Chinese Journal of Electronics
vol. 33, no. 1, pp. 128–138, January 2024
https://doi.org/10.23919/cje.2022.00.228

RESEARCH ARTICLE

Fast Cross-Platform Binary Code Similarity Detection Framework Based on CFGs Taking Advantage of NLP and Inductive GNN

Jinxue PENG, Yong WANG, Jingfeng XUE, and Zhenyan LIU

School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China

Corresponding author: Yong WANG, Email: wangyong@bit.edu.cn
Manuscript Received March 22, 2022; Accepted January 5, 2023
Copyright © 2024 Chinese Institute of Electronics

Abstract — Cross-platform binary code similarity detection aims at detecting whether two or more pieces of binary code are similar or not. Existing approaches that combine control flow graphs (CFGs)-based function representation and graph convolutional network (GCN)-based similarity analysis are the best-performing ones. Due to a large amount of convolutional computation and the loss of structural information, the use of convolution networks will inevitably bring problems such as high overhead and sometimes inaccuracy. To address these issues, we propose a fast cross-platform binary code similarity detection framework that takes advantage of natural language processing (NLP) and inductive graph neural network (GNN) for basic blocks embedding and function representation respectively by simulating extracting structural features and temporal features. GNN's node-centric and small batch is a suitable training way for large CFGs, it can greatly reduce computational overhead. Various NLP basic block embedding models and GNNs are evaluated. Experimental results show that the scheme with long short term memory (LSTM) for basic blocks embedding and inductive learning-based GraphSAGE(GAE) for function representation outperforms the state-of-the-art works. In our framework, we can take only 45% overhead. Improve efficiency significantly with a small performance trade-off.
Keywords — Control flow graph, Natural language processing, Inductive graph neural network, Binary code similarity detection.
Citation — Jinxue PENG, Yong WANG, Jingfeng XUE, *et al.*, "Fast Cross-Platform Binary Code Similarity Detection Framework Based on CFGs Taking Advantage of NLP and Inductive GNN," *Chinese Journal of Electronics*, vol. 33, no. 1, pp. 128–138, 2024. doi: 10.23919/cje.2022.00.228.

I. Introduction

With the development of smartphones and the Internet of things (IoT), the number of software is increasing exponentially, while the number of malware is also increasing recklessly. Statistics show that more than 100,000 malicious apps were uploaded to different platforms (a variety of operating systems and hardware environments) such as 9game and Google in 2019 [1]. The malware hides in benign ones, controlling execution and stealing information [2]. What's worse, IoT vendors are compiling and deploying third-party code developed on multiple platforms, which poses a huge challenge to our detection efforts.

The core part of malware detection is cross-platform binary code similarity detection, which can be can be applied at different granularities, such as instructions, basic blocks, functions, and whole programs [3]. In addition, cross-platform binary code similarity detection can be also used in vulnerability search [4]–[9] and patch analysis [10]–[14], etc. In real-world scenarios, the amount of data can be as high as one million or more. Over time, there will be problems such as excessive data volume and model aging. However, model updating requires excessive resource and time costs, so improving model efficiency is crucial.

Current solutions can be roughly divided into three categories: the graph-matching based [15]–[20], the graph-embedding based [21]–[25], and the deep-learning based [26]–[30]. The graph matching-based approaches detect whether two binary functions' control flow graphs (CFGs) are similar, which have much higher time complexity and

Associate Editor: Prof. Zhihua ZHOU, Nanjing University.

weaker portability [15]. The graph embedding-based solutions map binary code into multi-dimensional vector representations (embeddings), and then use vectors instead of binary code snippets [27]. However, the features of CFGs are usually manually defined, which will probably cause bias. An end-to-end similarity detection method based on graph convolutional network (GCN) is proposed to fix this defect, and the biggest improvement is to use unsupervised feature extraction to replace manual feature extraction. But the following problems still exist:

- Low performance, high overhead. In GCN-based approaches [22], [23], [26]–[31], binary functions are represented as structure-regular images or sequences, and then analyzed with convolutional neural networks (CNN). However, because of CFG's irregular and unstructured natures, the convolutional operations can't be performed effectively. GCN is a kind of transductive learning, which learns the nodes embeddings in a deterministic graph and the eigenvalue decomposition of the graph's Laplacian matrix. We find that the total runtime of [26] exceeds 10 hours when we reproduce it on our own server. For larger CFGs of complex binary functions, the time cost of the traversing subgraphs, the computational cost of the model training, and the storage cost become more uncontrollable.
- The loss of control flow structural information. Control flow semantic information represents the internal structure of a function, reflecting the dependencies between basic blocks and their contextual semantics, which is essential for distinguishing malware from different platforms [32]. Unfortunately, the structural information is not fully utilized in some approaches, since many graph embedding methods depend on the random walk strategy during the representation of CFGs. But the random walk cannot obtain all the structural information of the central node. What's worse, the loss of structural information may affect the detection accuracy.

To address the above issues, we propose a fast binary code similarity detection framework that combines NLP with inductive GNN. We solve the first problem by changing the neighbor sampling strategy and filtering invalid nodes. And for the second problem, we combine NLP and GNNs together. NLP can automatically learn the representations needed from raw data with the smallest possible human bias. Unlike transductive learning, the feature learning of each node in inductive representation learning is only related to its K-order neighbors instead of considering the full graph information, which makes distributed learning of large-scale graph data possible. To the best of our knowledge, the graph computing framework Plato proposed by Tencent can reduce the computation time of a super-scale graph with 1 billion nodes to the level of minutes.

In this paper, we investigate several approaches to embed the blocks of the CFGs. Specifically, we disassemble the binary code with third-party tools and extract the instruction sequence, CFG, and adjacency matrix (Adj) of binary functions. Then we successively perform instruction embedding, basic block embedding, and function representation using specific NLP models and GNNs. We evaluate our framework on the task of compiler provenance, which is first afforded by Rosenblum *et al.* [33]. Our framework can be adapted to various NLP models and GNNs. In our experiment, LSTM [34], CodBERT [35], and without-embedding are used in basic block embedding and graph attention network (GAT) [36] and GAE [37] for function representing. Finally, we form six implementation schemes in total. The work of [26] is our baseline which is the subsequent work of [27]. Experimental results show that the scheme of LSTM for basic block embedding and inductive learning-based GAE for function representation performs best, which takes only 45% of the time overhead of the baseline.

