IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

BEDetector: a Two-channel Encoding Method to Detect Vulnerabilities based on Binary Similarity

LU YU[1,2], YULIANG LU[1,2], YI SHEN[1,2], HUI HUANG[1,2], KAILONG ZHU[1,2].
[1]College of Electronic Engineering, National University of Defense Technology, Hefei, 230007, China
[2]Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei, 230007, China

Corresponding author: Yuliang Lu (e-mail: lulu071227@163.com).

ABSTRACT Applying neural network technology to binary similarity detection has become a promising search topic, and vulnerability detection is an important application field of binary similarity detection. When embedding binary code into matrix by neural network, the problem of feature representation also needs to be solved in vulnerability detection. However, most of the current researches extract the syntax or structural features of binary code, and take basic block as the minimum analysis unit, which is relatively coarse. In addition, the structural features of binary functions are usually represented by the dependency graph. In the embedding process, only the neighbour information of the node can be obtained, ignoring the global information of the graph. To solve these two problems, we propose a two-channel feature extraction method to obtain semantic feature in finer granularity and represent the structural features globally instead of locally. Inspired by natural language process, we propose a contextual semantic feature extraction method to obtain different granularity features of binary functions. It takes instruction as the minimum analysis unit and obtains the semantic relationship between instructions. Meanwhile, in order to represent the structural feature of each function, we propose a neural GAE model instead of the widely used structure2vec model. In this way, we can preserve and reconstruct the control dependencies between the basic blocks in the whole graph. We have implemented a prototype system BEDetector, evaluated the effectiveness of its neural model and compared the accuracy of vulnerability function detection with state-of-the-art system. Besides, we choose the real-world firmware files as the detection target and prove that BEDetector can achieve a relatively high detection rate. BEDetector could reach a precision of 88.8%, 86.7% and 100% when ranking top-50 candidate functions in the detection of the CVE vulnerability function $ssl3\_get\_key\_exchange$, $ssl3\_get\_new\_session\_ticket$ and $udhcp\_get\_option$, proving the efficiency of our method.

INDEX TERMS
semantic analysis, binary code embedding, graph autoencoder, binary similarity detection

I. INTRODUCTION

BINARY function similarity detection in software analysis allows one to analyze whether two binary functions are similar without accessing the corresponding source code. It has a wide range of applications in the field of security, such as vulnerability detection, malware detection, plagiarism detection and so on. Software vulnerability is an important strategic resource in cyberspace. The exploitation of vulnerabilities will lead to more and more serious security problems, which makes vulnerability detection become a crucial research area. Due to code reuse and sharing, vulnerabilities in third-party libraries may spread between devices of different hardware architectures and software platforms. In the Synopsys 2020 Open Source Security and Risk Analysis Report [1], it is found that 99% of the codebases audited in 2019 contain open source components, and 75% of the audited codebases contain at least one public vulnerability. An average of 82 vulnerabilities were identified in each code base. The average age of vulnerabilities found in the audited codebases (since the first release) is 4.5 years. The percentage of vulnerabilities released over 10 years ago is 19%, and the earliest one was released in 1999 (CVE-1999-0061). When developers apply third-party code to their own projects, they usually modify the third-party code slightly, or compile the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access

code in different optimization levels. So traditional methods based on code hash or graph isomorphism can not detect such vulnerabilities. With the rise of IoT (Internet of things) devices, third-party code library has been applied to different architectures. Considering the memory limitation, IoT manufacturers may use older version of the third-party library, and rarely update the firmware in time after the release of vulnerability. Such security issue in cross-architectures has also attracted the attention of researchers.

The binary code similarity detection technology encounters more difficulties than the relatively mature open-source code similarity detection method [2] [3] [4] [5] [6] [7]. During compiling, the code is usually optimized, such as function inlining, redundancy elimination, instruction reordering and conversion, etc. In addition, binary files compiled in different architectures vary in operands and operators, which makes binary similarity detection more difficult. Earlier work [8] on binary similarity detection compares the control flow graph (CFG) of two binary files. Symbolic execution and theorem proving [9] [10] are also proposed to check the semantic equivalence of code. However, the theorem proving based method is not scalable. Binjuice [11] normalizes the instructions inside one basic block to extract its semantic "juice" and compare the words of "juice" to detect similarity. However, this approach works only at the basic block level. David et al. [12] define a set of codes as execution tracelet. The similarity between two execution tracelets is calculated by measuring how much rewriting is required to get from one trace to another. This method is also not robust to compiler optimization. Pewny et al. [13] firstly apply binary vulnerability signature to search for known vulnerabilities in different architectures. They generate the vulnerability signature by Best Hit Broadening(BHB) algorithm. This method is relative precise with semantic similarity but is not scalable to larger code database. Nouh et al. [14] match binary functions through code fingerprints. The large-scale fingerprint matching is implemented by decomposing CFG into multiple execution paths with semantic information and obtaining the structural features of CFG. But it can not detect function inlining or deal with code obfuscation technology such as control flow flattening.

Due to the difficulty and limitation of earlier research, recent work apply machine learning method to binary similarity detection, especially cross-architecture vulnerability detection. The main challenge of applying machine learning method is to extract features that can represent the target analysis file [15] [16]. In 2016, Feng et al. [17] introduce a solution Genius. In order to transform original features of ACFG(attribute control flow graph) into feature vector, Genius uses bipartite graph matching algorithm to measure the similarity between specific ACFG. During comparison, the authors define codebook when embedding ACFG to matrix, which is very expensive and the runtime overhead increases linearly with the size of codebook. Xu et al. [18] rely on deep neural network to embed graph to matrix in their prototype Gemini. In the feature embedding process, Gemini applies structure2vec to get the features of nodes and their neighbours in ACFG like Genius. The features used by Genius and Gemini are statistical ones, which is scalable and efficient. However, it does not consider the semantic features of binary code and has high false positive rate. In Vulseeker, Gao et al. [19] generate LSFG(labeled semantic flow graph) to represent code feature and use DNN(deep neural network) to embed the features to vectors. Vulseeker outperforms Gemini in terms of accuracy. However, the LSFG extraction process is more complex and the features extraction method lacks semantic information like [18]. Zuo et al. [20] solve the code inclusion problems to compare similarity of two paths. They complete the feature embedding by neural network and path comparison by longest common subsequence algorithm(LCS). Inspired by NLP(natural language processing), Baldoni et al. [21] embed the instructions with word2vec model and optimize the hyperparameters using siamese structure. Redmond et al. [22] explore binary instruction embedding across architectures. They convert the binary code to intermediate language and recorded the input/output as signature for comparison. Alrabaee et al. [23] propose a free open-source software(FOSS) package reuse in binary files using Bayesian model to integrate the syntax, semantic and behavior features. However, this method is only applicable in x_86 architecture. They also propose a binary function clone detection method based on function execution trace(semantic integrated graph) [24]. They combine control flow graph, register flow graph and function call graph into semantic integrated graph and obtain graph trace as the comparison basis. Ding et al. [25] apply a vector representation method of assembly code to detect code clone. This method can learn the latent semantic information without preliminary knowledge of assembly code. However, this method can only works on x_86 platform. Bai et al. [26] propose a graph similarity comparison method directly based on node embedding instead of the widely used graph-level embedding. It is a general framework for similarity computation which can work with other models, and does not take semantic feature of each node into consideration.

Li et al. [27] aim to locate the vulnerability in basic block granularity by comparing the crashed execution trace and the normal trace with neural network. Their work is based on fuzzing and the execution is filtered by QEMU, which is time-consuming and has the low code coverage problem of dynamic execution. Hu et al. [28] also focus on the dynamic execution feature extraction to obtain the semantic feature of each target function. Their method has the same code coverage problem with Li et al. [27]. Zhang et al. [29] and Wang et al. [30] focus on the change between patched and unpatched code, and make similarity comparison in code snippet level instead of file level or function level comparison. The main contribution of Duan et al. [31] is that they extend the analysis scope from the CFG of a function to the program-wide CFG. They combine the NLP and TADW algorithm to obtain the semantic cross-function dependency feature. However, in the random walk, each basic block must

be included in at least two walks and each random walk has a length of 5 basic blocks, which is time consuming and has scalability problem.

