IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON ARTIFICIAL INTELLIGENCE IN CYBERSECURITY

Received September 30, 2019, accepted November 7, 2019, date of current version December 6, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2953173

BiN: A Two-Level Learning-Based Bug Search for Cross-Architecture Binary

HAO WU, HUI SHU, FEI KANG, AND XIAOBING XIONG
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
Corresponding author: Hao Wu (revinc3up@gmail.com)

This work was supported in part by the National Key Research and Development Project 2016YFB08011601.

ABSTRACT With the popularity of IoT (Internet of Things) devices, the security risks of these devices are increasing. However, due to the multisource heterogeneity of IoT devices, there are significant differences between the vulnerability detection of the Internet of Things and the PC-based vulnerability search method. Therefore, determining how to accurate search for vulnerabilities in large-scale cross-platform binary executable files is an urgent problem to be solved. At present, the solution to this problem mostly calculates code similarities by generating a CFG (control flow graph) from binary code, but due to the choice of architecture, OS (operating system) or compilation options, the same source code will be compiled into different assembly codes. The performance of existing vulnerability search methods for cross-architecture binaries has been challenged. To alleviate the vast differences in the assembly codes caused by different compilation scenarios, this paper proposes a cross-platform large-scale binary vulnerability search method based on two-level feature semantic learning. The contribution is that we have defined a new functional structured signature method to mitigate the massive grammatical and structural differences of binary files caused by different compilation environments. Moreover, we reasonably integrate the hierarchical model of Structure2Vec and GAT (graph attention network) and implement training from the internal control flow characteristics of the function and the call relationship between functions to obtain a more accurate functional semantic expression.

INDEX TERMS IoT, cross-platform, binary vulnerability, structured signature, deep learning.

I. INTRODUCTION

Using open source code or using third-party libraries is a common approach in the development process, and the same vendor often reuses code, which also provides fertile ground for the generation and survival of vulnerabilities. If an organization does not fully understand all of the code it uses or there are bugs in the code, it will not be able to withstand common attacks against known vulnerabilities in these components, and it will also be exposed to risk [25], [30]. It is foreseeable that the same vulnerability function with different architectures may appear in a large number of IoT devices. To address this critical issue, researchers are devoting their efforts to developing automated analysis technologies to meet the needs of IoT product security testing [1]–[3], [26], [27]. In response to a wide variety of IoT devices, the ability to perform vulnerability searches in an efficient and accurate manner is becoming increasingly important. This vulnerability search technology will enable security practitioners to find problems with high efficiency, saving time and resources.

The associate editor coordinating the review of this manuscript and approving it for publication was Chi-Yuan Chen.

A. PROBLEM STATEMENT

To solve the general cross-platform vulnerability search, recent research can be divided into two types. The static method attempts to extract various robust features that are architecture-independent from the binary CFG (control flow graph) and uses the graph matching algorithm to find the same vulnerability function in the binary file through the feature representation function. However, for the same function in different platforms and different compilation configurations, such as the optimization level, the CFG of the compiled binary function is significantly different [28], which greatly affects the accuracy of the function search. The dynamic method can produce accurate matching results by monitoring their execution, extracting semantic signatures, and comparing code similarities [29], [32]–[34]. The limitation is that it is very time-consuming to cope with large-scale vulnerability search work in the real world.

Fortunately, a learning-based approach for solving cross-platform binary similarity search problems has been proposed in recent research. The main idea of these studies is to learn advanced semantic features from assembly instructions in binary files. Among them, Genius and Gemini are two methods that work well.

Genius [2] learns advanced feature representations from control flow graphs and encodes (i.e., embeds) the graph into embeddings (i.e., high dimensional numerical vectors). Genius uses the graph matching algorithm to cluster similar functions to generate the codebook by extracting the robust features of the CFG in the different compilation environments of the cross-architecture, and generate the function embedding according to the codebook. Then, the firmware database and the vulnerability function database are built, and LSH (locality-sensitive hashing) is used for the large-scale vulnerability search. In our experiments, Genius's embedding-generation is not efficient. In addition, the search accuracy of Genius is not sufficient to meet the large-scale vulnerability search work in millions of firmware.

In Gemini [4], a method was proposed based on a deep neural network to generate the embedding of binary functions for similarity detection, which improved the accuracy and efficiency to some extent. Gemini extracts the robust features of the function across the architecture and feeds the extracted basic block-level features of the function and the representation of the CFG structure to a DNN model. Through several layers of Structure2Vec [15] iteration, the basic block node features are propagated to other nodes related to it, and the representations of all basic block nodes are aggregated to generate a high-dimensional vector representation of the function. However, Gemini did not overcome the limitations of Genius's graph-based matching method. In our experiments, to accurately find the vulnerability function, we need to analyze at least the top-86 candidate functions in the results. Efficient vulnerability search still requires tremendous manual analysis.

In addition, none of the above methods fully consider the impact of different compilation options on the CFG structure function. In our experiments, program syntax changes may cause the learning model to fail due to different complex compilation settings, so it is necessary to find features that can more accurately describe the semantics of the function in complex compilation settings.

B. OUR APPROACH

In this article, we focus on how to select, extract, and take advantage of the characteristics of binary functions to address these challenges. To alleviate the vast difference of function binary files in different compilation scenarios, we introduce the concept of the structured signature of functions. The binary code of the objective function is graphically described, and the signature information is extracted from the figure to facilitate comparison. We describe the binary function through three types of graphs and divide it into the following two levels:

1) INTRA-FUNCTION LEVEL

The control flow graph and data flow graph describe the control flow and the data flow of a function. All instructions of a function can be divided into several basic blocks. However, the nodes of the CFG and the DFG are composed of the basic blocks in different division. Therefore, we attach the data flow transfer information to the structure of CFG and mark 0 and 1 to indicate whether there is a data transfer between the two base blocks. The edges between the CFG nodes represent the control flow direction; The labels on the CFG edge indicates the data transfer between the two base blocks.

In the existing research, the functions control flow and data flow have a certain robustness in different architectures, different OSs and different compilation optimization levels [38]. We try to eliminate the effects by extracting the control flow and data flow as the semantic features of the function, that is, extracting the features that are versatile across platforms and different compilation options, independent of the architecture and compilation settings. Unlike methods like Gemini, which directly use the feature sets selected by discovRE for the graph matching algorithm, we designed a model-oriented GA(genetic algorithm) to select a suitable feature set.

2) INTER-FUNCTION LEVEL

The function call describes the calling relationship between the functions to be analyzed. The function call graph involves the function call relationship of the entire binary file. Each node represents a function, and each function node can be represented by function CFG with data flow information.

Even in different compilation environments, the call relationships between functions are very robust as relatively general features [39]. There is no difference in the functions that the same function calls in different environments. Therefore, we use the calling relationship between functions combined with the idea of crowd classification in social networks, and consider the impact of calling functions on function recognition, generating more accurate feature representations of functions.

We describe binary functions using the three graphs above and generate signature information for them based on the graph. In our method, we extract the basic block-level features that are robust under the cross-architecture, different optimization options in the CFG with data flow information and the information of the calling function in the function call graph as the signature factor of the function.

Our model uses two neural networks to learn the two-level features of functions. For the intra-function level learning model, our model is based on Gemini, and we add the data flow information to capture more semantic information of functions. Since the number of function calls is variable and the relationship between function calls cannot be well expressed on Gemini models, it is more difficult to deal with function calling relationships. Therefore, we need to introduce new mechanisms or models to learn the semantic

information of function calls. Finally, inspired by social network, we introduced GAT(graph attention graph) to try to solve the above problems.

The DNN (deep neural network) is used to train the learning function basic block-level features, and the GAT is used to train the influence of the calling relationship on the function to generate a high-dimensional feature vector containing more precise semantics. Finally, the similarity between functions is measured by calculating the distance of the function eigenvector to identify the vulnerability function.

C. RESULTS OVERVIEW

We propose a vulnerability search method based on hierarchical semantic learning [44] and implement a prototype for verification experiments. It performs two-level analysis of the binary file to extract the more precise features of the binary function, and uses deep learning [7] methods to learn the control flow and data flow characteristics of the function, and combines the function call relationship to generate high-dimensional function representation to calculate the similarity of functions.

In this prototype, we use the experience of manual vulnerability search [38], [39], adding data flow and function call information based on the characteristics of existing research. Accordingly, the limitations of existing architectures are overcome by resolving the effects on binary files compiled in the different architectures, different compilers, compilation settings and operating system environments. Then, several sets of experiments are performed to evaluate the accuracy and efficiency of our prototype, proving that the accuracy obtained in the case of a small increase in time is significantly better than that of the existing work.

D. CONTRIBUTIONS

In summary, our main contributions are the following:

1) Guided by manual vulnerability search, we propose a solution to reduce the impact of different compilation environments on function binaries;
2) We attach the function data flow to CFG and designed a model-oriented GA to select suitable features to obtain more complete semantics;
3) We apply a artificial neural network GAT to construct a network architecture based on the attention mechanism of neighbor nodes, and consider the call relationships between functions and generate richer semantic representations;
4) We establish a hierarchical model to fused the GAT [5] model and the Structure2Vec [6] model, and train them together from the intra-function characteristic and the call relationship between functions to achieve a more accurate functional similarity comparison;
5) We implemented a prototype called BiN. Our evaluation shows that BiN can achieve higher AUC than other state-of-the-art graphics-based matching methods in the test set built by OpenSSL and BusyBox;
6) We tested our prototypes on a larger data set, and the results showed that our method implementation was accurate and efficient enough to handle real-world vulnerability detection efforts.

II. BACKGROUND

Most of the existing graph-based function similarity calculation methods extract features directly from the function CFG, PDG(program dependency graph) [8], AST(abstract syntax tree) [9], etc., but in fact, by using different choices of architecture, OS or compilation options, the same source code may be compiled into assembly code with different structures, and the function features extracted by these methods cannot accurately express the function semantics [35]. In what follows, we describe examples of such disturbances.

A. LOOP OPTIMIZATION

Loops are a very important program structure, and the compiler uses some loop optimization techniques to reduce the computation time taken by the loop structure [40]. We take loop unrolling as an example to analyze the impact of the loop optimization technique function control flow graph. Loop unrolling is a loop optimization technique that attempts to optimize the execution speed of a program at the expense of space, rewrite the loop into a repeating sequence of similar independent statements, and reduce the number of iterations of the loop, thereby eliminating this overhead. However, after the loop is expanded, the assembly instruction of the function will be changed. The accuracy of the similarity of the function can be reduced. The accuracy of the method of comparing function similarity only by the CFG characteristic is reduced.