The main contributions of our work are as follows:

1) We propose a cross-platform binary code similarity detection framework, that combines NLP and GNN to solve the problems of high overhead and structural information loss in existing works;

2) We represent the blocks of CFGs using an inductive representing learning-based approach;

3) We extensively evaluate the framework by six different NLP models and GNNs, showing that LSTM for basic block embedding and GAE for function representation performs best. We can Improve efficiency significantly with a small performance trade-off;

4) We fully preserve and utilize the structural information of the CFG and analyze the final result in Section V (the section Discussion).

II. Related Work

While there has been a series of efforts on binary code similarity detection, most of them only work on binary code from a single platform. Due to the development of IoT, security practitioners have to focus on the problems caused by cross-platform applications. All the research can be divided into the following categories.

Graph matching-based solutions Those approaches based on graph match use the edit distance of two CFGs as the similarity value of the corresponding function pair. Edit distance refers to the minimum number of edits converted from one graph to another, and it is extended from string edit distance [16]. BinGold [17] extracts semantic information from CFGs and data flow graphs (DFGs) and performs similarity calculation using the graph match algorithm. BINGO [18] introduces function filtering before the similarity calculation to significantly reduce irrelevant target functions. discovRE [19] and SIGMA [20] detect whether the two graphs are completely similar first. If they are identical, the detection ends here and the edit distance between the two graphs is 0. Otherwise, discovRE uses the distance to prune and guide the next operation, and finally arranges the candidate functions with a distance less than the threshold in the order from smallest to largest (the similarity is from

largest to smallest), that is the similarity matching result. SIGMA executes fuzzy matching, calculating the edit distance of the graphs and taking it as the final result.

Graph embedding-based solutions Graph match is not only computationally expensive and time-consuming but also difficult to be applied to new tasks. In order to represent graphs in a computationally convenient form, Genius [21] takes advantage of graph embedding to convert CFGs into high-dimensional vectors. Gemini [22] improves Genius by using Structure2Vec [23] for graph embedding. Structure2Vec can capture the spatial structure similarity between two nodes. The concept of annotated CFG (ACFG) is first proposed in Gemini, which represents a graph containing manually selected features. Different from Gemini, VulSeeker [24] inputs the CFGs and DFGs of binary functions into the Structure2Vec to jointly guide the feature learning of nodes. VulSeeker-pro [25] improves VulSeeker by enhancing function semantic emulation based on semantic learning.

Deep learning-based solutions It is worth mentioning that the features in CFGs discussed above are usually manually defined, which inevitably introduces human bias. In the last few years, researchers have focused on tackling the problem of autonomous feature learning. αDiff [29] extracts the internal features after binary pre-processing by feeding the function directly into CNN. SAFE [27] and Asm2vec [28] use an unsupervised feature learning-based solution to train the model by treating each assembly instruction's operand as a token. Yu [30] combine semantic representation with structure representation, using recurrent neural network (RNN), message passing neural network (MPNN), and CNN to extract semantic, structure, and sequence information, which increases the F1 value to nearly 5% higher than Gemini.

III. Problem Definition and Solution

1. Problem statement

For easy reading, the symbols used for classification task in this article are shown as follows:

– A source code S and the binary code S';

– A function f belongs to S;

– A set of compile C;

– A binary code $b^s_{c_j}$ belongs to S and compiled by c_j;

– A graph $G=(V,E)$ where the the set of vertices V and the set of edge E; $p\&q$ is the number of nodes and edges in G;

– Positive integer set $\mathbf{N}^*$;

– Real-number set $\mathbf{R}$;

– The number of instructions in b is m;

– A set of instructions in f is I_f, n instructions in total;

– A set of instructions in b is I_{v_i}, mapping vector $\vec{\boldsymbol{I}}_{\boldsymbol{v_i}}$;

– A set of the multi-dimensional feature vectors of all blocks in G in $\vec{\boldsymbol{B}}$;

– The multi-dimensional of b is $\vec{\boldsymbol{B}}_{\boldsymbol{v_i}}$;

– A set of invalid nodes $\vec{\boldsymbol{B}}_{\boldsymbol{o}}$;

– A set of true labels of blocks T;

– The multi-dimensional feature vector of f is $\vec{\boldsymbol{F}}_{\boldsymbol{v_i}}$;

– The sampling depth in GNN K, the number of node in i-th layer is K_i.

Our framework is on the basic block level. The cross-platform binary code similarity analysis at the basic block level can be defined as two binary blocks b^s_1, b^s_2 compiled on different platforms that are similar if they are compiled by different compilers C: $\{c_1, c_2, \ldots\}$ but belong to the same original source code S. The difficulty lies in the fact that different CPU architectures different program versions and different compilers with four optimization levels may produce vastly different binary function segments [38].

Essentially, our framework is a transformation that maps a binary basic block b to a corresponding embedding. We are given a set of possible compilers C: $\{c_1, c_2, \ldots\}$, and a binary basic block b_{c_j} (the binary basic block b compiled by c_j) and we have to judge the compiler family $Y(c_j)|c_j \in C$.

Control flow graph (CFG) is the representation form of code during the compilation process. Denoting a CFG as $G=(V,E)$ where $V=(v_1,\ldots,v_p), v_j \in \mathbf{N}^*, 1 \le j \le p$ and $E=(e_1,\ldots,e_q), e_j \in \mathbf{N}^*, 1 \le j \le q$ are the set of vertices (basic blocks) and edges in G. We indicate the list of assembly instructions composing function f_1 with $I_{f_1}=(i_1,\ldots,i_n), i_j \in \mathbf{N}^*, 1 \le j \le n$, and represent the list of assembly instructions contain in v_i with $I_{v_i}=(i_1, \ldots, i_m), i_j \in \mathbf{N}^*, 1 \le j \le m$. We want to represent v_i as a vector in $\mathbf{R}$.