From the discussion above, to apply the machine learning method, especially neural network to binary similarity, current work focuses on how to extract and embed features to represent binary code. However, most of current studies focus more on syntax features than semantic ones. For example, Gemini extracted 6 statistical features and 2 structural digital ones and Vulseeker extracts 8 statistical features when extracting the basic block granularity features, all of which are syntax features instead of semantic ones. But the syntax features cannot fully represent the code behavior because of the complexity of program design. The semantic feature can represent the code more effectively. But the finer the extraction granularity is, the more complex the extraction method is. Both the semantic extraction technology and the extraction granularity can reduce the analysis efficiency. In addition, to represent the structural dependency features of function in binary code, current work usually records the neighbour information of each basic block node within certain number of hops (e.g., 2 hops), which lacks the global information of the dependency relationship. To deal with the efficiency problem, we are committed to extract as much semantic information as possible under the premise of less impact on the analysis efficiency, and propose a preferred extraction method when extracting features in different granularities. We divide the feature extraction into three granularity from down to top: instruction granularity, basic block granularity and function granularity. When extracting features, we pay more attention to instruction granularity analysis, and extract the contextual and semantic information inside each basic block based on the skip-thoughts model in NLP. To solve the problem of lack of global dependency relationship information inside each function, we apply the graph neural network (GNN) to embed the structural features instead of the widely used method based on structure2vec model.

When comparing the similarity of two functions in binary files, we propose a two-channel encoding method based on NLP technique and GNNs to represent the semantic and structural characteristics of binary code. The process of semantic feature coding is divided into two stages: contextual semantic feature extraction and structural feature embedding stage. Finally, the similarity of two binary functions is calculated by trained model to judge whether two functions are similar or not. We have implemented a prototype system BEDetector (Bi-channel Encoder based binary Detector), which makes improvement on contextual semantic feature extraction and structural feature embedding process. We divided the detection process into offline and online phases, and discussed the performance of BEDetector in the two phases. During the offline encoding process to extract the semantic features and embed the structural features, the median line training time is 0.43 seconds on average for each function. And during online phases, BEDetector is compared with Gemini and Vulseeker by training time. BEDetector also has a higher accuracy than Gemini in terms of AUC and detection precision in real-world firmware detection with three CVE vulnerable functions including $ssl3\_get\_key\_exchange$ (CVE-2015-0204), $ssl3\_get\_new\_session\_ticket$ (CVE-2015-1791) and $udhcp\_get\_option$ (CVE-2018-20679). BEDetector found 95 out of 107 firmware files among top-50 detection results of CVE-2015-0204, with precision of 88.8%. The detection precision of CVE-2015-1791 and CVE-2018-20679 were 100% and 86.7%. In addition, the robustness of the system against structural difference is also discussed by case study.

Contributions. Summing up, the main contributions of our work are as follows:

- We divide the extraction method into three granularity and pay more attention to the instruction granularity. Inspired by the encoding-decoding model that can help extract the semantics of a sentence in NLP, we apply the skip-thoughts model to extract the feature in instruction granularity, remaining the semantic information of instructions in single basic block and the contextual relationship between basic blocks.
- The graph neural network (GNN) is implemented to represent the graph feature of each function. The structural features are extracted and selected by the graph autoencoder(GAE), remaining the structural characteristic of the whole graph during embedding.
- We have implemented the proof-of-concept BEDetector, which aims to extract finer semantic and global structural features without reducing the efficiency too much. The offline extraction and online training time is acceptable, and the accuracy of it is higher than the state-of-the-art model.
- When applied to the real-world firmware vulnerability detection, BEDetector can achieve a better performance. Its accuracy rate exceeds 86% of the top-50 candidate functions when detecting three CVE vulnerabilities including CVE-2015-0204, CVE-2015-1791 and CVE-2018-20679. Also the robustness of BEDetector is testified by case studies.

II. OVERVIEW

Given a query binary function f and its control flow graph(CFG) in which each vertex represents a basic block, we hope to find similar functions in binary code compiled in different architectures and optimization levels. Feature representation can be divided into three granularity: instruction granularity, basic block granularity and function granularity. The system workflow is shown in Fig1. In order to compare the similarity between query function f and target function f', BEDetector completes the comparison process in three steps: $1)Data\ preprocess$, $2)Feature\ Extraction$ and $3)Feature\ Integration\ and\ Similarity\ Calculation$. In data preprocess procedure, BEDetector firstly disassembles the binary function, extracts the control dependence relationship between basic blocks, and generates control flow

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access

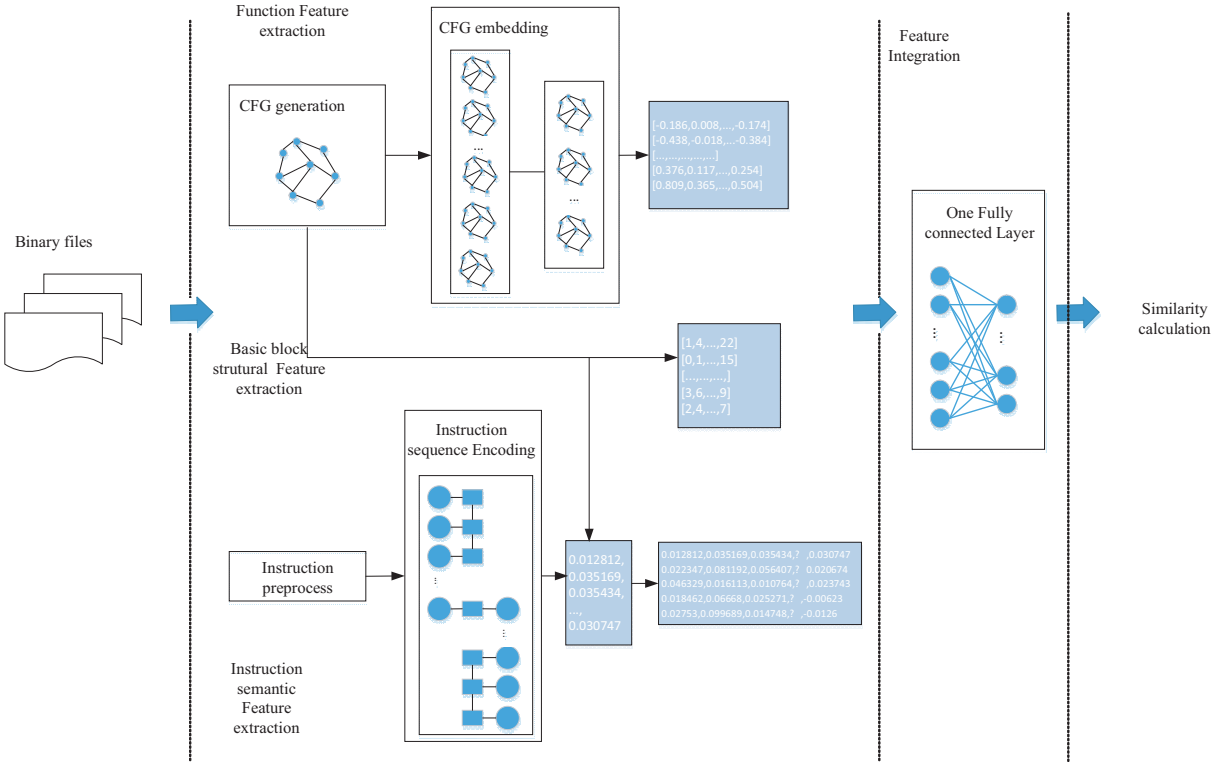


FIGURE 1. System workflow of BEDetector

graph(CFG). The feature extraction procedure is to obtain the features in different granularity. When extracting features of binary function, there are mainly two independent steps: *Instruction Semantic Feature Extraction* and *Structural Feature Extraction*. Then, in the feature integration and similarity calculation step, the feature matrix generated is integrated by a fully-connected neural layer. The fully-connected neural layer is contained in the siamese network and trained to adjust the hyperparameters. After training, given the query f and target function f', BEDetector can calculate the similarity score of the two functions to judge whether they are similar or not.

A. DATA PREPROCESS

Before extracting the features of binary code, we must make data preprocessing by disassembling the binary file, obtaining the assembly instruction sequence and function table. Meanwhile, the CFG of each function is constructed to reflect the control dependency of the basic block. In addition, when NLP technology is applied to code analysis, OOV(out of vocabulary) problems will inevitably occur. So we must deal with the OOV problem in data preprocess procedure and discuss this in detail in section III.

B. FEATURE EXTRACTION

The features extraction can be divided into instruction granularity, basic block granularity and function granularity from down to top. Different from previous work, the semantic contextual features in instruction granularity and the structural features in basic block granularity can be simultaneously extracted. Considering efficiency, we propose a granularity-preferred feature extraction method and pay more attention to the extraction in instruction granularity. In instruction granularity, the contextual and semantic feature of instruction sequence are extracted by adopting skip-thoughts model in natural language processing(NLP). In basic block granularity, only digital structural features of basic blocks are extracted. For each CFG, the structural feature can be presented by digital ones such as betweenness and offspring. In function granularity, the structural features of the whole graph is obtained by GNN(graph neural network). Graph autoencoder is one of GNN methods aiming to reconstruct the graph by the adjacent matrix and content of vertices of the origin graph. This method can learn the latent structural feature of graph. Each process of the three extraction granularity generates a matrix.