B. FUNCTION INLINE

To optimize the running speed, the compiler inlines small functions into the code of the calling function to reduce the performance cost required for the function to jump back and forth during the execution of the function call [41]. Thus, the control flow structure of the function is significantly changed. If the extracted feature is derived from a function containing an inline function, or if the target program does not recognize such an inline function, it will directly affect the function similarity based on the graph matching. The accuracy of the sexual calculations. Moreover, finding inline code for functions is a challenging task, and current methods are not able to meet the timeliness requirements of our goals [10].

C. CODE ELIMINATION

Common subexpression elimination is a classic compiler optimization technique that is mainly used to save computing resources and avoid redundant calculations. Therefore, the compiler deletes the time that has been calculated and has not changed since the calculation as a common subexpression [11]. In addition to this, code that never executes or code that does not make sense will be deleted.

Therefore, the directly extracted information is not sufficient for accurate binary code search. To satisfy the analysis

of different compilation environments, it is necessary to have some flexibility in the feature differences in the binary file. We need to find more characteristic features to reduce the impact, so we analyze the impact of the compilation environment on the binary file and find that the processing of the function data stream in different compilation environments is not significantly different. In addition, the current function similarity calculation methods are mostly concentrated inside a single function, ignoring the calling relationship between functions.

III. EXTRACTING FEATURES OF A BINARY FUNCTION

In this section, we will describe how we generate structured signatures for functions and how we perform the task of extracting the features of a binary function.

A. INTRA-FUNCTION FEATURE

The existing semantic learning methods rely on the CFG of the function to extract features for each basic block of the function and perform similarity comparisons based on these features. BiN reproduces the Genius extraction feature by first splitting the binary into the corresponding assembler using the IDA Pro [12] tool. It then creates a CFG for each assembly function using IDA Python provided by IDA Pro. Simultaneously, we also use IDA Pro's plugin named MIASM [13] to determine whether there is data transfer between two basic blocks.

Given the impact of the compile environment on the assembly code that we have discussed in the *Section Background*, we have made some changes to the features extracted in Gemini. Unlike Genius and Gemini directly use the function features selected by DiscovRE for graph-matching algorithm, and we designed the model-oriented genetic algorithm and selected the function features that are more suitable for our model.

In our implementation, we use genetic algorithms to select the best subset of features. We extracted 50 features of functions as in Wang's [36] work and selected the best performing 9 features; our experimental code was released on GitHub[1] [37]. The model-oriented genetic algorithm for selecting features of binary function is summarized in Algorithm 1.

In our model, population [42] is the selected subset of function features, and the generation refers to the round number. For each population in our model, we first initialize the mating set and the offspring, and feed function pairs with ground truth labels into the BiN model and get the fitness of this population. Then we rank them by population's fitness, after that we select population using stochastic sampling with replacement, crossover and mutating. Finally, we update the population. After T generations, we select the best performing subset of features as final selection. The features shown in Table 1 are the initial features of each basic block we used, including 8 statistical features and 1 structural feature. The initial features of each basic block of the function

[1]https://github.com/V1ncent7/GA_feature_select

Algorithm 1 Genetic Algorithm for Feature Selection

Input: function pair set $P = p_i, i \in 1, 2, \ldots, 8000$
ground truth label $L = l_i, i \in 0, 1$
extracted features set $F = f_1, f_2, \ldots, f_{50}$
Number_of _generations T
Number_of _population n
Output: Selected Features set *FinalSet*
Initialize mating set $\Pi = \pi_1, \pi_2, \ldots, \pi_n$,
$\pi = \alpha_1, \alpha_2, \ldots, \alpha_{50}, \alpha_i \in 0, 1$
Initialize offspring O
for $i = 1 \rightarrow T$ **do**
  Randomly generate probability C_1
  Randomly generate probability C_2
  for $j = 1 \rightarrow n$ **do**
    Input P, L, F, p_j into BiN Model
    Output AUC_j = AUC of BiN
  end for
  Rank $AUC\_k, k \in k, \ldots, 20$
  $Fitness_k = \gamma \times R(k), k \in 1, \ldots, 20$
  Add $n/2$ copies π_i to Π randomly according to $Fitness_i$
  Select a pair π_k and π_p from Π
  $O = O \cup crossover(\pi_k, \pi_p)$ with C_1
  Switch the α_k bit in $\alpha_p \in$ O with C_2
  Update the population $\Pi = Combine(\Pi, O)$
end for
return *FinalSet* = Max Fitness π

(9-dimensional vector) are input to the model to generate a semantic embedded vector of the function.

In addition, Bingold [38] considered the data flow as a reliable representation and believed that the data flow among basic blocks can capture data and variable dependencies, and they also believed that the data flow analysis provides the ability to locate the code that has been inlined. Bingold divided the basic blocks according to the data flow structure. Different from Bingold, we attach the information of the data flow to the basic block of the structure of CFG.

We label 0 and 1 on the edge of CFG to represent whether there is a data transfer between two basic blocks to enrich the function semantic. In addition, we determine the data transfer between two basic blocks by checking whether the instructions in basic blocks have access to the same address register. Since it considers the control structure and data flow information within a function, which can effectively alleviate the structural changes of CFG caused by different compilation environments. In figure 2(a), the dotted square illustrates an example of the CFG with data flow information.

However, this method does not capture the interaction between these functions when extracting function features from binary files and will lose considerable structural information. To obtain the structural information, a new function structured signature is introduced, and the call relationship between functions and the internal control flow information of the function are used together as the characteristics of the description function.

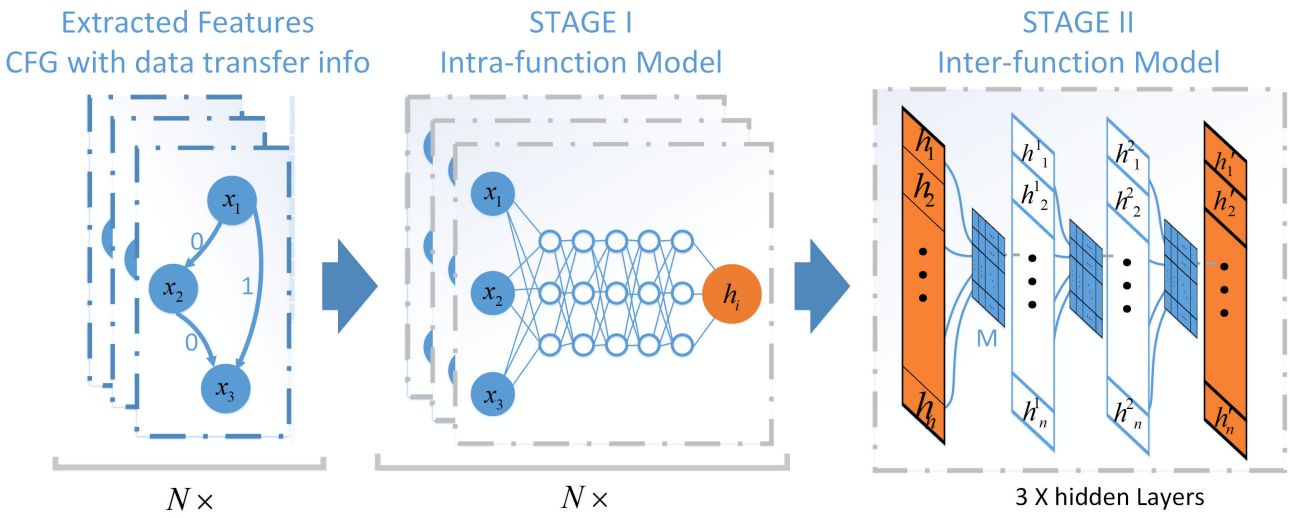


FIGURE 1. Overall model.

TABLE 1. Robust Intra-function features used by BiN.

Type	Feature Name	Example
Statistical features	No. of logical instructions	and, xor
	No. of arithmetic instructions	add, mul
	No. of library function calls	call strcpy
	No. of conditional jump instructions	jne, jb
	No. of unconditional jump instructions	jmp
	No. of comparative instructions	test
	No. of Numeric Constants	45533
	No. of string constants	passwd
Structure feature	No. of offspring	

B. INTER-FUNCTION FEATURE

The FCG (function call graph) [43] is a directed graph representation in which the vertices of the graph represent functions and the directed edges represent the calling relationships of functions. When extracting the FCG, for the dynamically loaded third-party function library, the function name is obtained as a label by reading the import address table of the executable file; for the local function, we use the starting address of the local function as the function label. We represent the caller-callee relationship between functions as a directed, unweighted edge. Finally, according to the FCG, the adjacency matrix of the function call relationship is constructed. In our model, we only consider the internal functions of the file and the statically linked third-party library functions, and the function call graph constitutes the adjacency list to represent the function call relationship of the binary file.

We extract the function features by the above method and use the extracted features as the structured signature of the function.

IV. SEMANTIC LEARNING PREDICTOR

In this section, we present our solution to the problem of code similarity based on hierarchical feature learning. We first present the overview of our solution in Section VI.A, and then, the two-level graph embedding network [15] that learns the intra-function characteristics and function call relationships is described in detail (Section IV.B & IV.C). In Section IV.D, the relationship between the two-level network in the model and the similarity of the training function is explained.

A. SOLUTION OVERVIEW

As discussed in Gemini, the measure of a code similarity can be task dependent. We try to find a mapping *phi* that converts a binary function into a representation of a high-dimensional eigenvector. Given two binary functions f_1 and f_2, the similarity of the two functions is judged by a given similarity calculation function $Sim(\cdot,\cdot)$, that is, if the two functions are similar, then $Sim(\phi(f_1), \phi(f_2))$ takes a value of 1. If the two functions are not similar, $Sim(\phi(f_1), \phi(f_2))$ takes a value of -1.

In our model, we try to learn how to generate function embedding by means of deep learning, that is, mapping ϕ. Different from Gemini, we fully consider the two-level characteristics of the function, the characteristics of the intra-function and the calling relationship between the functions, establishing a learning model for the two-level feature and integrating the training model of the two-level feature reasonably and effectively. Vectors are generated that express more semantics of the function to further improve accuracy.

B. OVERALL MODEL AND TRAINING METHOD

In this section, we will introduce how our model generates the embedding and how the model trains the similarity of functions through the learning of two-level features. The process of our model is illustrated in Figure 1. In the following sections, we will describe the implementation of the two-level model in detail separately.