2. Solution overview

To achieve cross-platform binary code similarity detection with lower overhead and higher accuracy. We propose a fast detection framework which combines NLP and inductive GNN. As shown in Figure 1, the whole detection process includes four phases.

1) Pre-processing. We use Radare2 [39] to analyze the binary code S' and get the instruction sequences I_f, CFG and Adj of f.

2) Basic block embedding. This phase aims at obtaining the multi-dimensional feature vector $\vec{\boldsymbol{B}}_{\boldsymbol{v_i}}=(\vec{\boldsymbol{i}}_{\boldsymbol{1}}, \ldots, \vec{\boldsymbol{i}}_{\boldsymbol{m}}), \vec{\boldsymbol{i}}_{\boldsymbol{j}} \in \boldsymbol{R}^{\boldsymbol{L}}, 1 \le j \le m$ associated with basic block v_i. L is the the dimension of $\vec{\boldsymbol{B}}_{\boldsymbol{v_i}}$. We embed the instructions in I_{v_i} by word2vec [40] in NLP. This model can be represented by a mapping function $h: I_{v_i} \to \vec{\boldsymbol{I}}_{\boldsymbol{v_i}}$, where I_{v_i} is the instruction sequences contained in v_i, and $\vec{\boldsymbol{I}}_{\boldsymbol{v_i}}$ is the mapping vector. $\vec{\boldsymbol{I}}_{\boldsymbol{v_i}}=(\vec{\boldsymbol{i}}_{\boldsymbol{1}}, \ldots, \vec{\boldsymbol{i}}_{\boldsymbol{m}}), \vec{\boldsymbol{i}}_{\boldsymbol{j}} \in \boldsymbol{R}^{\boldsymbol{L}}, 1 \le j \le m$, L is the dimension of $\vec{\boldsymbol{I}}_{\boldsymbol{v_i}}$. In order to further make full use of the temporal information between instruction sequences, we further mainly use NLP methods to embed the basic block as a whole, and obtain the feature vector $\vec{\boldsymbol{B}}_{\boldsymbol{v_i}}$ associated with v_i. And then a CFG is transformed into an ACFG. But in this process, we find that some $\vec{\boldsymbol{B}}_{\boldsymbol{v_i}}$ are composed of all zero vectors. We call such $\vec{\boldsymbol{B}}_{\boldsymbol{v_i}}$

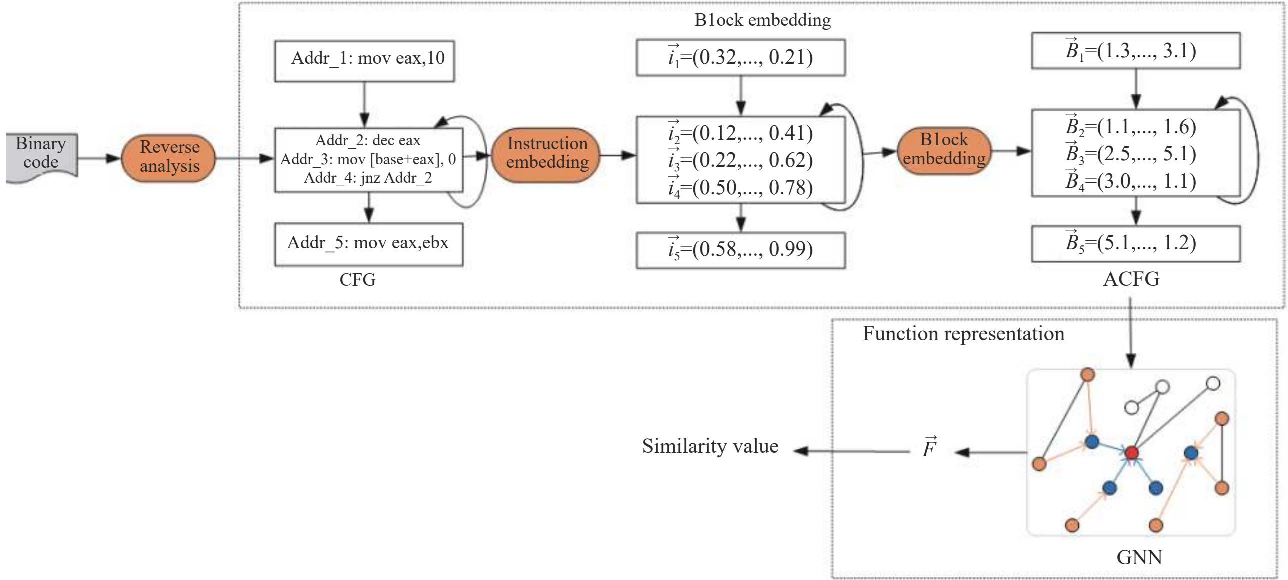


Figure 1 The system processing flow graph, input is a binary code, and output the similarity value.

invalid nodes marked as $\vec{B}_o$. We also analyze the specific reasons in Section III.4. Considering the invalid nodes will take up a lot of extra space in the following step, we design an algorithm (see Section III.4).

3) Function representation. This phase further considers the structural information in CFGs on the basis of basic block embedding. We use the inductive GNNs to accomplish it and indicate the result as $\vec{F}_{v_i} = (\vec{i}_1, \ldots, \vec{i}_m)$, $\vec{i}_j \in R^L, 1 \leq j \leq m$, L is the dimension of $\vec{F}_{v_i}$. $\vec{F}_{v_i}$ represents the final representation of each basic block v_i after instruction embedding, basic block embedding and function representation.

4) Similarity calculation and applications. In the last phase we can use the similarity between two $\vec{F}_{v_i}$ to represent the similarity between the corresponding binary basic blocks, and apply it in different downstream tasks. Such as vulnerability search, similar function detection and compiler provenance, etc.