C. FEATURE INTEGRATION AND SIMILARITY CALCULATION

Siamese structure is used to calculate the similarity of two functions. In siamese structure, comparison branches containing the fully connected layer can integrate the feature matrices and calculate the similarity score between two functions.

In the rest of this paper, we will mainly discuss the latter two parts of the BEDetector. Section III extracts and embeds

instruction granularity feature based on skip-thoughts model. Section VI mainly discusses the feature representation of basic block and function granularity. The feature integration and similarity calculation is accomplished in section V. We evaluate BEDetector by experiments in section VI to prove its efficiency.

III. INSTRUCTION SEMANTIC FEATURE EXTRACTION

In the instruction granularity feature extraction, we propose an instruction2vector model inspired by the skip-thoughts model in NLP to get the contextual semantic information of instructions. The structure of paragraph in natural languages is somewhat like the structure of binary code. To adopt the skip-thoughts model in binary instruction embedding, the instruction sequence disassembled is normalized to deal with the out-of-vocabulary(OOV) problem in NLP. Then the instruction sequence is encoded by the skip-thoughts model to get a vector that contains the contextual information of each instruction.

A. BACKGROUND

Skip-thoughts model is a sentence vector model based on the skip-gram in natural language processing. The neural network structure of skip-thoughts model is one of the most commonly used encoder-decoder frameworks in machine translation. The unit of encoder-decoder is the same as GRU(Gated Recurrent Unit). It consists of update gate z_t and reset gate r_t, which is shown in Fig 2. The update gate represents the extent to which the previous state information is brought to the current one, and reset gate controls how much the information from previous status is transferred to current candidate set $\widetilde{h}_t$. GRU can solve the long term dependence problem in Recurrent Neural Network(RNN).

For NLP model, every word has semantic relationship with its context. A sentence also has a semantic relationship with its context sentences in a single paragraph, which is the motivation of skip-thoughts model [32]. It takes sentence sequence as input and contiguous text as corpus to train model. In this way, the skip-thoughts model can reconstruct the surrounding sentences of the encoded passage. This method can overcome the shortcomings of traditional natural language processing methods, only paying attention to the correlation between sentences and ignoring the complex semantic relations. When NLP is applied to binary software function analysis, the operands and operators in instructions can be regarded as words, assembly instructions as sentences, and basic blocks as paragraphs. Based on skip-thoughts model, we can not only get the semantic relationship between the instructions inside each basic block, but also the relationship between basic blocks. This feature extraction method can obtain more semantic information than the previous methods.

B. DATA PREPROCESS

There is also out-of-vocabulary(OOV) problem in binary analysis based on natural language processing. The operands in an instructions can be any type, from register to immediate.

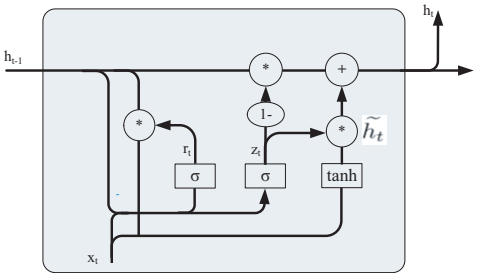


FIGURE 2. Unit of skip-thoughts model

In order to extract the semantic features of instructions, it is necessary to preprocess the operands(words in NLP) to solve the OOV problem. First, we deal with the immediate because it can be any value. We divide the immediate value into three different intervals. IMM1 represents an immediate value less than 0x100. Values between 0x101 and 0x200 are replaced by IMM2 with values greater than 0x200 by IMM. In addition, we reserve the registers(eax in X86, for example) and replace the base address with the symbol MEM. In data preprocessing, it is also necessary to extract the CFG graph of each function to get the control relationship between the basic blocks.

C. INSTRUCTION EMBEDDING

The skip-thoughts model includes encoder and decoder parts. We take the normalized instruction sequence in a basic block as the input to the skip-thoughts model. In the training of skip-thoughts model, the encoder regards instructions in a basic block as sentences in a paragraph and generates a vector. The decoder then rebuilds an instruction sequence to be used as the context sequence for these instructions. After training, we only use the encoder part of skip-thoughts model to generate vectors.

The encoder component is composed of GRU units, which takes the instructions in a basic block as the minimum input unit. Thus, each hidden state h_i^N represents the instruction sequence in one basic block. Just like [32], we encode the instruction sequence with the following formula:

$$z_t = \sigma(W_z \cdot [h_{t-1}, s_t]) \tag{1}$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, s_t]) \tag{2}$$

$$\widetilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, s_t]) \tag{3}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h}_t \tag{4}$$

where $\widetilde{h}_t$ is the proposed hidden state at time t ,and h_t is the hidden state that depend on h_{t-1} and $\widetilde{h}_t$. z_t is the update gate, r_t is the reset gate.

Here is an example of how the binary code are preprocessed and encoded. We take a code snippet in *ssl3_get_new_session_ticket* function in OpenSSL library. Fig 3 shows the data preprocess and instruction encoding procedure. Firstly, the instruction sequence is preprocessed to get the control dependence relationship. Instruction sequence

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access

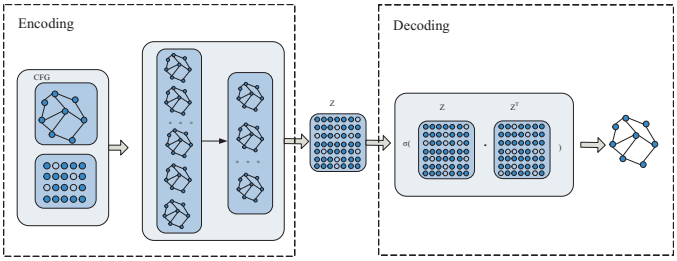


FIGURE 4. Graph encoder and decoder process

is divided into five basic blocks and the control dependence is recorded. Then, the skip-thoughts model is used to get the vector of each basic block, taking instruction sequence as input. After the encoding procedure, each basic block B_i is represented by a vector V_i, containing the semantic relationship among the instructions in it.

IV. STRUCTURAL FEATURE EXTRACTION

A. BASIC BLOCK FEATURE EXTRACTION

In the basic block granularity, only the digital features reflecting the vertex structure information in the graph(CFG) are extracted. Each basic block has two digital structural features, including betweenness and offspring. The betweenness of graph refers to the proportion of the number of paths passing through nodes to the total number of shortest paths, reflecting the influence of nodes on the whole graph. In the CFG of binary functions with basic blocks as vertices, the betweenness distribution can reflect the importance of basic blocks in the whole control flow graph. In addition, the offsprings of vertices are actually the number of connected basic blocks, which can also reflect the structural characteristics of graphs. In the basic block granularity, we only select betweenness and offspring for efficiency.

B. GRAPH EMBEDDING

In the function granularity feature extraction process, the main work is to maintain the structural feature of each function. As in the previous work, our analysis is based on control flow graph(CFG), where the vertices represent the basic blocks in the function with edges as control dependence relationship between basic blocks. Current work, such as Gemini and Vulseeker, applies a structure2vec method to integrate the neighborhood information of vertices through the adjacent matrix of CFG. All features are embedded by the adjacent matrix, recording information of neighbour within certain hops. In this section, we explore a graph neural network-based representation method for structure features of CFG, which is independent of contextual semantic feature extraction discussed in section III.

Traditional deep learning methods are successful in extracting features of the Euclidean spatial data, and the core assumption of deep learning algorithms is data sample independence. However, the vertices in a graph have relationship with their neighbours, which is not independent. Based on the idea of convolution network, cyclic network and depth automatic encoder, a graph neural network is designed to process graphics data. For CFG in a function, the relationship between vertices can be represented by GNN. Therefore, we propose a graph embedding method based on graph autoencoder(GAE) model to represent the structural features of CFG in functions. GAE model consists of two parts: encoder and decoder. The encoder uses graph convolutional network to embed function structure into matrix. The decoder uses the matrix generated by the encoder to reconstruct the graph. The workflow of GAE is shown in Fig 4. GAE model is trained by comparing the origin graph with the graph generated by decoder. In our graph embedding, only the encoder of training model is applied to embed CFG into the structural correlation matrix.