In Figure 1, the inputs are the extracted CFG with data flow information extracted from the firmware binary code, and N is the number of functions in the binary code. We feed them into the *Intra-function Feature Learning Model* and obtain the intermediate embeddings; in this paper, we have 5 iterations in the *Intra-function Feature Learning Model* and each iterations has a 2 fully connected neural networks to train the features. Then, we feed the intermediate embeddings into the *Inter-function Feature Learning Model*, as indicated in Figure 1, and there are a 3 hidden layers in our model. We use the adjacency matrix to determine the dependencies of functions and, through the attention mechanism, calculate the influence of adjacent nodes on the function nodes and generate the final representation that is graphically unrelated and contains the function call information.

We illustrate the performance of the model in comparing the similarities of inline functions. We assume that a function containing subroutines B is compiled into A and A' under different compilation environments, where A inlines function B, and A' doesn't inline function B, but calling function B. In our model, function A generates a vector representation containing function B's semantics in Stage I; function A' will include the semantics of function B through Stage II.

As mentioned in the first section of this chapter, we obtain a mapping ϕ that converts a binary function into a high-dimensional representation through deep learning. Next, we need to find a way to describe the similarity between functions by the high-dimensional representation of the function and then train our model. In data preprocessing, the function is divided into L groups of function pairs. If the pair has two of the same functions compiled by the same source in different compilation environments, i.e., $sim(\phi(f_i), \phi(f_i'))$, the function assigns a ground truth value of 1 to the label $label_{f_{ii'}}$; otherwise, if there are two different functions, i.e., $sim(\phi(f_i), \phi(f_j))$, the function assigns a ground truth value of -1 to the label $label_{f_{ij}}$. Further, we describe the similarity of functions by the cosine distance as

$$\mathrm{sim}\left(f_i, f_j\right) = \cos\left(\phi(f_i), \phi(f_j)\right) = \frac{\left\langle \phi(f_i), \phi\left(f_j\right)\right\rangle}{\|\phi(f_i)\| \cdot \left\|\phi\left(f_j\right)\right\|} \tag{1}$$

In the training phase, the model evaluates the quality of the mapping by comparing the difference between the generated similarity and the ground truth value of the function pair. We use mean square error (MSE) as a measure, the formula is as follows:

$$MSE = \frac{1}{L}\sum_{i=1}^{L}\left(\mathrm{sim}\left(f_i, f_j\right) - label_{f_{ij}}\right)^2 \tag{2}$$

Then, we train the shared parameters $\mathbf{W_1}, \mathbf{W_2}, \mathbf{P_1}, \cdots, \mathbf{P_n}$ in the intra-function model and $\mathbf{W}, \vec{a}$ in inter-function model to minimize *MSE* in Equation 1. In addition, we improve the generalization ability of the model by adding the *L2 regularization* and *Drop out* to prevent overfitting. We optimized *MSE* with the stochastic gradient descent algorithm. Ultimately, once the optimal value of shared parameters are learned, we can easily convert the functions to a high-dimensional representation for function similarity comparison.

C. INTRA-FUNCTION FEATURE LEARNING MODEL

The intra-function model is improved in Gemini's Structure2vec model, and the main purpose is to obtain high-dimensional vectors that represent functions, such as control flow and data flow, that is, to generate an embedding. Structure2Vec was inspired by the graph model inference algorithm. The features of its vertices are recursively nonlinearly aggregated according to the graph topology. After performing enough iterations, each vertex will contain information about neighbor vertices. After extracting the basic block-level 8-dimensional feature representation of the function in the target binary, the features are input to the learning model to generate the semantic embedding for similarity calculation.

Figure 2(a) is a CFG with data transfer representation denoted as $g = (\vartheta, \xi, \tilde{x_i})$, containing 3 vertices in the graph with block-level features $\tilde{x}$, where ϑ and ξ are the sets of vertices and edges, respectively. After T layer iterations, the DNN model will generate a p-dimensional embedding for each $v \in \vartheta$, and each iteration will generate p-dimensional vertex-specific features μ_i containing information about their neighborhood and the data-transferred vertices' features determined by the CFG with the data transfer representation [14]. After generating the embedding of each vertex μ^T, the embedding vector μ of g will be computed as an aggregation with the formula $W_2(\sum_{v\in\vartheta}\mu_v^T)$.

The method of updating the embedding at each iteration is visualized in Figure 2(b). We now discuss the method of updating the embedding shown as the following form:

$$\mu_v^t = tanh(W_1 x_v + \sigma_c(\sum_{i\in N(v)}\mu_i^{(t-1)}) + \sigma_d(\sum_{j\in D(v)}\mu_j^{(t-1)})) \tag{3}$$

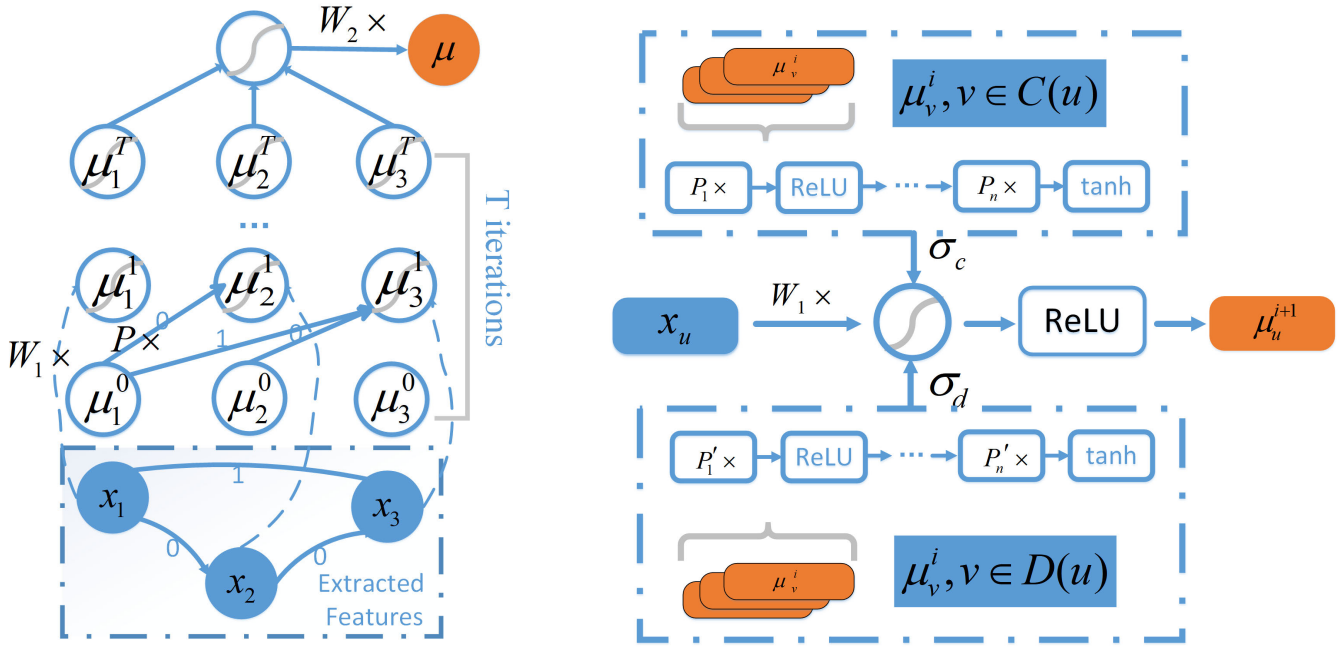
where x_v is a d-dimensional numerical vector for each vertex, W_1 is a $d \times p$ matrix. We separately denote $N(\mu)$ as the set of neighbors of vertex μ and $D(\mu)$ as the set of vertices that has data transferred with vertex μ in graph g. In addition, σ_c and σ_d are two nonlinear transformation $\sigma(\cdot)$, which defines n-layer fully connected networks to achieve the process of gathering other vertices' features as

$$\begin{aligned}\sigma_c(l_v) &= P_{n_c} \times \mathrm{ReLU}\left(P_2 \times \ldots \mathrm{ReLU}\left(P_1 l_v\right)\right)\\ \sigma_d(l_v') &= P_{n_d}' \times \mathrm{ReLU}\left(P_2' \times \ldots \mathrm{ReLU}\left(P_1' l_v'\right)\right)\end{aligned} \tag{4}$$

where $P_i (i = 1, \ldots, n)$ and $P_i' (i = 1, \ldots, n)$ are $p \times p$ matrix, and n is the *embedding depth*. ReLU is the activation function. The overall algorithm for generating the intra-function-level embedding is summarized in Algorithm 2.

After T iterations, the feature of each vertex is propagated to other vertices associated with it, and each vertex embedding contains the semantics of the context.

For intra-function feature learning model, we have added data flow information to capture more semantic information



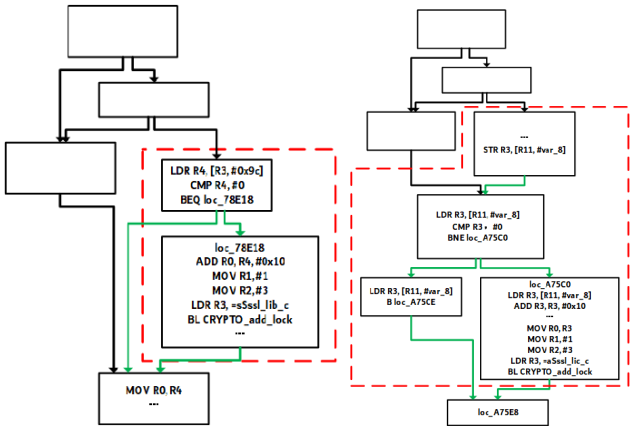
(a) DNN model for generating function embedding (b) Layer of DNN model

FIGURE 2. The DNN model of BiN.

Algorithm 2 Intra-Function Embedding Algorithm

Input: Control Flow Graph with data transfer $g = (\vartheta, \xi, \tilde{x_i})$
Output: Function Embedding $\phi(g)$

Initialize μ_v^0, for all $v \in \vartheta$
Function $\sigma(t, l_v, l'_v)$
$ret = \sigma_c(l_v) + \sigma_d(l'_v)$
Return ret
for $t = 1 \rightarrow T$ **do**
  for $v \in \vartheta$ **do**
    $l_v = \sum_{i \in N(v)} \mu_i^{t-1}$
    $l'_v = \sum_{j \in D(v)} \mu_j^{t-1}$
    $\mu_v^t = tanh(W_1 x_v + \sigma(t, l_v, l'_v))$
  end for
end for fixed point equation update
return $\phi(g) := W_2(\sum_{v \in \vartheta} \mu_v^T)$



(a) Compiled in level O0 (b) Compiled in level O3

FIGURE 3. Compiled in different optimization level.

of functions than DiscovRE, Genius and Gemini's existing research. We illustrate the idea of adding data flow by example.