3. Pre-processing

We exploit Radare2 to dissemble the binary code, counting the arithmetic instructions, constants, strings and jump etc. to obtain a sequence of instructions for each function. At the same time we also retain CFG and *Adj* of the corresponding function. Because of the complex structure of the compiled instruction set, we standardized it to facilitate the subsequent processing. Same as SAFE [27], we replace all basic memory addresses and all immediate addresses with MEM and IMM, respectively.

4. Basic block embedding

Instruction embedding The next step is instruction embedding. There are two common embedding methods: one-hot encoding and word2vec. The one-hot encoding can represent a relatively simple instruction for an input, but it can't capture the correlation of two similar instructions. In contrast, word2vec convert similar instructions into similar vectors. The core idea of word2vec is to portray a word in its context. There are two common models in word2vec: CBow and Skip-gram [41]. The Skip-gram model can achieve better performance on large data sets, so we finally choose Skip-gram to accomplish the instruction embedding.

Basic block embedding Basic block embedding based on the instruction embedding, with the goal of obtaining the associated multi-dimensional feature vector $\vec{B}_{v_i}$ for each block in CFG.

Tensorflow [42] (a deep learning framework) requires train batches of uniform dimension. Actually, the total number of basic blocks contained in each CFG(p) and the number of instruction sequences in each basic block(m) are both uncertain. Therefore, we firstly need to determine the value of p and m. Extensive experiments show that our framework can achieve best performance when $m = 50$ and $p = 100$ and this is done by two operations: padding or truncation.

We use LSTM, CodeBERT to embed the basic block. For comparison, we also do the without-embedding experiments. Note that CodeBERT is only for basic block embedding here, function representation and similarity calculation are still done by GNNs. This section takes LSTM as an example, and other models will be further detailed in Section IV.5. The core idea of LSTM is to control the transmission state by gating the state. Compared with RNN, LSTM can selectively forget or remember the input from the previous node, instead of just one memory superposition mode. So, LSTM can solve the problem vanishing gradient and exploding gradient in long- sequence training.

Each component of the padding vector is zero. That's why invalid nodes appear. Generally speaking, the invalid nodes will be generated when the actual total num-

ber of basic blocks formed by a function after disassembly is less than p. In order to avoid information omission such as semantics, we usually set higher values of p, so it is inevitable that invalid nodes will appear in most cases. However, in the following step, the invalid nodes have no practical significance (zero vector means no feature valid) and will cause additional system overhead, so we design an algorithm that can delete the invalid nodes while retaining the structure information.

The input to Algorithm 1 includes all basic blocks's embeddings $\vec{\boldsymbol{B}}$, Adj and the true label set T of G three parts. We are aiming at filtering out invalid nodes and keeping valid nodes and the structural information in G. The algorithm pseudo-code is shown in Algorithm 1.

Algorithm 1 Invalid node filtering

Input: $\vec{\boldsymbol{B}}$: all basic blocks's embeddings in G; Adj: Adjacency matrix; T: true label set.

Output: $\vec{\boldsymbol{B}}_{v_i}$.

1: global values A,D;$Invalid$, $Valid$, Res = $\emptyset$;
2: for each $\vec{\boldsymbol{B}}_{v_i}$ IN $\vec{\boldsymbol{B}}$
3: if $\vec{\boldsymbol{B}}_{v_i}$ equals(0)
4: $Invalid$.Append(A);
5: else
6: X = CONCAT(A, $\vec{\boldsymbol{B}}_{v_i}$, T);
7: $Valid$.Append(X);
8: end if
9: C.Increment;
10: end for
11: for i=0 to p IN Adj
12: $temp = \lfloor D/p \rfloor$;
13: for j=0 to p IN Adj
14: $Index1 = p * \lfloor p * temp \rfloor + i$;
15: $Index2 = p * \lfloor p * temp \rfloor + j$;
16: if $Index1$ &$Index2$ NOT IN $Invalid$
17: Res.Append($Index1$ &$Index2$);
18: end if
19: D.Increment;
20: end for
21: end for

5. Function representation

To take full advantage of the structural information in the CFGs, we utilize GNNs to embed the basic blocks in the ACFGs. Considering that GCN belongs to transductive learning whose training method is a full graph form and the loss of all nodes will only contribute to the gradient data once, which cannot do the small batch update usually used in DNN. This is very inefficient in terms of the number of gradient updates [31]. Therefore, in this section we will use GAE, a method based on inductive representation learning, for the function representation.

The core idea of GAE is to generate the embedding of the target node by learning a function that aggregates the representation of neighboring nodes. The embedding of all nodes $\vec{\boldsymbol{B}}$ and the $Adjs$ are regarded as the entire graph G and input to GAE. The output is the multi-dimensional feature vector $\vec{\boldsymbol{F}}_{v_i}$ associated with the basic block v_i. Current node v_i updates its state by aggregating the neighboring hidden states and its own state in the previous time step, which can be expressed as [37]

$$\boldsymbol{f}^{\boldsymbol{k}}_{\boldsymbol{\eta}(v_i)} \leftarrow \text{AGGREGATE}_k(\boldsymbol{f}^{\boldsymbol{k-1}}_{\boldsymbol{u}}, \forall_{\boldsymbol{u}} \in \boldsymbol{\eta}(\boldsymbol{v_i})) \quad (1)$$

$$\boldsymbol{f}^{\boldsymbol{k}}_{\boldsymbol{v_i}} \leftarrow \sigma(\boldsymbol{W}^{\boldsymbol{k}} \cdot \text{CONCAT}(\boldsymbol{f}^{\boldsymbol{k-1}}_{\boldsymbol{v_i}}, \boldsymbol{f}^{\boldsymbol{k}}_{\boldsymbol{\eta}(v_i)})) \quad (2)$$

First, current node v_i aggregates its immediate neighbourhood representations into a single vector $\boldsymbol{f}^{\boldsymbol{k-1}}_{\boldsymbol{N}(v_i)}$. K denotes the depth of search. η is defined as $\eta : v \rightarrow 2^v$, representing a neighborhood function. AGGREGATE$_k$ represents different aggregator functions.