1) Encoding

The control flow graph represented by G=(V,E) is used as the input of encoder. For graph G, we need to record its adjacent matrix A and the degree matrix D. The vertex feature matrix X with dimension $n*d$ records the number of nodes n in the graph and number of features d. In the encoding process, the inference model is a two-layer graph convolutional network and can encode the graph into matrix Z.

$$Z = GCN(X, A) \tag{5}$$

$$GCN(X, A) = A' Relu(A' X W_0) W_1 \tag{6}$$

, where $A' = D^{-1/2} A D^{-1/2}$ is a symmetric normalized adjacency matrix. The generated matrix Z is the embedding matrix representing structural features of CFG.

2) Decoding

When training the GAE model, the decoder is to reconstruct a control flow graph. To ensure that the matrix Z can retain the structural features of the CFG, we reconstruct the adjacency matrix A', calculating the probability that there is an edge between two vertices. In the training process, by adjusting the parameters, the generated matrix A' is compared to matrix A until the reconstructed graph is closest to the origin CFG. After the training, each graph is represented by a matrix Z which records the relationship among basic blocks in one function.

$$A' = sigmoid(Z Z^T) \tag{7}$$

V. FEATURE INTEGRATION AND SIMILARITY CALCULATION

A. FEATURE INTEGRATION BASED ON MLP

We obtain different kinds of data recorded after the feature extraction procedure. In the instruction granularity, the feature matrix can reflect the semantic context information between instructions, while the feature matrix of function granularity represents the structural characteristics of the whole function. Here we use a naive method to concatenate the feature matrices of different granularity. Therefore, to take them as the input of similarity comparison, it is necessary to integrate the matrices and the digital variables.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access

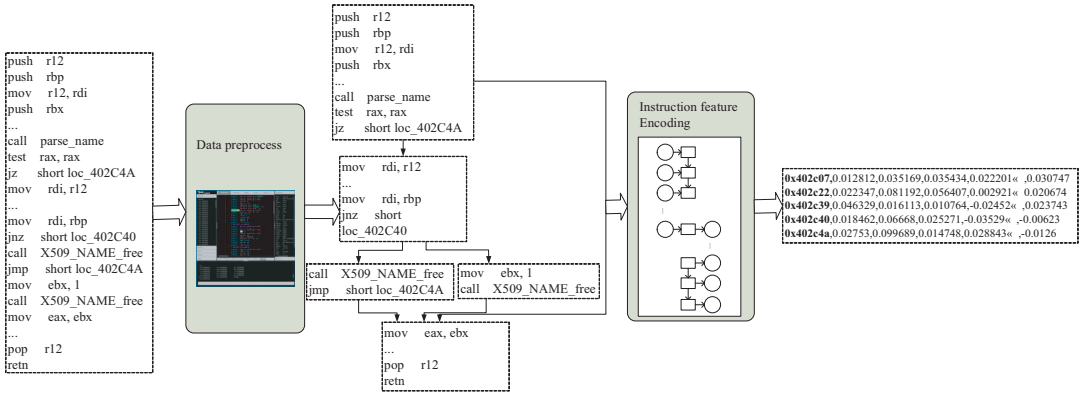


FIGURE 3. Data preprocess and instruction encoding example

1) Integration of Feature Matrices with Different Granularity

After the feature extraction procedure, we get the instruction granularity feature matrix V with dimension $N * d_1$ and matrix S with dimension $N * d_2$ generated in function granularity. Also the structural features in basic block granularity including betweenness and offspring are obtained. The matrix integration of different granularity is shown in Fig 5, the feature matrix V and digital structural features betweenness and offspring are concatenated first. After processed by ReLU activation function, matrix S is multiplied by weight P_1. At last, we can get the concatenated matrix including V, S, betweenness and offspring.

The left part of Fig 5 shows the concatenation of instruction granularity and basic block granularity features. Algorithm 1 describes the concatenation method with output matrix D, feature matrix V, betweenness and offspring as input. The method takes three inputs: feature vector V_i of basic block B_i, betweenness T_i and offspring P_i. Basic block B_i has a M dimension vector V_i, including elements v_i. V_i is concatenated to T_i and P_i. Since betweenness and offspring value have different influences on the representation of basic blocks, we give them different weights α and β. The output vector D of each basic block B_i has a $M + 2$ dimension. The output matrix M after integration is calculated by the equation below.

$$M = tanh(DW_1 + ReLu(SP_2)P_1) \tag{8}$$

where W_1, P_1, P_2 is the hyperparemeters, W_1 is a d_1*p dimensional weight matrix, P_1 is a p*p dimension weight matrix and P_2 a d_2*p dimension parameter matrix.

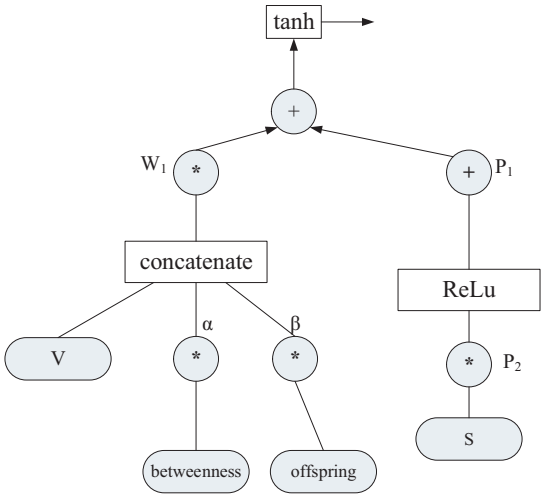


FIGURE 5. Matrix integration of different granularity

Algorithm 1 Feature integration in basic block granularity

/*V_i is the feature vector of basic block B_i in function F*/
/*T_i and P_i is the betweenness and offspring of B_i*/
/*N is the total number of basic blocks in function F.*/
Begin:
1: For vector V_i, let $v_i{}^1, ..., v_i{}^j, ..., v_i{}^M$ be the elements of V_i where M is the vector dimension.
2: Get the betweenness and offspring value T_i and P_i and assign them weight value α, β.
3: Concatenate value of V_i, αT_i and weighted value βP_i of betweenness and offspring.
4: Define $V_i{}' = (v_i{}^1, ..., v_i{}^M, \alpha T_i, \beta P_i)$.
5: D=$(V_1{}', V_2{}', ..., V_N{}')^T$.
6: return D
End

2) Similarity Model Training

Siamese network in Fig 6 is introduced to make similarity comparison of the two functions f_1 and f_2. Siamese network has two identical subnets sharing the same hyperparameters. For matrix D and the structural matrix S, we choose MLP to deal with the concatenation. For f_1 and f_2, subnets of siamese output matrix μ_1 and μ_2. The similarity of the two functions is calculated by the distance between the two matrices. During the training of siamese network, stochastic gradient descent method is used to optimize the hyperparameters in equation 8. According to the topological structure of the graph, the gradient parameters are calculated iteratively until the network performance is good. In the training process, the functional structure features based on CFG and contextual semantic feature are given different weights, so

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access

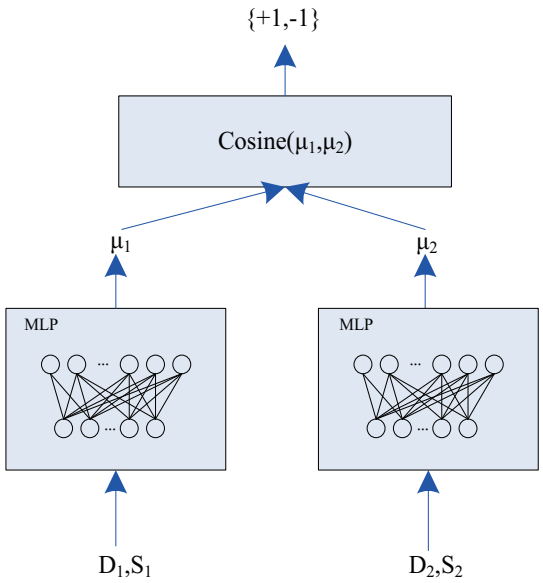


FIGURE 6. Siamese network to make feature embedding and similarity comparison

that they can influence the training process to a certain extent, instead of being integrated by the CFG using structure2vec. After training, given two functions, siamese can calculate the similarity score and judge whether they are similar.

VI. EVALUATION

A. IMPLEMENTATION AND SETUP

This section gives an empirical evaluation of our approach: Firstly, the embedding neural network model is trained with binary function dataset. We divide the embedding process into two stages: offline feature extraction and online training. The offline feature extraction is mainly to extract the features in different granularity, providing data and matrices as input to online training which is related the similarity calculation of two functions. Then, we use the validation data subset and real-world firmware dataset to verify the accuracy of the trained model. In the detection of real-world firmware dataset, the similarity score between real-world firmware functions and CVE vulnerable functions is calculated and sorted to obtain the candidate functions. We compare the detection results of BEDetector, BEDetector-S (ignoring the semantic instruction feature) and Gemini, and discuss the effectiveness of the detection method.