Figure 3 is the CFG of two binary functions compiled by GCC 5.4.0 under O0 and O3 optimization level under arm architecture of function SSL_get_peer_certificate. Obviously, the CFG structure of the two functions is somewhat different.

The red dotted block is the data flow information of the function we observe, and the green line refers to a data transfer between the two basic blocks. With the help of data flow information, we can preliminarily judge that there is a strong data dependence and logical correlation between the basic blocks in the red dotted box. After learning, the basic block information inside the red dotted square is propagated to the others in the square through the nonlinear propagation function. After several iterations, each basic block node will contain information about its neighborhood determined by both data flow topology and the involved vertex features. In other words, the model will learn the semantics that expressed by all the basic blocks in the red square.

In our example, the semantics of the basic blocks within the two red dotted squares are roughly the same, that is to say, the CRYPTO_add_lock function is called after a

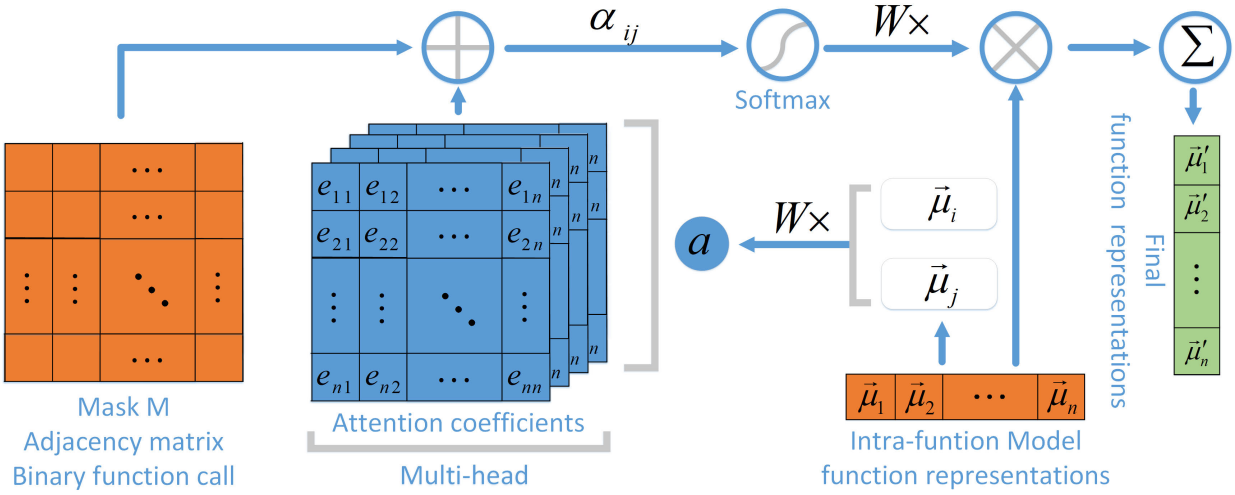


FIGURE 4. GAT model for generating function embedding.

conditional jump. Since the basic blocks within the two red dotted squares have a strong data dependence, the semantics expressed by the basic blocks within the two squares through our model learning will be closer than the semantics expressed by the pure CFG structure learning on Gemini. Therefore, the data flow information alleviates the error caused by the difference of CFG under different compilation environments to some extent.

D. INTER-FUNCTION FEATURE LEARNING MODEL

As mentioned earlier, function calling relationships is a robust semantic under different compilation environments. Since the number of function calls is variable and the relationship between function calls cannot be well expressed on Gemini models, it is more difficult to deal with function calling relationships. Therefore, we need to introduce new mechanisms or models to learn the semantic information of function calls. Ideally, we want the model to have the following characteristics:

(1) the model can act on the neighborhood of the node, i.e., the function calls;

(2) the model can assign different importance to the different adjacent nodes with functions;

(3) the model is suitable for inductive problems and can deal with any untrained graph structure.

It should be noted that the function does not have a fixed number of adjacent nodes in the calling relation matrix. Therefore, inspired by social network, in the construction of the embedded network of the function call relationships, our model is modified on the basis of Petar's GAT [5] network. GAT uses a hidden self-attention layer [17] to handle problems in some graph convolutions. No complicated matrix operations or prior knowledge of the graph structure are required. By stacking the self-attention layer, different importance is assigned to different nodes in the neighborhood during the convolution process, and different sizes of neighborhoods are processed at the same time. In addition, due to the edgewise mechanism, GAT does not depend on the global graph structure and is easy to apply to the induction problem. Inspired by the GAT model, our inter-function feature learning method includes the following main steps, as shown in Figure 4.

The input is a set of function embeddings, that is, the high-dimensional representation of the function generated in the previous stage, $\mu = \{\vec{\mu}_1, \vec{\mu}_2, \ldots, \vec{\mu}_N\}, \vec{\mu}_i \in \mathbb{R}^F$, where N is the number of functions, and F represents the dimensions of features in each function. In addition to this, we also need to feed the adjacency matrix representation $\mathbf{M}$ of the binary file function call to our model. Considering the first-order neighbor node set $\mathcal{C}_i$ of each function node i, our model generates the final function vector representation $\mu' = \{\vec{\mu}'_1, \vec{\mu}'_2, \ldots, \vec{\mu}'_N\}, \vec{\mu}'_i \in \mathbb{R}^{F'}$, according to the attention coefficients α_{ij} of node i for each called function node j.

To transform the input features into a higher-level function embedding that contains the semantics of the function being called, we need to find a way to indicate the importance of function node i to each function node j it called. Fortunately, we perform *self attention* on function nodes, that is, a shared attention mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R}$ to compute the importance; we denote it as *attention coefficients*, $e_{ij} = a(\mathbf{W}\vec{\mu}_i, \mathbf{W}\vec{\mu}_j)$, where $\vec{\mu}_i$ and $\vec{\mu}_j$ are two functions' initial features, $\mathbf{W}$ is a shared parameterized *weight matrix* applied to every feature vertex for linear transformation. Once obtaining the normalized attention coefficients α_{ij}, we can compute the final function embedding after a nonlinear transformation $\vec{\mu}'_i = \sigma\left(\sum_{j \in \mathcal{C}_i} \alpha_{ij} \mathbf{W}\vec{\mu}_j\right)$, where $\mathcal{C}_i$ are the functions called by function i.

For easy understanding, we explain the attention coefficients, which in our model can also be called "importance", through an example. For the third-party library Openssl, the "importance" of *BIO_printf* to *dsa_main* is smaller than the "importance" of *EVP_PKEY_get1_DSA* to *dsa_main*. The reason is a large number of functions call the function *BIO_printf*, and only three functions

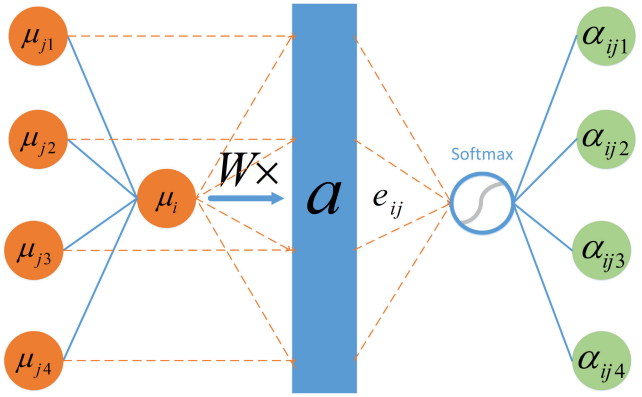


FIGURE 5. Self-Attention mechanism.

dsa_main, *d2i_DSA_PUBKEY*, *pkey_get_dsa* call the function *EVP_PKEY_get1_DSA*, so the effect of the function *EVP_PKEY_get1_DSA* is more important for the recognition of function *dsa_main*. We refer to A, B, C as three functions, *dsa_main*, *BIO_printf*, and *EVP_PKEY_get1_DSA*, respectively. Our example is to illustrates that the "importance" of function B and C to function A does not depend on function A itself, but depends on how well functions B and C can be used to identify function A, that is, when function A calls a function such as the function *EVP_PKEY_get1_DSA* that is called less frequently, A is more easily identified as the function *dsa_main*.

Ideally, we want to see that the model learns different attention weights, that is, one or two function neighbors are much more important than the others. For function A and its two subroutines B and C, "importance" refers to the ability to identify function A through B and C. The reason is that the model finally generates a vector representation of function A which contains the semantics of function B and function C, and if function C is a more special function than function B, then it is easier to find function A through function C, and the final vector representation of A should contain more function C's semantic information to be more easily identifiable.

The following will discuss the implementation of the model. As we mentioned earlier, we need to initialize a weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$ as a linear transformation shared between function nodes. Each vertex also needs a self-attention mechanism to compute the importance of the function vertex j to vertex i, thereby finding a way to obtain the importance between two functions using initial feature vectors.

An overview of the self-attention mechanism is presented in Figure 5. In our model, the attention mechanism is a one-layer feedforward neural network, which realizes the importance of computing nodes through a parameterized weight vector $\vec{a} \in \mathbb{R}^{2\,F'}$, which can be represented as

$$e_{ij} = \text{LeakyReLU}\left(\vec{a}^T \left[\mathbf{W}\vec{\mu}_i \| \mathbf{W}\vec{\mu}_k\right]\right) \tag{5}$$

where LeakyReLU is a nonlinear activation function, $\cdot^T$ represents transposition and $\|$ means the concatenation operation. Considering efficiency issues, only use first-order calls to calculate e_{ij} and normalize e_{ij} using softmax.

$$\alpha_{ij} = \text{softmax}_j\left(e_{ij}\right) = \frac{\exp\left(e_{ij}\right)}{\sum_{k \in \mathcal{C}_i} \exp\left(e_{ik}\right)} \tag{6}$$

Now, the attention coefficient is obtained, and the final function presentation can be calculated by

$$\vec{\mu}_i' = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\vec{\mu}_j\right) \tag{7}$$

We choose *ELU* and *softmax* as $\sigma(\cdot)$ in our model. To ensure the stability of the attention mechanism learning process, a multihead mechanism is added, that is, K independent attention mechanisms are executed at the same time, and the presentation of each function is generated by averaging.

$$\vec{\mu}_i' = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{\mu}_j\right) \tag{8}$$

The intra-function feature learning model is represented in Algorithm 3

Algorithm 3 Inter-Function Embedding Algorithm

Input: Intra-Function Model Output μ, adjacency matrix $\mathbf{M}$
Output: Final Function Embedding $\mathbf{h}'$

Initialize shared parameterized weight matrix $\mathbf{W}$
Initialize parameterized weight vector $\vec{a}$
for $t = 1 \rightarrow T - 1$ **do**
  for $\mu_i \in \mu$ **do**
    $sum_{\mu_i} = \sum_{k \in \mathcal{C}_{\mu_i}} \exp\left(e_{ik}\right)$
    for $h_j \in \mathcal{C}_{\mu_i}$ **do**
      for each head in multihead K **do**
        $e_{ij} = \text{LeakyReLU}\left(\vec{a}^T \left[\mathbf{W}\vec{\mu}_i \| \mathbf{W}\vec{\mu}_j\right]\right)$
        $\alpha_{ij} = \frac{\exp(e_{ij})}{sum_{\mu_i}}$
      end for
      $\vec{\mu}_i^t = \text{ELU}\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{\mu}_j\right)$
    end for
  end for
  $\vec{\mu}_i' = \text{softmax}\left(\sum_{j \in \mathcal{C}_i} \alpha_{ij} \mathbf{W}\vec{\mu}_j^T\right)$
end for

V. EVALUATION

Our prototype consists of two modules, a feature extraction module and a function high-dimensional representation generation module. Like Gemini, we also use the extractor generated by Genius's features, but at the same time, we have made some modifications based on this to extract more accurate original features with more semantic information. The function high-dimensional feature generation module is divided into two parts: for the feature generation of the control flow graph in the function, we modify it on the basis of Gemini so that the model can learn the new original features extracted; for the feature representation of the call between functions, we implemented the graph self-attention network

model through TensorFlow, so that it can fit the scenario of function vulnerability detection.

