule uses BERT module to capture sequence information to generate instruction embeddings. The second submodule aggregates the instruction embeddings in basic block through FFN to obtain the embedding of the basic block. The third submodule gets the structural information of the function through basic blocks and the adjacency matrix, thereby obtaining the semantic vector of the entire function.

Through extensive comparative experiments on our dataset, we have proved that the proposed model can better deal with the obfuscation options of O-LLVM. However, other obfuscation options are not used for verification, which can be further done as our future research direction.

Data Availability

Our dataset can be available from the GitHub repositories, which mainly includes 9 projects, such as OpenSSL (https://github.com/openssl/openssl), libGmp (https://github.com/sethtroisi/libgmp), libTomCrypt (https://github.com/libtom/libtomcrypt), SQLite (https://github.com/sqlite/sqlite), Busybox (https://github.com/mirror/busybox), Coreutils (https://github.com/coreutils/coreutils), Diffutils (https://www.gnu.org/software/diffutils/), Libcurl (https://github.com/curl/curl), and Zlib (https://github.com/madler/zlib).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by Natural Science Foundation of China under Grant no. 61802435.

References

[1] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys*, vol. 54, no. 3, 2021.

[2] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[3] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network- based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the ACM Conference on Computer and Communica- tions Security*, pp. 363–376, Dallas Texas USA, 2017.

[4] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 896–899, Montpellier France, 2018.

[5] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, California, 2019.

[6] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489, San Diego, California, 2019.

[7] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2019*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds., vol. 11543 of Lecture Notes in Computer Science, pp. 309–329, Springer, Cham, 2019.

[8] X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma, "Similarity metric method for binary basic blocks of cross-instruction set architecture," in *Proceedings 2020 Workshop on Binary Analysis Research*, San Diego, California, 2020.

[9] J. Yang, C. Fu, X. Y. Liu, H. Yin, and P. Zhou, "Codee: a tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, p. 1, 2021.

[10] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Informa- tion Processing Systems - Volume 2. NIPS'13*, pp. 3111–3119, Red Hook, NY, USA, 2013.

[11] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1145–1152, New York, 2020.

[12] H. Koo, S. Park, D. Choi, and T. Kim, "Semantic-aware binary code representation with bert," 2021, https://arxiv.org/abs/2106.05478.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," 2018, https://arxiv.org/abs/1810.04805.

[14] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?," in *International Conference on Machine Learning*, pp. 8476–8486, 2021.

[15] A. Vaswani, N. Shazeer, N. Parmar et al., "Attention is all you need," *Advances in Neural Information Processing Systems*, pp. 5999–6009, 2017.

[16] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1647–1664, 2016.

[17] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "FOSSIL: a resilient and e_cient system for identifying FOSS functions in malware binaries," *ACM Transactions on Privacy and Security*, vol. 21, no. 2, pp. 1–34, 2018.

[18] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*, pp. 3–9, Florence, Italy, 2015.

[19] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, pp. 1310–4546, 2013.

[20] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, https://arxiv.org/abs/1609.02907.

[21] Y. Huang, M. Qiao, F. Liu, X. Li, H. Gui, and C. Zhang, "Binary code traceability of multigranularity information fusion from the perspective of software genes," *Computers & Security*, vol. 114, article 102607, 2022.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2018, https://arxiv.org/abs/1710.10903.

[23] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: methods and applications," 2017, https://arxiv.org/abs/1709.05584.

[24] M. Qiao, X. Zhang, H. Sun et al., "Multi-level cross-architecture binary code similarity metric," *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8603–8615, 2021.

[25] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings 2019 Workshop on Binary Analysis Research*, pp. 21–24, San Diego, California, 2019.

[26] L. Lv, Z. Wu, J. Zhang, Z. Tan, L. Zhang, and Z. Tian, "A vmd and lstm based hybrid model of load forecasting for power grid security," *IEEE Transactions on Industrial Informatics*, p. 1, 2021.

[27] L. Zhang, C. Xu, Y. Gao, Y. Han, X. Du, and Z. Tian, "Improved dota2 lineup recommendation model based on a bidirectional lstm," *Tsinghua Science and Technology*, vol. 25, pp. 712–720, 2020.

[28] L. Zhang, Z. Huang, W. Liu, Z. Guo, and Z. Zhang, "Weather radar echo prediction method based on convolution neural network and long short- term memory networks for sustainable e-agriculture," *Journal of Cleaner Production*, vol. 298, 2021.