Next, GAE concatenates the node's current representation $\boldsymbol{f}^{\boldsymbol{k-1}}_{\boldsymbol{v_i}}$ with the aggregated neighborhood vector $\boldsymbol{f}^{\boldsymbol{k-1}}_{\boldsymbol{N}(v_i)}$ through a fully connected layer with nonlinear transformation σ, as shown in (2). $\boldsymbol{W}^{\boldsymbol{k}}$ is the weight matrices. For notational convenience, we denote the final representations output at depth K as $\vec{\boldsymbol{F}}_{v_i} \equiv \boldsymbol{f}^{\boldsymbol{k}}_{\boldsymbol{v_i}}, \forall v_i \in V$.

This approach improves GCN in two aspects. On the one hand, through the strategy of sampling neighbors, the training way is changed from full batch to node-centered mini batch, and the embedding representation of the current node is only relevant to its K-order neighbor nodes. On the other hand, more aggregation functions (aggregator) are used to aggregate information about neighbor nodes. Aggregate functions can generate the embedding of a node directly. Through the improvement of the above two points, GAE greatly reduces the time to generate a new node embedding, and also achieves breakthrough progress when used in large-scale graph. L. Hamilton *et al.* [37] also show that GAE can achieve high performance with $K = 2$ and $K_1 \cdot K_2 \leq 500$, where K_1 and K_2 represent the sampling number of neighbor nodes when $K = 1$ and $K = 2$ respectively. In addition, we also evaluated GAT, which assigns weights to nodes to consider different important inputs.

6. Similarity calculation and applications

After getting the embeddings of binary blocks, we can apply them to many specific tasks, such as finding similar binary functions, compiler provenance, etc. For the first task, given a certain bug (binary function or basic block), we have to search for similar ones in a large dataset created using different compilers. In the compiler provenance task, we determine which compiler is a basic block compiled by.

IV. Evaluation

In this section, we evaluate our framework from the following aspects: i) How much improved the performance overhead compared to the baseline? ii) What are

the advantages of our framework over the other state-of-the-art works?

1. Experiment settings

1) Platform. We do our experiments on a Lenovo E5 server equipped with Intel Xeon E5 v4 core processor, 32G running memory, and 500G disk space. We use a plug-in of the tool Radare2 to analyze the binary code and extract CFG, *Adj* and an instruction sequence from each binary function. Our framework is implemented in TensorFlow.

2) Hyper-parameters. We only show the values of all hyper-parameters used in the final experiment. The maximum number of blocks in CFG is $100(p)$, and each block contains $50(m)$ instructions. We set the batch size to 160 and the hidden units as 25. We still use the parameter settings in GAE, set the K to 2, and K_1, K_2 are both 25. We also make a wide range of adjustments to these parameters, more details will be given in the next.

2. Dataset

We use the restricted compiler dataset from [26], which compiled different open-source projects: openssl-3.1.1, openssl-1.1.1, ccv-0.7, binutils2.30, curl-7.61.0, valgrind3.13.0, coreutils- 8.29, libhttpd-2.0. Each project has been compiled for AMD64 with three compilers: gcc-3.4 and gcc-5.0 and clang3.9 and all 4 optimization levels (O0, O1, O2, and O3). We randomly select some data form this to form our datasets A and B, where A contains 55863 basic blocks with 69107 edges and B contains 25188 basic blocks with 20565 edges. Both A and B are derived from the Restricted Compiler Dataset, but their scales and the binary function sets they belong to are different. Dataset A is taken from the first part of the original data set, and dataset B is taken from the second half.

3. Evaluation metric

We use the following common metrics to evaluate our framework:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{3}$$

$$\text{Precission} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5}$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6}$$

Accuracy refers to the percentage of correctly identified blocks, in which true positive (TP) indicates positive samples predicted by the model as positive one, false positive (FP) indicates negative samples predicted by the model to be positive one, false negative (FN) indicates positive samples predicted by the model to be negative and true negative (TN) indicates negative samples predicted by the model to be negative one. Precision and recall measures the percentage of matched blocks that are correctly labeled and the ability to identify matched block correctly. F1-score considers both the precision rate and the recall rate, so that both reach the highest level and achieve a balance.

In addition, we also use the total running time (TRT) and the average running time (ART) to measure the efficiency. The total running time is the sum of running time per round, and the average running time is defined as follows, where γ is the traning round number.

$$\text{ART} = \frac{\text{TRT}}{\gamma} \tag{7}$$

4. Evaluation on various implementation schemes

We implement two inductive function representation models: GAT and GAE. Combining with the models used in basic block embedding and function representation, we form six different implementation solutions in total. We use [26] as baseline and evaluate the above schemes from two aspects: accuracy and performance. The details are as follows.

Accuracy In order to ensure the fairness of the comparative experiments, we adjust the parameters of GAT and GAE to best, and used default parameters for [26]. We use Microsoft's CodeBERT-base pre-trained model. Table 1 shows the results on accuracy, precision, recall, and F1-score of different implementation schemes on dataset A and B. Result indicates that the scheme with LSTM for basic block embedding and GAE for function representation performs better than others and with comparable accuracy to the baseline.

Performance Figure 2 shows the first 20 rounds of seven approaches with data set A. Note that we only use part of the origin dataset, so we take the baseline as the standard and carry out an equal transformation of the data listed in the Figure 2 (for example the original dataset is double size of ours, and the values of running time except the baseline in Figure 2 are twice of the actual running time). Figure 2 shows that the time overhead of the LSTM+GAE is only slightly larger than without-embedding+GAE and far smaller than the baseline. While the inference time of a scheme is also important in the deployment. So we record the total runtime and runtime on the test set of each scheme. The results are shown in the Table 2 below. Dimension is determined by the basic block embedding scheme used.

For the data in Table 2, the following points need special attention. According to our statistics, the total running time of the baseline is 48667 s, which we scaled according to the size of the dataset. Relevant parameters of the GAT model in the table have been fine-tuned to a certain extent (the optimal parameters acceptable to the current system), because when we try to run the GAT-related model with the same parameters as GAE, the system will Kill the process because high overhead.