We validate the validity of our BiN model by comparing Gemini and bipartite graph matching methods. Gemini and VulSeeker [14] provide two baselines for graph embedding, and BGM [45] provides a baseline to evaluate the accuracy of pairwise graph matching approaches. The experiments were conducted on a server with an NVIDIA 1080Ti GPU and an 8 core with 16 threads, 5 GHz Turbo CPU, with 128 GB memory and a 1 TB SSD.

Dataset: In our assessment, we collected 3 data sets.

- **Dataset I - Baseline evaluation.** We obtain the binary file for training by compiling the source code and keep the name of the original function as a criterion for determining the similarity of the function, so this data set can be compared well with the ground true label for the homology function under different architectures, compilers and optimization levels. We have used GCC v5.4 to compile OpenSSL (version 1.0.1e and 1.0.1u) and BusyBox (version 1.27.2). The compiler is set to emit code in x86, AMD64, MIPS, MIPS64 and ARM, ARM64, with optimization levels O0 - O3. In addition, we extract the function features by using disassembler IDA 6.8.
- **Dataset II - Real-world dataset.** We crawled the firmware from the network, including routers, IP cameras, printers, etc., and involving vendors such as Cisco, D-link, Tp-link, Dahua, Hikvision, HP, and Epson, including 21,350 images. In total, 8753 images can be unpacked successfully.
- **Dataset III - Vulnerability dataset.** To build a vulnerability database that can be used for searching, we need to obtain the binary code of the vulnerability function. Therefore, we found the vulnerability in the open source library widely used in the firmware on the official website and recorded the firmware version, the name of the function, and so on. We extract the vulnerability function feature from the corresponding version firmware according to the function name and other features, use the prototype BiN to generate the high-dimensional feature representation of the vulnerability function, and store it in a database. Ultimately, we obtained 84 vulnerability functions.

A. ACCURACY

We implemented three methods, Gemini, VulSeeker and BGM, and conducted comparisons with them on dataset I. We evaluated five groups of functions selected in dataset I, which are Dataset I(All), Dataset I(32bit), Dataset I(BGMWeak), Dataset I(Inlined function) and Dataset I(BGMWeak except inlined function). Since dataset I is compiled on the source code, we can easily identify the same function in different files and accurately define the ground truth label. Figure 6 illustrates the ROC curves for our model (BiN) and the other three baseline approaches. We can see that BiN is more accurate than Gemini, Genius and BGM.

For the first group, we evaluated it on dataset I(All). We randomly selected 4000 pairs of the same functions and 4000 pairs of different functions and divided them into three parts: train, validation, and test in a ratio of 8:1:1. We treat two different compiled versions of the same source function as a pair of similar functions. We performed experiments on these four implementations, all of which were configured with optimal parameters.

We observed that the BiN ROC curve is higher than that for Gemini, which means that it is more likely for BiN to place the positive sample in front of the negative sample, that is, the model judges better. The AUC and ACC values of BiN were 92.73% and 90.12%, which were 12.47% and 19.23% higher than Gemini and 4.24% and 8.83% higher than VulSeeker, respectively.

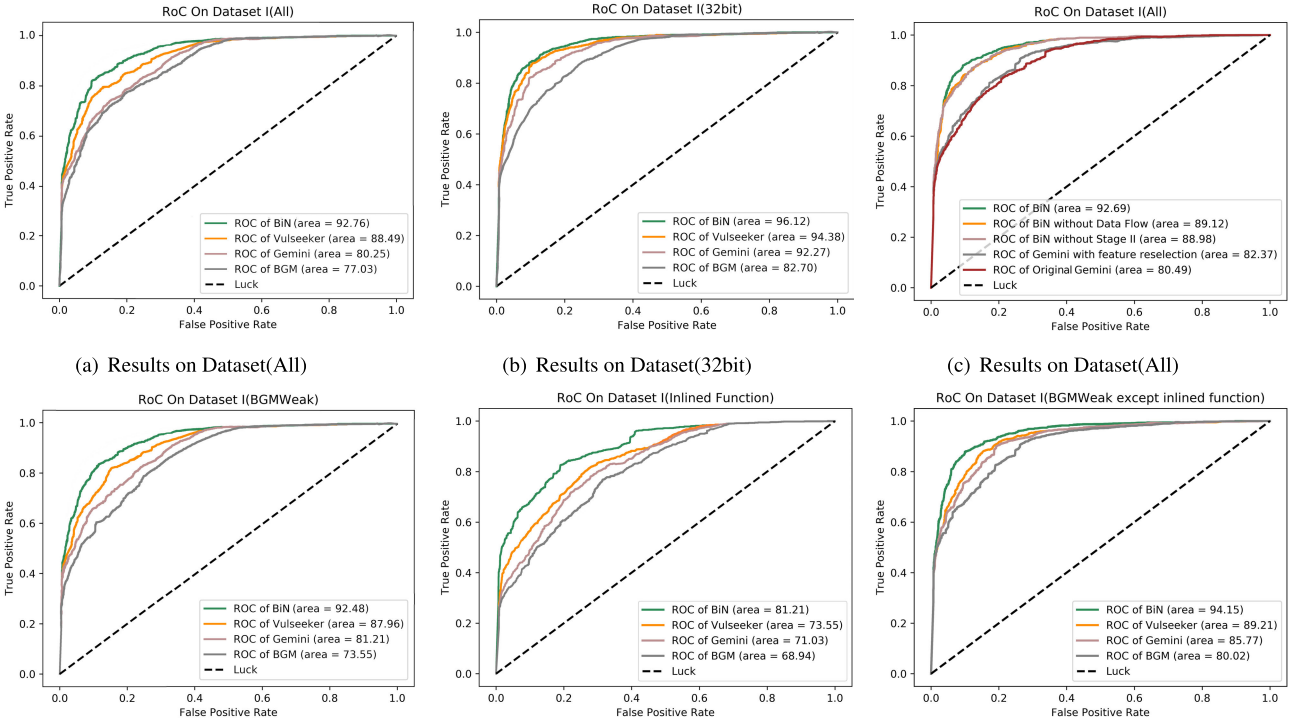
In the first set of evaluations, Gemini's experimental results were somewhat different from the descriptions in the paper. We believe that our data set contains 64-bit binaries, and Gemini's support for this is not very good. VulSeeker has a clear optimization of 64-bit support. It reselects features and adds DFG, but its accuracy can still be improved.

In the second set of evaluations, we evaluated on dataset(32bit),which only selected the 32-bit binaries compiled by OpenSSL. We can see that the Gemini's ROC is close to 93%, that VulSeeker can obtain a 94% AUC value, and that our model increases the value to 96%.

In the third group of evaluations, we demonstrated whether the improvement of BiN is effective. Our improvements mainly include the feature reselection using by the model-oriented GA, data flow and the inter-function feature learning. Therefore, we compared the original Gemini, Gemini with feature reselection, BiN without Stage II, BiN without Data Flow, and BiN for evaluation. The AUC of original Gemini raised 1.88% by reselecting features. And the AUC of BiN without Stage II increased to 88.98% by adding data flow information on Gemini with feature reselection, while when Gemini with feature reselection adds Stage II, the AUC go up to 89.12%, that is close to BiN without Stage II. The results proved that these improvements have positive effects. When we combined all improvements, the AUC improved to 92.69%, better than others.

In addition to the above three sets of assessments, we performed three sets of evaluations on the set of functions that did not perform well in Gemini. We define that the same function does not perform well when the similarity score is less than 0.7 in Gemini under different compilation environments, and we constitute the BGMWeak dataset. Further, we found that the inline function is an important factor in the low similarity score in the dataset. Therefore, we also evaluated the accuracy of the model, facing the key challenge of inline function. We chose the inline function in BGMWeak as a test set. In contrast, we performed a further evaluation of the remaining functions in BGMWeak.

As illustrated in Figure 6(d), Gemini and VulSeeker are not ideal in this dataset. The graph matching method or the



(a) Results on Dataset(All) (b) Results on Dataset(32bit) (c) Results on Dataset(All)

(d) Results on Dataset(BGMWeak) (e) Results on Dataset(Inlined function) (f) Results on Dataset(BGMWeak except inlined function)

FIGURE 6. Comparison result on dataset I.

features provided by CFG cannot solve the impact of the different optimization options and architecture of the function. Furthermore, for inline functions, the performance of the three models is not very satisfactory, but BiN still has some progress in the identification of inline function compared to Gemini and VulSeeker.

For the functions other than inline function, all models have been improved, and the effect of BiN is the most obvious. The AUC can reach 94.15%, which is 4.94% and 8.38% higher than VulSeeker and Gemini respectively. We illustrate these two evaluations in Figure 6(e) and 6(f). Fortunately, BiN has a proven performance in these instances, proving that our model can further mitigate the impact of optimization options and architecture.

In summary, BiN is better than Gemini and VulSeeker in our evaluation. This finding is because we added data flow information attach to the CFG to track the data transfer between basic blocks and added information about function calls. In the process of function semantic generation, BiN obtains more powerful semantic information through the learning of two-level features, which is beneficial for the effective recognition of homologous functions. However, the study on the similarity of inline functions should still be explored.