Our experiments were conducted on a server equipped with two GeForce 2080 GPU cards. During the training process, only one GPU card was used. In our paper, GPU is used to accelerate the training process. We prepared the representative, state-of-the-art, cross architecture vulnerability search technologies to establish our evaluation baseline: Gemini [18]. Also we implemented the BEDetector-S without extracting the semantic instruction features to verify the effectiveness of our extraction method.

Dataset In our evaluation, we collected three databases like Gemini: (1) Dataset I, which is used to train neural networks and evaluate the efficiency of trained model. (2) Dataset II, containing firmware images from Genius [17] that is also used by Gemini. (3)Vulnerability dataset, with CVE functions and related information.

In dataset I we use gcc 5.4.0 to compiled OpenSSL (v1.0.1f and v1.0.1u) and BusyBox(v1.27.2) at the optimization level of O0-O3 under x86, x64, MIPS32, MIPS64, ARM32 and ARM64 architecture, and all the functions containing multiple basic blocks in the compiled binary file are obtained. Dataset I contains 450,518 functions related to OpenSSL and BusyBox compiled in different optimization levels and different architectures. Dataset II contains real-world firmware images, with 4,017 images containing the OpenSSL library and 9,976 images containing Busybox tool. The total number of candidate functions in dataset II is 180,775. The vulnerability dataset includes the CVE information of vulnerability and the features of related vulnerability functions.

B. MODEL TRAINING

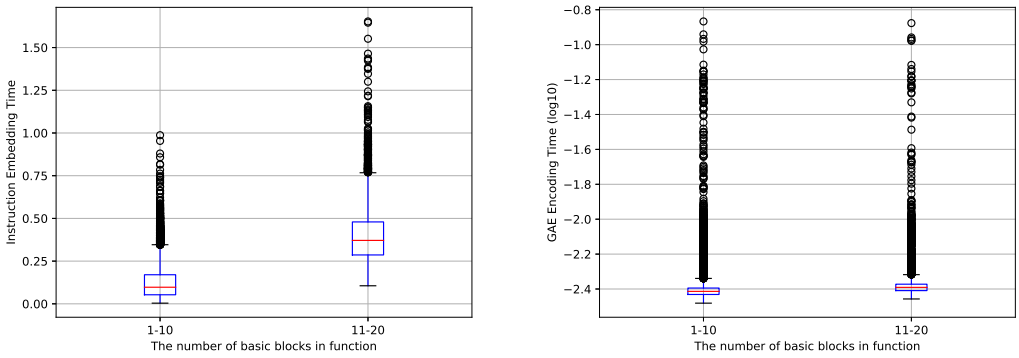
Firstly, we use functions in dataset I to generate three subsets: training set, test set and validation set. We randomly select function pairs from dataset I and labeled each pair. Then, function pairs compiled with the same source code at different architectures and optimization levels are chosen. For example, we get the function f_1 compiled at O0 and O1 optimization level, and name them with f^1_{O0} and f^1_{O1}. Therefore, the function pair $< f^1_{O0}, f^1_{O1} >$ has a ground true label 1. Function pairs labeled -1 are obtained by selecting different functions in dataset I. For functions f_1 and f_2 compiled at the O0 level, we can obtain the function pair $< f^1_{O0}, f^2_{O0} >$ with label -1. Totally, 100,000 pairs of functions labeled 1 and 100,000 pairs of functions labeled -1 make up the dataset. Of the 200,000 pairs of functions, the ratio of training set, test set and validation set is 18:1:1.

The training process includes offline features extraction and online training. In the process of offline feature extraction, three different granularity features are extracted and recorded. So for offline preparation, we discuss the efficiency in terms of time cost. Online training integrates the features of different granularity to generate a new matrix and adjusts the hyperparameters by the labeled function pairs in training dateset.

1) Offline Efficiency

To discuss the offline efficiency, we consider the instruction embedding time and graph encoding time, corresponding to instruction granularity and function granularity feature extraction. The basic block granularity feature extraction only obtains two digital features, and its time cost is not comparable with the other two granularity. The offline efficiency does not influence the overall efficiency as much as online does, because for the functions in the training dataset, offline features extraction is done one and for all. However, for efficiency, we want to discuss the time cost of offline feature extraction.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access



(a) Instruction encoding time of functions with less than 20 basic blocks

(b) Graph embedding time of functions with less than 20 basic blocks

FIGURE 7. Instruction encoding time of function granularity

Offline feature extraction involves not only the feature extraction in the process of neural network model training, but also the feature extraction of firmware for vulnerability detection. And the time cost of functions with different size should be considered. Among the 31,692 functions in BusyBox v1.27.2 in dataset I, there are 25,286 functions with less than 20 basic blocks, accounting for 79.8% of the total functions. However, in real world firmware binary files, the proportion of functions with basic blocks less than 20 is much larger. We randomly selected 311,514 functions from dataset II and got 298,269 ones with less than 20 basic blocks, accounting for 95.75% of the total ones. This is mainly due to limited memory with unnecessary code deleted. Therefore, we pay more attention to the time cost of functions with basic blocks less than 20.

During the instruction encoding process, we recorded the processing time of instruction by skip-thoughts model. The processing time of each function depends on the number of basic blocks in it and the number of instructions in the basic block. However, we focus on the processing time of each function for the subsequent comparison is based on function granularity, and record the instruction encoding time of the function with less than 20 basic blocks in Fig7 a). For functions with less than 10 basic blocks, the median encoding time is 0.084 seconds. It takes 0.35 median line seconds to process functions with 11 to 20 basic blocks, and 0.55 median line seconds for functions with 21 to 30 basic blocks. Functions with basic blocks number more than 50 need 1.87 median line seconds.

Besides instruction embedding time, graph encoding time should be considered. Fig 7 b) describes the graph encoding time of functions with less than 20 basic blocks. The median time cost of functions with less than 10 basic blocks was 0.082 seconds, and that of 11 to 20 was 0.95 seconds. In addition, the median time for functions with 21 to 30 basic blocks was 2.35 seconds, and that of functions 31 to 50 was 3.49 seconds. We also recorded time cost of functions with more than 51 basic blocks, with a median time of 4.9 seconds.

Although the instruction embedding time and graph encoding time is relatively long for functions with more than 20 basic blocks, they account for a small proportion, especially in the real world firmware. To estimate the time cost of offline training, we calculated a binary file containing 500 functions. The average time cost of instruction embedding procedure was 78.7 seconds, and that of graph encoding was 135.8 seconds. So the average offline time cost of each function was 0.43 seconds. The offline coding runs only once. The offline time of larger functions will not have a great impact on the detection efficiency.

2) Online Training Efficiency

The online training process of BEDetector is to train the network, and the similarity score between functions can be obtained by adjusting the hyperparameters, which is somewhat like Gemini and Vulseeker. So the efficiency of online training is evaluated by comparing the training time cost of BEDetector neural model with Gemini and Vulseeker. Vulseeker is based on Gemini, but it takes more time than Gemini by considering both CFG and DFG in the training process. Here we set the training iteration to 500. The time cost in each iteration of the three models is shown in Fig 8. It can be seen that the training time of BEDetector is slightly longer than that of Gemini, and each iteration takes 1 to 2 seconds. However, the cost of Vulseeker is 5 to 7 seconds because it considers both data flow dependency and control flow dependency in the embedding process. However, BEDetector extracts feature matrix containing semantic context information and structural matrix based on CFG, which needs more time than Gemini. Different from Gemini and Vulseeker, its embedding method is simply a concatenation of the two matrices with weights in basic block granularity and it does not take too long time, making the training time close to Gemini. From the time training comparison analysis, the time cost of BEDetector is acceptable.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3064687, IEEE Access