Table 1 The result of different implementation schemes

Scheme	Data set A				Data set B			
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
Without-embedding+GAT	57.7	56.1	56.9	56.5	56.4	56	74.6	64.0
Without-embedding+GAE	61.3	67.0	49.5	57.0	59.9	59.5	52.0	55.5
LSTM+GAT	65.1	64.4	63.0	63.7	62.1	61.0	71.4	65.8
CodeBERT+GAE	56.7	61.0	56.0	50.8	66.7	59.7	58.1	58.4
CodeBERT+GAT	55.1	56.8	55.7	55.1	62.3	57.3	51.8	62.3
LSTM+GAE	83.4	82.0	84.4	83.2	84.2	81.4	85.7	83.5
Baseline	87	87	87	87	87	87	87	87

So we default to the same basic block embedding method, GAT is less efficient than GAE. From the experimental results we can know that the input dimension is also positively related to the inference time when using the same GNN models. The efficiency of the without-embedding method is the highest, LSTM is slightly worse than the first one, and the CodeBERT is the lowest. The experimental results are basically consistent with our initial assumptions. Combing accuracy rate, we still think LSTM+GAE is the optimal solution for Binary code similarity detection.

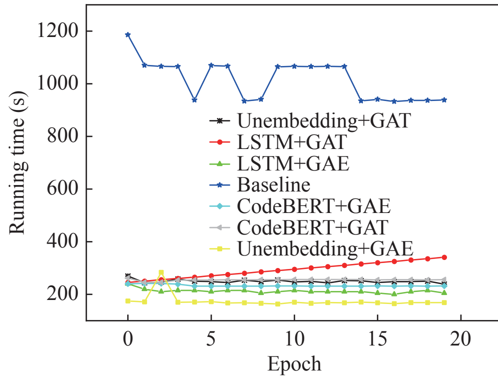


Figure 2 The running time of each scheme in first 20 epoch.

5. Evaluation on various basic block embedding models

The embedding of basic block is a key part of the overall detection, because the feature vectors of the basic blocks, as the input to GNNs, will directly affect the representing results and the detection accuracy. We evaluate three basic block embedding approaches (without-embedding, CodeBERT and LSTM) on GAT and GAE, recording their performance in terms of accuracy (Table 1) and efficiency (Table 2).

In the Without-embedding, we omit the steps of instruction embedding and basic block embedding, and only function representation of CFGs. This approach can't make full use of semantic information of the basic blocks, so the effect on both models is not ideal.

Before using CodeBERT for basic block embedding, we also tried the BERT [43], but the results were not satisfactory. CodeBert performs better than BERT in terms of accuracy and efficiency, but not as well as LSTM. We take full advantage of LSTM in long sequences and achieve good results. The scheme with LSTM for basic block embedding and GAE for function representation achieve the highest accuracy during the experiment. Surprisingly, the test time of this scheme is only slightly larger than Without-embedding + GAE and much smaller than others.

6. Evaluation on the number of neighbors sampled in GNNs

In this part, we explore the effects of the number of neighbors sampled in GAE. Define a array $[k_1, k_2]$ where K_1 is the number of neighbor samples when $K = 1$ and the same to K_2. Table 3 shows the effect of the number neighbors sampled on the test set when using data set B. With the increase of the number of neighbors sampled, the better accuracy is, and the ART is also greatly increased. Considering that more nodes will lead to higher

Table 2 The inference time and running time of different schemes in data set A&B

Scheme	Dimension	Data set A		Data set B	
		Running time (s)	Test time (s)	Running time (s)	Test time (s)
LSTM+GAE	100	3227	75.5	3525	79.5
CodeBERT+GAE	768	3894	100.5	2259	94.4
Without-embedding+GAE	50	3032	67.8	3060	69.2
LSTM+GAT	100	3866	56.8	3293	48.8
CodeBERT+GAT	768	4265	47.8	4302	47.1
Without-embedding+GAT	50	3271	45.2	3280	46.6
Baseline	50	7373	345.6	3743	345.6

Table 3 Evaluation on the number of neighbors sampled in GAE

Number of neighbor nodes	Accuracy	Precision	Recall	F1-score	ART (s)
[5,5]	80.0	83.4	75.9	79.5	11
[8,8]	82.1	79.5	85.6	82.3	14
[10,10]	81.7	82.2	81.8	82.0	18
[15,15]	81.1	78.8	84.1	81.3	24
[16,16]	80.5	80.2	79.8	80.0	25
[8,16]	80.6	84.8	75.5	80.0	14
[20,20]	82.9	85.7	78.2	81.8	32
[25,25]	84.2	81.4	85.7	83.5	44
[16,25]	82.3	86.3	77.6	81.7	26
[30,30]	83.8	84.0	82.6	83.3	54
[32,32]	82.4	83.9	81.0	82.4	54

consumption, and there is no further improvement in accuracy, we finally set K_1 and K_2 to 25.

7. Evaluation on the hidden units in GNNs

We find the number of hidden units in GNN affects the results a lot. So, we carry out experiments from 4 to 32 different hidden units on GAE.

Figure 3(a) and (b) show the results when set B is used. Generally speaking, as the increase of hidden units, the accuracy, recall and F1-score all increase, and ART does not significantly increase or decrease. Therefore, we think there is no obvious linear relationship between system performance and the number of hidden layers. Extensive experimental data shows that setting the number to 25 is a good choice.

V. Discussion

In our framework, we propose an approach that utilizes NLP combined with inductive GNNs to solve the problem of high overhead and the loss of structural information when graphs are applied to GCNs.