B. EFFICIENCY

We use the firmware function of Dataset II to evaluate the model efficiency. Our evaluation is divided into two parts: (1) the function feature extraction and (2) the function high-dimensional feature generation.

For the extraction of function features, we compare the time at which the function is extracted in the Gemini method to use as the baseline. Gemini extracts the six basic intrablock features in the function and the structural features between the two basic blocks. The 8 intrablock features and an interblock feature extracted by our model extract the data stream characteristics of the function and obtain the features. The call information of the function.

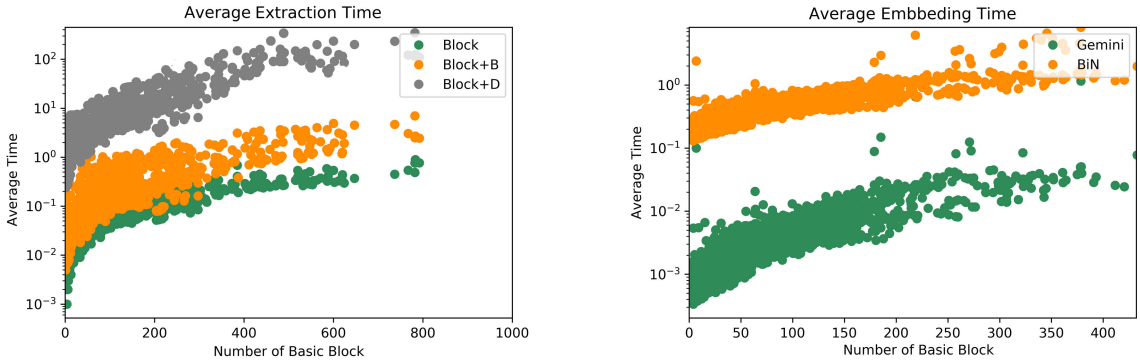
For the high-dimensional feature generation phase of the function, we still use Gemini as a baseline for comparison. Note that our evaluation is based on all functions in the binary, since the call information for the function is extracted from the entire binary.

1) FUNCTION FEATURES EXTRACTION TIME

Figure 7a illustrates the results. We can observe that the time required for Gemini to extract features is more than the time required to extract CFG features and function call information. However, after adding the extraction of the function data stream, the time is significantly increased, which is higher than the extraction time of all other schemes. This increase in time is tolerable because it is significantly smaller than Genius's extraction time, approximately 2× on Gemini.

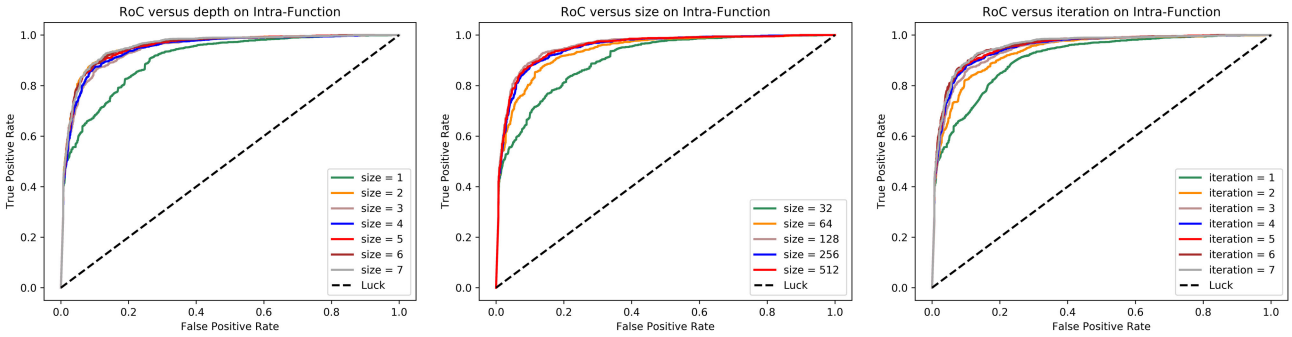
2) FUNCTION REPRESENTATION GENERATION TIME

The generation time is represented in Figure 7b. We can observe that generation representations in our model and



(a) Oringinal Features Extraction Time (b) Embedding Features Vector Generation Time

FIGURE 7. **Efficiency evaluation on dataset III. The figure plots one point for each sample selected in dataset III.**



(a) ROC versus depth-Intra (b) ROC versus size-Intra (c) ROC versus iteration-Intra

FIGURE 8. **Evaluation of different hyperparameters of the intra-model on dataset III.**

Gemini require similar time. Although our model is more complex than Gemini, most of the operations of the model use matrix operations, and parallelization can be achieved by the underlying multicore CPU, so the time increment is acceptable. It should be stated that due to the choice of model parameters, the final performance of the model will be greatly affected. Therefore, our evaluation is carried out employing the parameters with the best results in our experiments.

C. HYPERPARAMETERS

In this section, we evaluate the effectiveness of hyperparameters in the BiN model. Our evaluation is divided into two parts.

1) INTRA-FUNCTION MODEL HYPERPARAMETERS

Since our intra-function model has changed on Gemini, we need to re-evaluate the hyperparameters such as the embedding depth, embedding size, and iteration number. Our initial settings are the same as those of Gemini, and we control the variables to test the hyperparameters one-by-one to find the most appropriate parameter settings, where the embedding size p is 64, the embedding depth n is 2, and the number of iterations T per basic block is 5.

a: EMBEDDING DEPTH

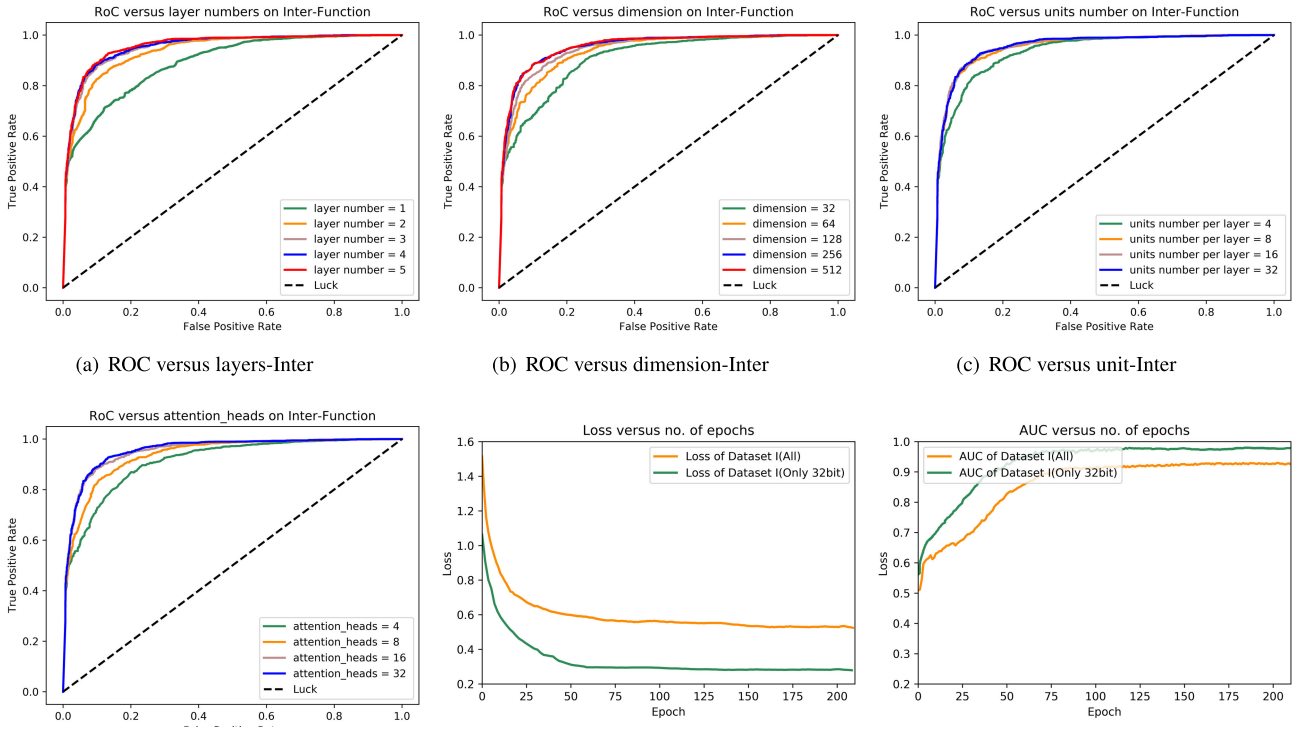
We changed the depth of function embedding in the intra-function model and visually demonstrated the effects of different parameters through the ROC curve. From Figure 8a, we observe that when the embedding depth is 2, the ROC curve can reach a higher level, and when the number of layers is continuously increased, the improvement effect is almost negligible.

b: EMBEDDING SIZE

We vary the number of embedding sizes, and we can observe that the ROC curve reached a stable level when the embedding size is 128, as shown in Figure 8b. The larger sizes also obtain curves close to size 128, and the larger the size is, the longer the evaluation time is; thus, we choose the embedding size to be 128, considering the efficiency.

c: NUMBER OF ITERATIONS

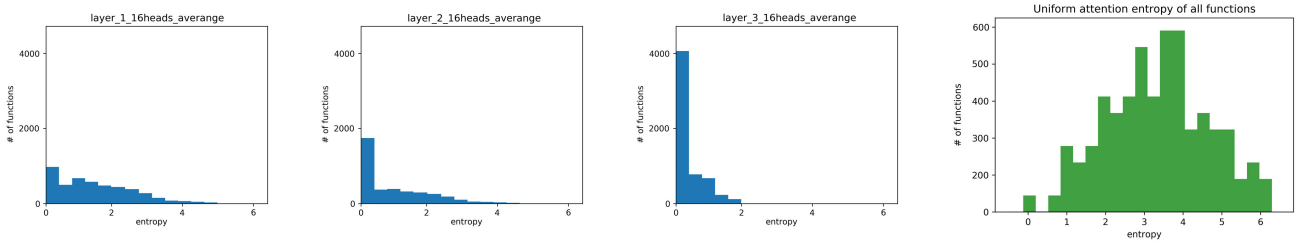
We observe that the model achieves the largest AUC value when the number of iterations is larger than 4 in Figure 8c. We trade off efficiency and accuracy and choose 5 as the number of iterations.



(a) ROC versus layers-Inter (b) ROC versus dimension-Inter (c) ROC versus unit-Inter

(d) ROC versus attention head-Inter (e) Loss versus epoch-Overall (f) AUC versus epoch-Overall

FIGURE 9. Evaluation of different hyperparameters of inter-model and overall on dataset III.