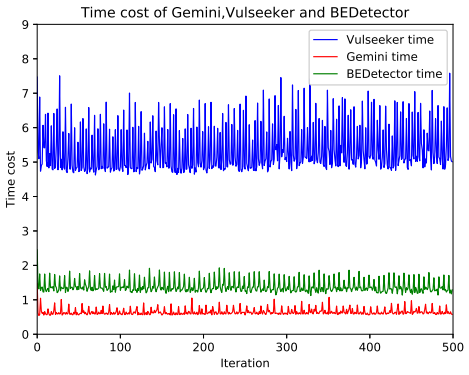


FIGURE 8. Time cost of each training iteration of BEDetector and Gemini

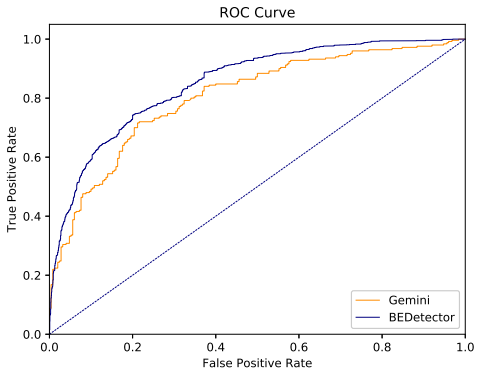


FIGURE 9. ROC curves of BEDetector and Gemini

C. PERFORMANCE OF NEURAL MODEL

1) Performance on Validation Dataset

The performance of BEDetector neural model is evaluated by the AUC. The AUC value and receiver operating characteristic(ROC) are firstly calculated in the validation dataset. Receiver operating characteristic (ROC) curve reflects the relationship between sensitivity and specificity. It shows how the relationship between $recall$ and $accuracy$ changes when the threshold identified as a positive example changes. The ROC curve divides the figure into two parts. The area below the curve is called AUC (area under curve), which is used to represent the prediction accuracy. The higher the AUC value is, the larger the area under the curve and the higher the prediction accuracy. If the curve is closer to the upper left corner(the larger Y with smaller X), the prediction accuracy is higher.

We compare the accuracy of BEDetector and Gemini by validation dataset, which consists of 10,000 pairs of functions. For comparison, we set the epoch number of both models to 200 and embedding size to 64. The ROC of BEDetector and Gemini is shown in Fig 9. It can be seen from Fig 9 that BEDetector has the curve which is closer to the upper left corner than the other one, meaning a higher prediction accuracy. The AUC of BEDetector and Gemini was 0.9 and 0.82 respectively. The AUC of Gemini is different from [18]. This may be because our training dataset is larger than Gemini and contains different function library.

2) Performance Comparison on Subset

Evaluation metrics In this comparison experiment, our goal is to detect the binary codes compiled by the same source code in different architectures and compilation levels. To find out whether the code compiled by the same source is judged to be the same. We use the top-1 rate and the mean reciprocal rank(MRR) in the recommender system. We randomly selected 500 functions out of 450,518 ones in dataset I and obtained binary functions compiled in O0-O3 optimization level and under x86 and ARM architecture.

Compared Methods Our model has two main components: semantic-aware modeling(skip-thoughts in instruction granularity) and structural-aware modeling(graph autoencoder). Different experiments are conducted to find out the effect of each components. Firstly, we choose GCN, DeepWalk and Node2vec to compare the structural-aware modeling of our method. Then to compare the semantic-aware modeling, Word2vec combined with graph auto-encoder(GAE) is chosen. Word2vec [33] is a fundamental method to learn word embeddings. When applying Word2vec to represent the semantic feature of instruction, we get the token embedding by Word2vec and calculate the sum of token embeddings as the block embeddings. We also compare our model with structure2vec(used by Gemini and Vulseeker).

Overall Performance For functions in the selected subset, we calculate the similarity scoring of functions compiled in the same optimization level but in different architectures. Then the ranking of functions in target file is recorded and used to get the mean reciprocal rank(MRR). Also we pay attention to the top-1 ranking of the analyzed function.

Table 1 shows the MRR and top-1 rate of different models. The first block shows the results of structure-aware modeling, and the second shows the comparison result of semantic-related feature extraction models. Of the three graph embedding models GCN, DeepWalk and Node2vec in the first block in Table 1, GCN and Node2vec perform better than DeepWalk. This is because Node2vec makes improvements on DeepWalk which randomly selects the next node. Node2vec takes the relationship among nodes into consideration. The GCN model uses both adjacency matrix and simple feature matrix, making a better performance. Meanwhile, semantic-aware modeling in the second block outperforms the only structure related modeling. Thus, semantic features play an important role in the representation of binary code.

When comparing the semantic-aware modeling, our model outperforms Gemini(structure2vec) and Word2vec+GAE model because the skip-thoughts model extracts more semantic features than the other two models. Structure2vec combines structural and statistical features when embedding features, and the comparison result proved that semantic features

TABLE 1. Similarity score rank of different optimization level

Model	MRR/top-1-O0	MRR/top-1-O1	MRR/top-1-O2	MRR/top-1-O3
Node2vec [34]	0.4129/0.3984	0.4018/0.3805	0.3407/0.3128	0.2715/0.2527
DeepWalk [35]	0.2539/0.2152	0.2304/0.2080	0.1596/0.1345	0.1499/0.1187
GCN [36]	0.4241/0.4003	0.4090/0.3929	0.2720/0.2510	0.2598/0.2399
Structure2vec [18]	0.6359/0.5848	0.6290/0.5920	0.5530/0.5030	0.5428/0.4835
Word2vec+GAE	0.7526/0.7028	0.7335/0.6985	0.6892/0.6425	0.6638/0.6358
Our model	0.7904/0.7215	0.7729/0.7101	0.7452/0.7039	0.6829/0.6521

express code better than statistical features. Word2vec+GAE model considers both semantic and structural features. However, Word2vec is a word embedding model. When it is applied to binary code embedding, it generates vectors to represent instructions. In order to represent basic block, we have to process the vectors of its instructions. For example, the vector of instructions can be summed and averaged to get the vector representation of the basic block. This kind of methods usually can not fully represent the semantic relationship between instructions. On the other hand, the skip-thoughts model is a sentence embedding model. Therefore, when applied to binary file representation, it can represent the semantic relationship between instructions in a basic block. So with the same structural model GAE, our method performs better than Word2vec in presenting the semantic features of binary code. In addition, we found that all models perform best at O0 level optimization, and the detection ability decreases from O0 to O3. For example, our model has a MRR of 0.7904 in O0 optimization level and 0.6829 at O3 optimization level. Other models also face the same problem as our model, that is, the impact of optimized code structure on detection results. The reason for different detection results is the degree and content of optimization, which has different influence on representation of binary code structure. We examined the differences between the optimization levels of GCC and found that there is no optimization when compiling codes at the O0 level. O1 level is mainly about the optimization of code branches, constants and expressions, which is a partial optimization. O2 performs almost all optimizations without time and space trade-offs, such as rearrangement of basic blocks and rearrangement of instructions. O3 further optimizes O2 by turning on the $-finline-functions$, $-fweb$, $-frename-registers$ and $-funswitch-loops$ options, including inline and loop optimization. The O2 and O3 optimization level make more modification on the compiled binary code, making the detection result gap between optimization levels O1 and O2 the largest. Our model can narrow the gap between the two optimization levels compared to the other five models. The O1 and O2 MRRs of our model were 0.7729 and 0.7452 respectively, while those of other models such as GCN were 0.4090 and 0.2720. This also shows that our model is more robust to the optimal structural detection.

D. REAL-WORLD FIRMWARE VULNERABILITY DETECTION

In this section, we will further discuss the performance of BEDetector model in real-world similarity detection. Here we apply the model to vulnerability detection and calculate the similarity score between the functions in real-world firmware files and the vulnerability functions.

For comparison, we recorded the vulnerability detection result of BEDetector and Gemini in real-world firmware. Given a CVE vulnerability function, BEDetector can detect whether the target firmware is affected by a certain vulnerability by calculating the similarity score between target function in firmware file and the vulnerable function. OpenSSL and BusyBox are widely used third-party library codes in firmware. We pay attention to their vulnerabilities and detect whether these vulnerabilities exist in real-world firmware files. Features of both the vulnerable functions and the functions in firmware files are extracted and embedded into the matrices. Finally, the similarity score is calculated by the trained neural model and sorted. The higher the similarity score of the functions, the more likely they are to be the vulnerable ones. To evaluate the detection accuracy, we compare the scoring ranks of vulnerability function in firmware of BEDetector with Gemini.

When firmware manufacturers apply third-party code to their devices, they usually delete unnecessary functions due to limited memory. In manual analysis, we also found out that the firmware binaries only contain partial components in OpenSSL or BusyBox. Therefore, not all BusyBox and OpenSSL vulnerable functions exist in the real-world firmware, and vulnerable functions such as $MDC2\_Update$ of CVE-2016-6303 rarely appear in firmware files. We randomly selected 400 firmware files from dataset II and focused on three CVE vulnerabilities to determine whether these vulnerabilities exist in firmware files or not.