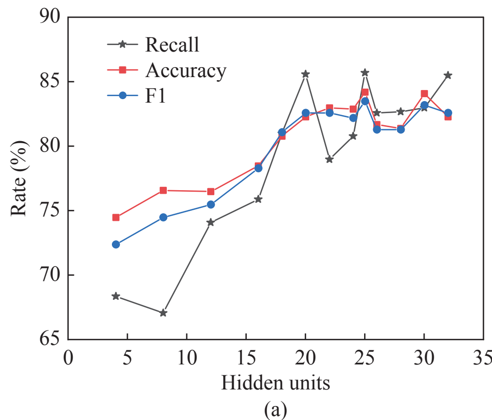
The block's feature vectors are essential, as they will act as input to function representation and participate in the similarity detection calculation. During the continuous experimentation with the basic block embedding approaches, we have tried to use BERT to complete the tasks of basic block representation and binary function similarity detection, but failed because the feature lexicon corresponding to BERT is not applicable to the instruction set. Therefore, we try again to use the bimodal pre-trained model CodeBERT to complete these experimental groups. CodeBERT performs very well in the classification task of source code level (we use CodeBERT to classify the source code of the data set, and the accuracy rate can be as high as 93%), but not as well as LSTM when applied to the basic block embedding. We think that for the data set in this paper, the sequential relationship between instructions is much more important. This is probably the most important reason why LSTM performs better.

The inference time of each scheme is closely related to the graph neural network model used. GAE can get the embedding of current node based on neighboring nodes, while GAT needs to calculate the importance of each node and assign weight accordingly, so it is far less efficient than GAE. We further extract more structural information than [26]. Interestingly, although reference [26] uses less structural information, its accuracy is comparable or better than our framework (87% for both datasets). So, we think the temporal information of the basic blocks is more important than the structural information. The following two facts support our point. Firstly, LSTM performs best among all the basic block embedding models. Secondly the two works by L. Massarelli *et al.* [26], [27] also show that there are some special features of the binary function that cannot be captured by representing it as a graph.

VI. Conclusion

Addressing the low performance and control flow semantic information loss problems in existing work is the starting point of this paper. We propose an inductive binary function representing framework that combines NLP with GNN. There are two mainly improvements.

Low performance, high overhead This part mainly



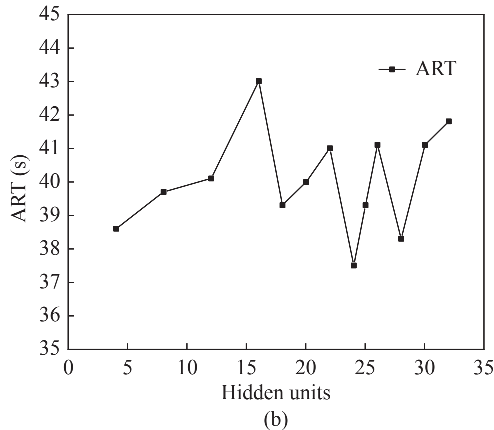


Figure 3 Result under the different hidden units in GAE.

includes the following two points. As mentioned above, we use the inductive representing learning to modify the training way from the full graph to node-centered mini batch, which fundamentally saves the cost. In addition, we filter the invalid nodes in the feature vector of the basic block, greatly reducing the unnecessary operation.

The loss of control flow structural information We use the *Adjs* to save the structural information as node pairs, which will serve as a partial input to the function representing module. Thanks to the powerful data understanding ability and cognitive ability of GNNs, the structural features of the graph data are integrated into the implementation process of the algorithm.

We investigate several models in the block embedding and function representation and form six implementation schemes finally. In order to avoid the influence of objective factors as much as possible, we reset the experimental environment for each experiment and make sure that no other processes running at the same time. We evaluated those implementation schemes on two data sets, and the experimental results show that the scheme with LSTM +GAE has the best comprehensive performance. We achieve similar detection results with only about 45% of the baseline overhead. In our writing, we also find the follow future works.

Different basic block embedding models We observe a large impact of different block embedding approaches on the test set result. In this paper, we use a variety of models, such as BERT and LSTM. [44] evaluates network embedding techniques' performances in software bug prediction. The influence of different basic block embedding models on GNN is also a direction worth exploring. As far as we know, no related works have been published yet.

Binary code similarity detection based on FCG Our framework, like existing static analysis solutions, is weak in anti-obfuscation. Due to the single reliance on CFGs, the semantic information that can be extracted is more limited if obfuscated. Function call graphs (FCGs) not only consider the instruction information, but also the more advanced internal features of the program, which can better deal with the anti-detection techniques commonly used in malware. Therefore, we can consider how to effectively combine the features in CFG and FCG to improve the anti-obfuscation capability while ensuring accuracy.

Enhance the generalization ability of the framework With the gradual popularization of open source and code reuse, many seemingly new versions of software only add a small amount of code to the original basis. So new nodes will appear on the corresponding CFG. How to enhance the generalization ability of the framework and identify these new nodes is also a direction worthy of in-depth study.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (Grant. No. 62172042) and the Major Scientific and Technological Innovation Projects of Shandong Province (Grant No. 2020CXGC 010116).

References

[1] Tableau, "Number of available applications in the google play store from December 2009 to March 2023", Available at: https://www.statista.com/statistics/266210/number, 2023-06-19

[2] X. Hu, S. Bhatkar, K. Griffin, *et al*., "MutantX-S: scalable malware clustering based on static features," in *Proceedings of 2013 USENIX conference on Annual Technical Conference*, San Jose, CA, USA, pp. 187–198, 2013.

[3] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys*, vol. 54, no. 3, article no. 51, 2021.

[4] Z. M. Tai, H. Washizaki, Y. Fukazawa, *et al*., "Binary similarity analysis for vulnerability detection," in *Proceedings of the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, Madrid, Spain, pp. 1121–1122, 2020.

[5] Y. David and E. Yahav, "Tracelet-based code search in executables," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 349–360, 2014.

[6] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.

[7] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona, Spain, pp. 79–94, 2017.

[8] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Williamsburg, VA, USA, pp. 392–404, 2018.

[9] P. Shirani, L. Collard, B. L. Agba, *et al*., "BINARM: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Saclay, France, pp. 114–138, 2018.

[10] J. Jang, S. Choi, and J. Hong, "A method for resilient graph-based comparison of executable objects," in *Proceedings of 2012 ACM Research in Applied Computation Symposium*, San Antonio, TX, USA, pp. 288–289, 2012.

[11] H. Flake, "Structural comparison of executable objects, " in *Proceedings of Detection of intrusions and malware & vulnerability assessment*, Dortmund, Germany, pp. 161–173, 2004.