(a) Layer 1 attention weight (b) Layer 2 attention weight (c) Layer 3 attention weight (d) Uniform attention weight distribution

FIGURE 10. Attention weight entropy of different layers of inter-model on dataset I.

As a result, the embedding size p is 128, the embedding depth n is 3, and the number of iterations T per basic block is 5.

2) INTER-FUNCTION MODEL HYPERPARAMETERS

For the inter-function model, we examine the impact of the layer number, output vector dimension, units per layer, attention heads, etc.

a: LAYER NUMBER

We changed the depth in the inter-function model and visually demonstrated the effects of different parameters through the ROC curve as well. From Figure 9a, we observe that when the layer number is more than 3, the ROC curves reach a stable level, so we set 3 layers in our model.

b: OUTPUT VECTOR DIMENSION

The output vector is the final presentation of a function, and we set different numbers of dimensions for our model to generate the vector. As seen in Figure 9b, the larger the vector is, the larger the AUC value that we obtain. Considering the efficiency, we choose a 128-dimension vector finally.

c: UNITS PER LAYER

We have chosen different numbers of hidden units per each attention head in each layer to determine the final choice. We can observe in Figure 9c that 8 is a good trade-off choice.

d: ATTENTION HEADS

Note that the final layer must have only one head, so the choice we made is on the other layers. In Figure 9d,

we observe that the ROC curves have a good performance when attention heads are 8 or more.

In order to verify that the attention distribution learned by our model can distinguish the importance of different functions under these parameters, we introduce the entropy of attention distribution [46]. For any function f_i, $\{\alpha_{ij}\}_{j\in\mathcal{N}(i)}$ forms a discrete probability distribution over all functions f_i called with the entropy given by

$$E\left(\alpha_{ij_{j\in\mathcal{N}(i)}}\right) = -\sum_{j\in\mathcal{N}(i)} \alpha_{ij}\log\alpha_{ij} \tag{9}$$

That is to say, low entropy means a high concentration and vice versa. An entropy of 0 means that all attention is concentrated on one function node.

We performed a verification on dataset I. Since the function nodes can have different degrees and the maximum entropy is different, we plot the aggregate histogram of the entropy values of all functions in the binary file. Figure 10 are the attention histograms learned at different layers. As a reference, Figure 10(d) is the histogram if all functions have uniform attention weight distribution. The entropy value we expressed is the average of each attention head. Clearly, GAT does learn sharp attention weights, and the attention gets sharper in 3 layers.

As a result, the layer number is 3, the output vector dimension p' is 128, the units per layer m is 8, and the number of attention heads for each layer is 16.

3) OVERALL HYPERPARAMETERS

The variation of the number of epochs is also within the scope of our research. As seen in Figure 9e and 9f, our model achieves a relatively stable result at approximately 85 epochs.

D. ACCURACY OF VULNERABILITY SEARCH

We use Dataset I to evaluate the effectiveness of BiN in vulnerability searches and compare Gemini as a baseline. We extract the features of the function on data set II, involving a total of 3,619,200 functions, and select the vulnerability function from Dataset III to search. To facilitate comparison with Gemini, we selected the same CVE-2015-1791 [23] and CVE-2014-3508 [24] vulnerabilities as Gemini and extracted G0 compiled O0-O3 under x86, MIPS, and ARM, three 32-bit architectures. We extract the features of the functions in the real world firmware and input them into the BiN model, which then generates a final vector representation of the function and compares it with the vulnerability function in Dataset III to find the vulnerability function candidate set. Four optimization levels of OpenSSL's corresponding vulnerability functions ssl3_get_new_session_ticket and OBJ_obj2txt are searched.

We examined the validity of the top-K most similar results of the search results and compared them with the results of Gemini. Table 2 is our result. The K value represents the selection of K functions, the second column is the number of functions in the first K results, and the third column is the corresponding percentage. For each firmware, we searched with 12 functions of 4 optimization levels for each vulnerability, sorted the functions according to the descending order of similarity, and took the average of 12 function search results. For these 2 CVEs, we discovered the vulnerability in vendors such as D-link, Tp-link, Netgear and Cisco. The result shows BiN On average, the vulnerability function can be found in the 9th bit, where the maximum is 22, while the Gemini is ranked 47 and the maximum is 189. A comparison of all vulnerability functions search result between BiN and Gemini is shown in Table 3 in Section Appendix, the average rank of BiN is 8.78, while the average rank of Gemini goes up to 25.73.

TABLE 2. Accuracy of vulnerability search.

Model	Top-K	Number of Vulnerable Functions	Percent
BiN	5	3.2	64%
	10	6.3	63.0%
	20	15.6	78.0%
	50	42.6	85.2%
	100	87.9	87.9%
Gemini	5	2.1	42.0%
	10	3.8	38.0%
	20	10.4	52.0%
	50	35.3	70.6%
	100	76.3	76.3%

The experimental results demonstrated that the accuracy of our model in real-world vulnerability search could reach up to 87.9%, which is a great improvement on the search accuracy compared with Gemini. In our model, we only need to manually analyze the top-50 candidate vulnerability functions to achieve better results than Gemini's top-100 vulnerability function candidates.

VI. RELATED WORK

We discussed our work in detail throughout the paper, and in this section, we will briefly introduce other related work. Our work focuses on finding known vulnerabilities through code similarity in cross-architecture binaries. This section does not discuss ways to explore unknown vulnerabilities.

A. A BUG SEARCH BASED ON ORIGINAL FEATURES

Many of the current methods for searching for bugs in cross-architecture binaries are mostly expensive. For example, both Zynamics BinDiff [18] and BinSlayer [19] use an expensive graph isomorphism to quantify code similarity by calculating the similarity of control flow graphs. Pewny et al. [20] used the MinHash method to optimize the search matching time, but the graph matching algorithm still could not cope with the vulnerability search of large-scale firmware. DiscovRe [1] filters and selects robust CFG features by preprocessing to further improve the efficiency of graph matching, but the accuracy of prefiltering is difficult to guarantee. On this basis, Pewny Genius uses the ACFG graph to form a codebook to generate a high-dimensional feature representation of the function to obtain the function similarity and optimizes the search process through LSH; the efficiency is greatly improved, but the performance in the million-level function search is still poor.

TABLE 3. Vul-func comparison between BiN and Gemini.

CVE number	vendors	vulnerable function	Averange rank of BiN	Averange rank of Gemini
CVE-2012-2110	openssl	asn1_d2i_read_bio	10.51	21.08
CVE-2013-1944	libcurl	tailmatch	6.17	22.64
CVE-2013-2174	libcurl	curl_easy_unescape	9.70	38.81
CVE-2014-0088	nginx	ngx_http_spdy_module	8.18	23.82
CVE-2014-0195	openssl	dtls1_reassemble_fragment	10.03	22.01
CVE-2014-0198	openssl	do_ssl3_write	5.83	41.06
CVE-2014-0221	openssl	dtls1_get_message_fragment	13.60	24.20
CVE-2014-0244	samba	sys_recvfrom	12.67	19.36
CVE-2014-2855	samba	check_secret	8.76	18.42
CVE-2014-3470	openssl	ssl3_send_client_key_exchange	3.87	39.06
CVE-2014-3493	samba	push_ascii	10.00	24.21
CVE-2014-3508	openssl	OBJ_obj2txt	8.57	29.30
CVE-2014-3509	openssl	ssl_parse_serverhello_tlsext	12.99	18.18
CVE-2014-3510	openssl	ssl3_send_client_key_exchange	7.51	8.63
CVE-2014-3567	openssl	tls_decrypt_ticket	8.60	23.21
CVE-2014-3569	openssl	ssl23_get_client_hello	14.62	43.14
CVE-2014-3571	openssl	dtls1_get_record	10.84	23.02
CVE-2014-3707	libcurl	curl_easy_duphandle	7.19	10.47
CVE-2014-5139	openssl	ssl_set_client_disabled	4.32	12.76
CVE-2014-8150	libcurl	parse_url_and_fill_conn	11.80	38.98
CVE-2014-8176	openssl	dtls1_clear_queues	8.49	37.08
CVE-2014-9116	mutt	write_one_header	3.53	12.63
CVE-2014-9645	busybox	add_probe	3.01	30.15
CVE-2015-0206	openssl	dtls1_buffer_record	10.54	14.85
CVE-2015-0207	openssl	dtls1_listen function	12.86	39.37
CVE-2015-0208	openssl	rsa_item_verify	3.06	8.65
CVE-2015-0209	openssl	d2i_ECPrivateKey	4.41	10.55
CVE-2015-0240	samba	_netr_ServerPasswordSet	11.72	42.20
CVE-2015-0286	openssl	ASN1_TYPE_cmp	12.81	48.80
CVE-2015-0287	openssl	ASN1_item_ex_d2i	13.76	11.09
CVE-2015-0288	openssl	X509_to_X509_REQ?	11.49	18.40
CVE-2015-0290	openssl	ssl3_write_bytes	7.17	27.36
CVE-2015-0292	openssl	EVP_DecodeUpdate	13.39	43.73
CVE-2015-1787	openssl	ssl3_get_client_key_exchange	2.50	22.90
CVE-2015-1788	openssl	BN_GF2m_mod_inv	4.59	27.48
CVE-2015-1789	openssl	X509_cmp_time	7.18	24.83
CVE-2015-1790	openssl	PKCS7_dataDecode	10.29	13.67
CVE-2015-1791	openssl	ssl3_get_new_session_ticket	9.13	36.63
CVE-2015-1792	openssl	do_free_upto	7.22	23.57
CVE-2015-1794	openssl	ssl3_get_key_exchange	10.86	30.92
CVE-2015-3195	openssl	ASN1_TFLG_COMBINE	10.02	14.25
CVE-2015-3216	openssl	ssleay_rand_bytes function	2.85	17.49
CVE-2015-3223	samba	ldb_wildcard_compare	8.96	15.74
CVE-2015-3237	libcurl	smb_request_state	7.26	34.83
CVE-2015-3310	samba	rc_mksid	10.41	12.62
CVE-2015-5194	ntp	log_config_command	12.41	28.58
CVE-2015-5219	ntp	ulogtod	8.29	44.33
CVE-2015-5299	samba	shadow_copy2_get_shadow_copy_data	12.51	43.44
CVE-2015-7691	ntp	crypto_xmit	10.19	19.92
CVE-2015-7701	ntp	crypto_assoc	8.12	29.77
CVE-2015-8467	samba	samldb_check_user_account_control_acl	3.05	11.75
CVE-2015-9261	busybox	huft_build	9.46	20.52
CVE-2016-0701	openssl	DH_check_pub_key	11.89	17.49
CVE-2016-0702	openssl	MOD_EXP_CTIME_COPY_FROM_PREBUF	14.38	39.78
CVE-2016-0703	openssl	get_client_master_key	3.85	30.31
CVE-2016-0704	openssl	get_client_master_key	7.53	31.78
CVE-2016-0705	openssl	dsa_priv_decode	5.22	21.90
CVE-2016-0797	openssl	BN_dec2bn or BN_hex2bn	12.39	24.14
CVE-2016-0798	openssl	SRP_VBASE_get_by_user	11.20	7.52
CVE-2016-0799	openssl	fmtstr	7.54	36.50
CVE-2016-2105	openssl	EVP_EncodeUpdate	9.19	33.92
CVE-2016-2106	openssl	EVP_EncryptUpdate	11.35	45.79
CVE-2016-2176	openssl	X509_NAME_oneline	6.31	23.55
CVE-2016-2178	openssl	dsa_sign_setup	13.90	27.65
CVE-2016-2180	openssl	TS_OBJ_print_bio	5.81	11.39
CVE-2016-2182	openssl	BN_bn2dec	9.23	12.66