The three selected vulnerability functions are ssl3_get_key_exchange (CVE-2015-0204), ssl3_get_new_session_ticket (CVE-2015-1791) and udhcp_get_option (CVE-2018-20679). The first two functions affect OpenSSL and the last one affects BusyBox files. In CVE-2015-0204, the ssl3_get_key_exchange function allows remote SSL servers to perform downgrade attacks and facilitate brute-force decryption by providing a weak ephemeral RSA key in a noncompliant role. CVE-2015-1791 is a race condition security vulnerability, the exploration of function ssl3_get_new_session_ticket in file $ssl/s2\_clnt.c$ can cause

DOS(Deny of Service) attack. CVE-2018-20679 is an out of bounds read vulnerability in udhcp components (consumed by the DHCP server, client, and relay), allowing a remote attacker to leak sensitive information from the stack by sending a crafted DHCP message.

By manual analysis, we found out that of the 400 selected firmware files, 69 files are affected by CVE-2015-1791 ($ssl3\_get\_new\_session\_ticket$), 107 files affected by CVE-2015-0204 ($ssl3\_get\_key\_exchange$), and 30 files are affected by CVE-2018-20679 ($udhcp\_get\_option$). BEDetector can calculate the similarity score between functions in firmware files and the vulnerable functions, and sort the score to determine whether the vulnerabilities exist.

We made the comparison experiment using BEDetector and Gemini to compare their efficiency in the vulnerability detection process in real-world firmware. To verify the advantage of feature selection in BEDetector, we deleted the instruction semantic features and implemented BEDetector-S for comparison. BEDetector-S takes the statistical information as basic block granularity features just like Gemini, which is the number of string constants, numeric constants, transfer instructions, calls, instructions, arithmetic instructions, offspring and betweenness. Still, BEDetector-S takes the same graph neural model as BEDetector. So we can judge the influence of semantic features among the instructions in single basic block. And the structural features can be testified by comparing BEDetector-S and Gemini. We compare the top-5, top-10 and top-50 detection number of three tools when detecting the CVE vulnerability functions in firmware dataset containing 400 files. Of the 69 files affected by CVE-2015-1791, 107 files affected by CVE-2015-0204 and 30 files affected by CVE-2018-20679, the top-5, top-10 and top-50 detection result of three tools is shown in Fig 10. BEDetector performs better than the other two tools in terms of detection result. Of the total 107 files containing vulnerability CVE-2015-0204, the top-5 detection number is 39, 21 and 20 for BEDetector, BEDetector-S and Gemini respectively, while the top-10 detection number is 77, 55 and 63, as shown in Fig 10 b). When detecting the other two vulnerability functions, BEDetector has better performance than BEDetector-S and Gemini as shown in Fig 10 a) and c).

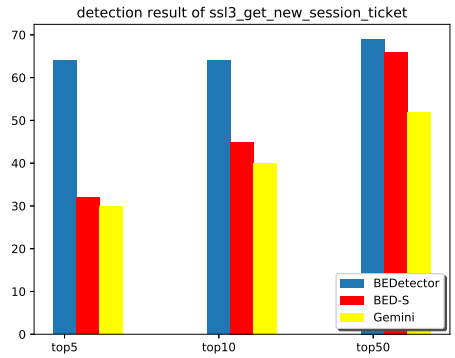
Besides, we compared BEDetector, BEDetector-S and Gemini in terms of detection rate. Here the detection rate is defined as the ratio of the binary number of detected vulnerability functions to the number of binaries containing vulnerability functions. So of the 107 firmware files containing CVE-2015-0204 vulnerability, BEDetector detects there are 95 files containing the vulnerability when ranking top-50 candidate functions and the detection rate is 88.8%. The detection result is shown in Table 2. We firstly compared the detection efficiency of BEDetector and BEDetector-S. In the detection of function ssl3_get_new_session_ticket, the top-50 detection rate of BEDetector-S is 95.7%, which is relatively high. However, it is lower than BEDetector, which is 100%. And from Table 2, the detection rate of BEDetector is the highest, with a more than 86.7% rate in top-50 detection process. BEDetector performs better than BEDetector-S just because when extracting features in basic block and instruction granularity, BEDetector focuses on the semantic features inside basic blocks which can better represent the features of binary code. Also we compared BEDetector-S with Gemini to testify the effectiveness of graph embedding method, because the only difference between the two tools is the graph model they use. BEDetector-S applies the same graph encoding model as BEDetector which embeds the features by graph autoencoder, while Gemini apply the structure2vec model and embedding by iterating the neighbour features of each node in the graph. From Table 2 it can be seen the overall performance of BEDetector-S is better than Gemini except for the top-10 ssl3_get_key_exchange detection rate. Of the 69 files containing ssl3_get_new_session_ticket vulnerability function, BEDetector-S can verify 45 files ranking top-10 while Gemini verified 40 files. And the top-50 ranking detection number of BEDetector-S is 66, achieving a 95.7% detection rate compared with a 75.4% detection rate of Gemini. As discussed above, the main difference between BEDetector-S and Gemini is the graph embedding model and the better performance of BEDetector-S shows that the graph autoencoder method is relatively more effective than the structure2vec method.

E. CASE STUDY

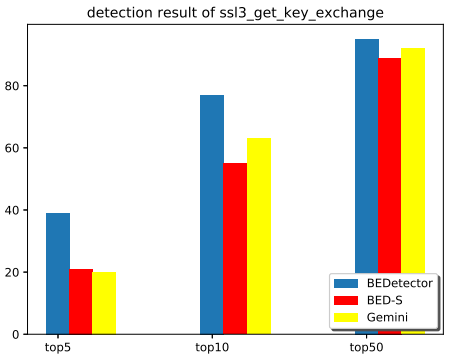
In this section, we explore case study to further verify the robustness of BEDetector. Robustness here refers to the structure difference between the two functions when the BEDetector detects that they are similar. If the system can detect functions with larger structural differences, it will not be affected by structural features too much and is robust to the structural difference. We choose vulnerability function $ssl3\_get\_new\_session\_ticket$ and the firmware files containing the function to test the robustness of BEDetector.

As discussed in the previous subsection, CVE-2015-1791 is the race condition in the $ssl3\_get\_new\_session\_ticket$ function in $ssl/s3\_clnt.c$ in OpenSSL. If a multi-threaded client receives a NewSessionTicket object when attempting to reuse a previous ticket, a race condition may occur, resulting in a double free of ticket data. Fix code is shown in Fig 11, where the line with symbol + represents the code added to fix the vulnerability. Therefore, before the code is fixed, the pointer p and d are assigned by the $init\_msg$ in NewSessionTicket s and then the type conversion is implemented by function $n2l$ and $n2s$, which is in line 2240, 2273 and 2274 in Fig 11. And the added codes from line 2241 to 2271 fix the vulnerability by handling the multi-threaded client to delete the old session.

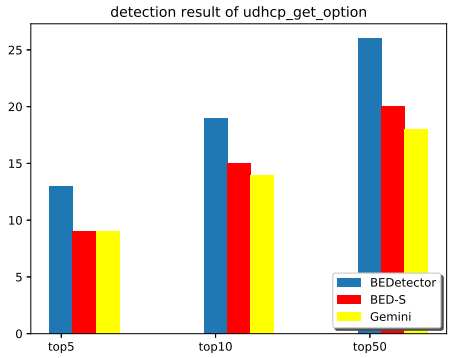
When detecting this vulnerable function in the real-world firmware files, we calculate the similarity score of each function of the firmware. The 1st similarity score ranking firmware $wg-webflash$(wg302v1) was chosen and analyzed. And the vulnerability of $wg-webflash$ is confirmed by manual analysis. Although the similarity score is the 1st when comparing the similarity of vulnerability function and



(a) Detected firmware file number of ssl3_get_new_session_ticket function(CVE-2015-1791)



(b) Detected firmware file number of ssl3_get_key_exchange function(CVE-2015-0204)



(c) Detected firmware file number of udhcp_get_option function(CVE-2018-20679)

FIGURE 10. Detected real-world firmware file number containing CVE vulnerability

TABLE 2. Similarity Score Ranking in Real-world Firmware

	ssl3_get_new_session_ticket			ssl3_get_key_exchange			udhcp_get_option		
tool	Top-5 rate	Top-10 rate	Top-50 rate	Top-5 rate	Top-10 rate	Top-50 rate	Top-5 rate	Top-10 rate	Top-50 rate
BEDetector	92.8%	92.8%	100%	36.4%	72.0%	88.8%	43.3%	63.3%	86.7%
BEDetector-S	46.4%	65.2%	95.7%	19.6%	51.4%	83.2%	30%	50%	66.7%
Gemini	43.5%	58.0%	75.4%	18.7%	58.9%	86.0%	30%	46/7%	60%

```
2240      p=d=(unsigned char *)s->init_msg;
2241 +
2242 +    if(s->session->session_id_length>0){
2243 +        int i=s->session_ctx->session_cache_mode;
2244 +        SSL_SESSION *new_sess;
...
2269 +    SSL_SESSION_free(s->session);
2270 +    s->session=new_sess;
2271 +    }
2272 +
2273      n2l(p,s->session->tlsext_tick_lifetime_hint);
2274      n2s(p,ticklen)
```