[12] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 10th International Conference on Information and Communications Security*, Birmingham, UK, pp. 238–255, 2008.

[13] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL, USA, pp. 342–352, 2017.

[14] Z. Z. Xu, B. H. Chen, M. Chandramohan, *et al*., "SPAIN: Security patch analysis for binaries towards understanding the pain and pills," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*,

Buenos Aires, Argentina, pp. 462–472, 2017.

[15] T. Kim, Y. R. Lee, B. Kang, *et al.*, "Binary executable file similarity calculation using function matching," *The Journal of Supercomputing*, vol. 75, no. 2, pp. 607–622, 2019.

[16] D. Katsaros, "Structural pattern recognition with graph edit distance: approximation algorithms and applications," *Computing Reviews*, vol. 57, no. 11, pp. 665–665, 2016.

[17] S. Alrabaee, L. Y. Wang, and M. Debbabi, "*BinGold*: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGS)," *Digital Investigation*, vol. 18, no. S, pp. S11–S22, 2016.

[18] M. Chandramohan, Y. X. Xue, Z. Z. Xu, *et al.*, "BinGo: Cross-architecture cross-OS binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, WA, USA, pp. 678–689, 2016.

[19] S. Eschweiler, K. Yakdan, and Gerhards-Padilla E, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, pp. 58–79, 2016.

[20] S. Alrabaee, P. Shirani, L. Y. Wang, *et al.*, "*SIGMA*: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, no. S1, pp. S61–S71, 2015.

[21] F. Qian, R. D. Zhou, C. C. Xu, *et al.*, "Scalable graph-based bug search for firmware images," in *Proceedings of 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, pp. 480–491, 2016.

[22] X. J. Xu, C. Liu, Q. Feng, *et al.*, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, USA, pp. 363–376, 2017.

[23] H. J. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, New York City, NY, USA, pp. 2702–2711, 2016.

[24] J. Gao, X. Yang, Y. Fu, *et al.*, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, pp. 896–899, 2018.

[25] J. Gao, X. Yang, Y. Fu, *et al.*, "VulSeeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA, pp. 803–808, 2018.

[26] L. Massarelli, G. A. Di Luna, F. Petroni, *et al.*, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of Workshop on Binary Analysis Research*, San Diego, CA, USA, pp. 1–11, 2019.

[27] L. Massarelli, G. A. Di Luna, F. Petroni, *et al.*, "SAFE: Self-attentive function embeddings for binary similarity," in *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Gothenburg, Sweden, pp. 309–329, 2019.

[28] 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 *Proceedings of 2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, pp. 472–489, 2019.

[29] B. C. Liu, W. Huo, C. Zhang, *et al.*, "αDiff: cross-version binary code similarity detection with DNN," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, France, pp. 667–678, 2018.

[30] Z. P. Yu, R. Cao, Q. Y. Tang, *et al.*, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, New York, NY, USA, pp. 1145–1152, 2020.

[31] C. Y. Zhuang and Q. Ma, "Dual graph convolutional networks for graph-based semi-supervised classification," in *Proceedings of 2018 World Wide Web Conference*, Lyon, France, pp. 499–508, 2018.

[32] X. Hu, T. C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, pp. 611–620, 2009.

[33] N. E. Rosenblum, B. P. Miller, and X. J. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, Toronto, Canada, pp. 21–28, 2010.

[34] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[35] Z. Y. Feng, D. Y. Guo, and D. Y. Tang, "CodeBERT: A pre-trained model for programming and natural languages," *in Proceedings ofFindings of the Association for Computational Linguistics: EMNLP 2020*, Online, pp. 1536–1547, 2020.

[36] P. Veličković, G. Cucurull, A. Casanova, *et al.*, "Graph attention networks," in *Proceedings of the 6th International Conference on Learning Representations*, ICLR 2018 OpenReview.net, 2018.

[37] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, CA, USA, pp. 1025–1035, 2017.

[38] S. G. Yang, L. Cheng, Y. C. Zeng, *et al.*, "Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection," in *Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Taipei, China, pp. 224–236, 2021.

[39] Radareorg, "Radare2," Available at: https://github.com/radareorg/radare2, 2021-12.

[40] P. B. M. Abadi and E. A. J. Chen, "Word2vec skip-gram implementation in tensor-flow," Available at: https://www.tensorflow:tutorials/representation/word2vec, 2021-12.

[41] T. Mikolov, K. Chen, G. Corrado, *et al.*, "Efficient estimation of word representations in vector space," in *Proceedings of the 1st International Conference on Learning Representations*, Scottsdale, AZ, USA, pp. 1–12, 2013.

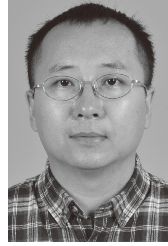
[42] M. Abadi, P. Barham, J. M. Chen, *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Savannah, GA, USA, pp. 265–283, 2016.

[43] J. Devlin, M. W. Chang, K. Lee, *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Minneapolis, MN, USA, pp. 4171–4186, 2018.

[44] Y. Qu and H. Yin, "Evaluating network embedding techniques' performances in software bug prediction," *Empirical Software Engineering*, vol. 26, no. 4, article no. 60, 2021.

Jinxue PENG was born in 1999. She is a postgraduate of Beijing Institute of Technology, China. Her main research interests focus on binary code similarity detection and machine learning. (Email: 3120201124@bit.edu.cn)

Yong WANG was born in 1975. She is an Associate Professor of Beijing Institute of Technology, China. Her main research interests focus on cyber security and machine leaning.
(Email: wangyong@bit.edu.cn)



Jingfeng XUE was born in 1975. He is a Professor and Ph.D. Supervisor in Beijing Institute of Technology, China. His main research interests focus on network security and software security. (Email: xuejf@bit.edu.cn)

Zhenyan LIU was born in 1975. She is an Associate Professor of Beijing Institute of Technology, China. Her main research interests focus on machine learning.
(Email: zhenyanliu@bit.edu.cn)