TABLE 3. ***(Continued.)*** **Vul-func comparison between BiN and Gemini.**

CVE number	vendors	vulnerable function	Averange rank of BiN	Averange rank of Gemini
CVE-2016-2842	openssl	doapr_outch	2.67	41.84
CVE-2016-4954	ntp	process_packet	5.63	32.07
CVE-2016-6301	busybox	recv_and_process_client_pkt	13.05	31.31
CVE-2016-6302	openssl	tls_decrypt_ticket	17.54	39.72
CVE-2016-6303	openssl	MDC2_Update	8.26	21.13
CVE-2016-6305	openssl	ssl3_read_bytes	14.60	38.27
CVE-2016-7434	ntp	read_mru_list	14.93	21.96
CVE-2017-15873	busybox	get_next_block	9.13	22.30
CVE-2017-16544	busybox	add_match	8.08	24.46
CVE-2017-16548	samba	receive_xattr	10.72	11.32
CVE-2017-17433	samba	recv_files	6.95	17.62
CVE-2017-17434	samba	read_ndx_and_attrs	3.80	9.61
CVE-2017-6451	ntp	mx4200_send	12.72	45.84
CVE-2018-16845	nginx	ngx_http_mp4_module	5.60	22.66
CVE-2018-16890	libcurl	ntlm_decode_type2_target	13.57	32.59
CVE-2018-20679	busybox	udhcp_get_option	9.23	18.53
CVE-2018-5764	samba	parse_arguments	7.89	19.69
CVE-2018-7182	ntp	ctl_getitem	4.68	25.60

B. A BUG SEARCH BASED ON DEEP LEARNING

Xu Gemini [4] uses the extracted features to learn to generate graph embedding on the DNN model to efficiently generate function feature embedding and perform function similarity comparison. However, this method does not fundamentally solve the limitation of similarity comparison using graph matching and generates semantics that embed the inability to express the function accurately. Wang added a second phase to improve accuracy, but its method is less efficient and cannot perform accurate vulnerability mining for large-scale data. VulSeeker [14] adds a new semantic feature DFG when extracting function semantics, which improves the integrity of functional semantic expression, but its characteristics are still limited to the internal features of the function, lacking the semantic information of the function call relationship.

C. DEEP LEARNING-BASED GRAPH EMBEDDING APPROACHES

CNN cannot process data of non-Euclidean structure [47], that is, the number of adjacent vertices of each vertex in the topology map must be the same, but since the number of function calls is not fixed, it is impossible to use convolution kernels of the same size to perform a convolution operation. The GCN [21] attempts to generalize and apply the neural network to any graph structure data, but this method cannot adapt to the new graph structure and cannot be used for function search in firmware.

VII. CONCLUSION

In this paper, we propose a two-level feature-based neural network method, which is used to generate high-dimensional feature representations of binary functions in an attempt to solve the problem of detecting vulnerabilities in large-scale IoT devices in the real world. We implemented a prototype called BiN. In our evaluation, BiN outperforms the state-of-the-art approaches in terms of the accuracy and scope of the function similarity detection and only increased an acceptable time overhead. In the search for real-world function vulnerabilities, BiN can more accurately identify the vulnerability function. We believe that our model has made effective progress in the application of deep learning in security issues. However, our model does not completely solve the problem of inline function to binary function similarity comparison, the study of inline function should be continued.

APPENDIX

See Table 3.

REFERENCES

[1] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. NDSS*, San Diego, CA, USA, Feb. 2016.

[2] F. Qian, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 480–491.

[3] P. Bryan, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 701–710.

[4] 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 *Proc. SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.

[5] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*. [Online]. Available: https://arxiv.org/abs/1710.10903

[6] L. F. R. Ribeiro, P. H. Saverese, and D. R. Figueiredo, "Struc2vec: Learning node representations from structural identity," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2017, pp. 385–394.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] A. K. Srivastava, "Function embedding generation using program dependency graph based neural network," Ph.D. dissertation, Univ. California, Riverside, California, CA, USA, 2018.

[9] A. Chandran, L. Jain, S. Rawat, and K. Srinathan, "Discovering vulnerable functions: A code similarity based approach," in *Proc. Int. Symp. Secur. Comput. Commun.* Singapore, Springer, 2016, pp. 390–402.

[10] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, Mar. 2015, pp. 261–270.

[11] S. Glesner, and J. O. Blech, "Classifying and formally verifying integer constant folding," *Electron. Notes Theor. Comput. Sci.*, vol. 82, no. 2, pp. 410–425, 2004.

[12] *The IDA Pro Disassembler and Debugger*. Accessed: May 18, 2019. [Online]. Available: http://www.datarescue.com/idabase/

[13] *MIASM. Reverse Engineering Framework*. Accessed: May 18, 2019. [Online]. Available: https://github.com/cea-sec/ miasm

[14] G. Jian, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 896–899.

[15] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2702–2711.

[16] W. Xu, Y. Li, Y. Tang, and B. Wang, "Research on cross-architecture vulnerabilities searching in binary executables," *Netinfo Secur.*, vol. 9, p. 6, 2017.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017.

[18] D. Thomas, and R. Rolles, "Graph-based comparison of executable objects," *SSTIC*, vol. 5, no. 1, p. 3, 2005.

[19] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proc. 2nd SIGPLAN Program Protection Reverse Eng. Workshop*, 2013, p. 4.

[20] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," Sep. 2016, *arXiv:1609.02907*. [Online]. Available: https://arxiv.org/abs/1609.02907

[22] J.-B. Hou, T. Li, and C. Chang, "Research for vulnerability detection of embedded system firmware," *Proc. Comput. Sci.*, vol. 107, pp. 814–818, 2017.

[23] *CVE-2015-1791*. Accessed: Jun. 16, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0224

[24] *CVE-2014-3508*. Accessed: Jun. 16, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0224

[25] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, "A survey on security and privacy issues in Internet-of-Things," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1250–1258, Oct. 2017.

[26] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αDiff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 667–678.

[27] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," in *Proc. 24th SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 678–689.

[28] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proc. Asia Conf. Comput. Commun. Secur.*, 2017, pp. 346–359.

[29] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, Vancouver, BC, Canada, 2017, pp. 253–270.

[30] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," Tech. Rep., 2013.

[31] D. Minoli, K. Sohraby, and J. Kouns, "IoT security (IoTSec) considerations, requirements, and architectures," in *Proc. 14th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2017, pp. 1006–1007.

[32] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework for dynamic security analysis of embedded systems' firmwares," in *Proc. 21st Symp. Netw. Distrib. Syst. Secur. (NDSS)*, Reston, VA, USA: Internet Society, 2014.

[33] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. NDSS*, 2016, pp. 1–22.

[34] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res. Colocated NDSS Symp.*, vol. 18, Feb. 2018, pp. 1–11.

[35] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 392–404.

[36] Y. Wang, J. Shen, J. Lin, and R. Lou, "Staged method of code similarity analysis for firmware vulnerability detection," *IEEE Access*, vol. 7, pp. 14171–14185, 2019.

[37] *Github*. Accessed: May 18, 2019. [Online]. Available: https://github.com/

[38] S. Alrabaee, L. Wang, and M. Debbabi, "*BinGold*: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs)," *Digit. Invest.*, vol. 18, pp. S11–S22, Aug. 2016.

[39] H.-J. Song, S.-B. Park, and S. Y. Park, "Computation of program source code similarity by composition of parse tree and call graph," *Math. Problems Eng.*, vol. 2015, Dec. 2014, Art. no. 429807.

[40] T. Nakahira, and M. Haraguchi, "Loop optimization compile processing method," U.S. Patent 5 842 022 A, Nov. 24, 1998.

[41] P. P. Chang and W.-W. Hwu, "Inline function expansion for compiling C programs," in *Proc. ACM SIGPLAN Symp. Interpreters Interpretive Techn.*, 1989, vol. 24, no. 7, pp. 246–257.

[42] J. Yang, and V. Honavar, "Feature subset selection using a genetic algorithm," in *Feature Extraction, Construction and Selection*. Boston, MA, USA: Springer, 1998, pp. 117–136.

[43] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. 16th Conf. Comput. Commun. Secur.*, 2009, pp. 611–620.

[44] P. Stone, and M. Veloso, "Layered learning," in *Proc. Eur. Conf. Mach. Learn.*, Berlin, Germany: Springer, 2000.

[45] K. Riesen, and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image Vis. Comput.*, vol. 27, no. 7, pp. 950–959, 2009.

[46] S. T. Kay, "The entropy distribution in clusters: Evidence of feedback?" *Monthly Notices Roy. Astronomical Soc.*, vol. 347, no. 2, pp. L13–L17, 2004.

[47] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3844–3852.

HAO WU was born in 1995. He received the B.S. degree in network engineering from Information Engineering University, Zhengzhou, in 2017. He is currently pursuing the M.S. degree in cyberspace security with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include the Internet of Things security and binary program analysis.

HUI SHU was born in 1974. He received the Ph.D. degree in computer science and technology from Information Engineering University, in 2001. He is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include reverse analysis and the IoT security.

FEI KANG was born in 1972. She is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. Her research interests include network security mechanism analysis and malware analysis.

XIAOBING XIONG was born in 1985. He received the Ph.D. degree in computer science and technology from Information Engineering University, in 2013. He is currently an Assistant Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include reverse analysis and malware detection.