FIGURE 11. Fixed code related to CVE-2015-1791 vulnerability

function in $wg-webflash$, there are different structural CFG in them. Through manual analysis, we found out that the firmware file is compiled under ARM architecture. The control flow graph of $wg-webflash$ and the baseline OpenSSL v1.0.1f compiled in ARM architecture are shown in Fig 12 (a and (b. Each basic block in the CFGs contains a sequence of instructions. To make the CFG graph clearer, we blank the contents of each basic block in the CFG graph, and only record the corresponding instructions contained in the entry basic block of the function. It can be seen that the two CFGs are not exactly the same. BEDetector still can find the target function in $wg-webflash$ by the different structural features. This is mainly because that the neural model takes other features such as the semantic contextual features to

reduce the influence of structure on the final comparison results.

VII. CONCLUSION

Binary function similarity detection technology is the basis of vulnerability detection, and has drawn a lot of attention. In this paper, we investigate a two-channel feature encoder method based on natural language processing and graph neural network to make binary similarity detection. Aiming to solve the limitation of current work, which is the lack of contextual semantic and locality of the structural feature, we divide the feature extraction into three granularity and pay more attention to instruction granularity, extracting the contextual semantic feature inspired by the skip-thoughts model. Also the graph autoencoder network is applied to get the global structural features of function. To tolerate the structural influence on the similarity detection result, we implement the contextual semantic feature and graph embedding independently. Finally we adopt the widely-used siamese network to calculate the similarity scores between functions and sort them to judge whether functions are similar. We implemented a proof-of-concept BEDetector and compared it with state-of-the-art Gemini in the validation dataset, subset and real-world firmware files. Besides, in the experiment we made case study to testify the robustness of BEDetector. It shows our method can still detect the functions with different structural feature due to the compiler optimization. This vulnerability detection technology can also be applied to other fields based on similarity detection, such as code plagiarism and malware detection. However, although the top-50 detection rate is above 86% in real-world firmware. The top-1 detection rate is relatively low, indicating the limitation of static based method. In later research, we will focus on the lightweight dynamic feature extraction of binary function that can represents its execution behavior.

ACKNOWLEDGMENT

We thank JianGao for sharing his code of Vulseeker in github, which inspired us a lot when implementing BEDetector. This work is supported by the National Key Research and Development Program of China (No. 2017YFB0802900)

REFERENCES

[1] Synopsys 2020 open source security and risk analysis report. [EB/OL]. https://www.synopsys.com/software-integrity/resources/analyst-reports/2020-open-source-security-risk-analysis.html?cmp=pr-sig Accessed September 4, 2020.

[2] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR), 50(4):56, 2017.

[3] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. Sysevr: a framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756, 2018.

[4] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. Discovering software vulnerabilities using data-flow analysis and machine learning. In Proceedings of the 13th International Conference on Availability, Reliability and Security, page 6. acm, 2018.

[5] Boris Chernis and Rakesh Verma. Machine learning methods for software vulnerability detection. In Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, pages 31–39. ACM, 2018.

[6] Michael Pradel and Koushik Sen. Deep learning to find bugs. TU Darmstadt, Department of Computer Science, 2017.

[7] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. arXiv preprint arXiv:1803.04497, 2018.

[8] Halvar Flake. Structural comparison of executable objects. In DIMVA, volume 46, pages 161–173. Citeseer, 2004.

[9] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In International Conference on Information and Communications Security, pages 238–255. Springer, 2008.

[10] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In International Conference on Information Security and Cryptology, pages 92–109. Springer, 2012.

[11] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic'juice'. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, page 5. ACM, 2013.

[12] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In Acm Sigplan Notices, volume 49, pages 349–360. ACM, 2014.

[13] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In 2015 IEEE Symposium on Security and Privacy, pages 709–724. IEEE, 2015.

[14] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: fingerprinting binary functions to support automated analysis of code executables. In IFIP International Conference on ICT Systems Security and Privacy Protection, pages 341–355. Springer, 2017.

[15] Luo Fulin, Huang Hong, Duan Yule, Liu Jiamin, and Liao Yinghua. Local geometric structure feature for dimensionality reduction of hyperspectral imagery. Remote Sensing, 9(8):790, 2017.

[16] Guangyao Shi, Hong Huang, and Lihua Wang. Unsupervised dimensionality reduction for hyperspectral imagery via local geometric structure feature learning. IEEE Geoscience and Remote Sensing Letters, PP(99):1–5, 2019.

[17] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 480–491. ACM, 2016.

[18] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 363–376. ACM, 2017.

[19] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 896–899. ACM, 2018.

[20] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706, 2018.

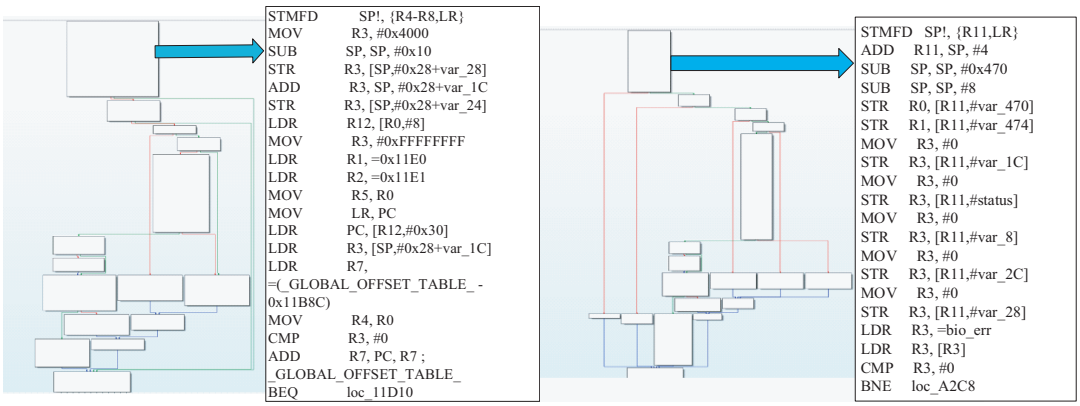
[21] Roberto Baldoni, Giuseppe Antonio Di Luna, Luca Massarelli, Fabio Petroni, and Leonardo Querzoni. Unsupervised features extraction for binary similarity using graph embedding neural networks. arXiv preprint arXiv:1810.09683, 2018.

[22] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. arXiv preprint arXiv:1812.09652, 2018.

[23] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: a resilient and efficient system for identifying foss functions in malware binaries. ACM Transactions on Privacy and Security (TOPS), 21(2):1–34, 2018.

[24] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused.

[25] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against



(a) CFG of vulnerable function in wg-webflash (b) CFG of vulnerable function in OpenSSL

FIGURE 12. Structural comparion of $ssl3\_get\_new\_session\_ticket$ in wg-webflash and official OpenSSL 1.0.1f compiled in ARM architecture

code obfuscation and compiler optimization. In 2019 IEEE Symposium on Security and Privacy (SP), pages 472–489. IEEE, 2019.

[26] Yunsheng Bai, Hao Ding, Ken Gu, Yizhou Sun, and Wei Wang. Learning-based efficient graph similarity computation via multi-scale convolutional set matching. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 3219–3226, 2020.

[27] Zhang C. Feng C. Li, R. H. Locating vulnerability in binaries using deep neural networks. Ieee Access, 7:134660–134676, 2019.

[28] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. A semantics-based hybrid approach on binary code similarity comparison. IEEE Transactions on Software Engineering, pages 1–1, 2019.

[29] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 887–902, 2018.

[30] Shi-Chao Wang, Chu-Lei Liu, Yao Li, and Wei-Yang Xu. Semdiff: Finding semtic differences in binary programs based on angr. In ITM Web of Conferences, volume 12, page 03029. EDP Sciences, 2017.

[31] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. 01 2020.

[32] Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In Advances in neural information processing systems, pages 3294–3302, 2015.

[33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013.

[34] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016.

[35] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, pages 701–710, New York, NY, USA, 2014. ACM.

[36] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In International Conference on Learning Representations (ICLR), 2017.