IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 28, 2018, accepted January 11, 2019, date of publication January 17, 2019, date of current version February 8, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2893733

Staged Method of Code Similarity Analysis for Firmware Vulnerability Detection

YISEN WANG, JIANJING SHEN, JIAN LIN, AND RUI LOU
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
Corresponding author: Yisen Wang (851067568@qq.com)

This work was supported by the National Natural Science Foundation of China under Grant 61802431.

ABSTRACT The security situation of the Internet of Things (IoT) is more serious than ever, and there is an urgent need to detect and patch device vulnerability rapidly. With the astronomical numbers of IoT devices, it is very difficult to execute regular security inspections. Existing vulnerability detection technology based on simple feature matching cannot reach high accuracy to detect firmware vulnerabilities while using a control flow graph matching directly has proven to be too expensive. To address the problem of accurate and efficient, we present a method of staged firmware vulnerability detection based on code similarity. The first stage, function embedding based on neural network is used to analyze the similarities among functions, and large-scale firmware security inspection can be achieved efficiently. The second stage, the similarity among function local call flow graphs is calculated for fine-grained firmware security analysis, and this stage can improve the accuracy of vulnerability detection. We compared our method with state-of-the-art approaches, and the experimental results demonstrate that our method is more accurate. The average retraining time of our method is 1 h, and the real-world firmware vulnerability detection experiment of our method demonstrates that the true positive rate of the top 30 is as high as 86%.

INDEX TERMS Firmware security, network embedding, code similarity detection, firmware vulnerability, neural network.

I. INTRODUCTION

Function similarity analysis technology plays an important role in many fields, such as software plagiarism detection [1] and malicious code detection [2]. Function similarity technology has been applied in the security of the Internet of things (IoT). With the aim of extending the connectivity beyond standard computers to each physical device, the IoT has added another dimension to Internet development. However, with the endless emergence and ubiquitous deployment of IoT devices, a significant number of potential targets are also exposed to the outside world. IoT devices have become one of the most popular targets for hackers and one of the easiest to attack, as proven by the increasing attacking events targeting IoT devices in recent years [3]. To make matters worse, owing to rigorous limitations on costs and time-to-market, IoT vendors tend to reuse easy-to-obtain yet unsafe software modules in their device firmware [4], and vulnerabilities in certain software modules may affect a large number of IoT devices, as even their application scenarios exhibit significant differences [5], [6]. Therefore, for vendors and network security corporations, it is important not only to identify the vulnerabilities of a single IoT device, but also to evaluate the hazards of the vulnerabilities to other IoT devices as extensively as possible [7].

However, as opposed to software security analysis in the field of PC, almost all firmware of IoT equipment is not open source [8], [9]; moreover, the processor architectures and operating systems of IoT equipment are diverse. Firmware cannot be analyzed by methods used on PC software, such as symbolic execution or dynamic analysis. Although the accuracy of symbolic execution is high, the performance overhead is substantially greater [10]. Thus, symbolic execution is not suitable for firmware analysis, as the size of the firmware is often large. IoT equipment includes numerous peripherals, which make it hard to build an integrated environment for firmware; thus, firmware dynamic analysis is difficult work [11], [12]. Among the existing firmware security analysis methods, feature-based analysis is more effective, and depends on the quality of extracted features [13]. To the best of our knowledge, none of these existing methods have considered the invoke relations among functions; however, the function invoke relation determines whether

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

or not bugs can be triggered. Furthermore, we believe that function invoke relations are more robust to heterogeneity of the processor architecture and compiler, which is critical to function similarity analysis. Gemini [14] offers great efficiency, but we argue that Gemini has not taken into account the bug trigger mechanism [15], as it ignores the invocation relations among functions, which may have an impact on vulnerability similarity accuracy. All of these methods play a critical role in firmware analysis, but they share at least one of the following shortcomings. (i) Incomplete features: neither string nor control flow graph (CFG) features can represent a function effectively, as they ignore the invoke relations among functions, which is critical to bug triggering. (ii) High overheads: the operation of exiting methods is expensive, such as tree edit distance calculation [16] and codebook generation [17], so it is difficult to implement large-scale firmware security inspection. (iii) Poor extensibility: the retraining of existing models requires days or even weeks, which makes it impossible to adapt to a new training dataset effectively. However, Gemini, which can be retained quickly, is an exception.

We propose a staged firmware function similarity analysis method, which includes coarse-grained and fine-grained analysis of firmware binary code. Section *II* introduces the background of the function similarity analysis. An overview of our method is presented in section *III*. The details of our method are provided in sections *IV*, *V*, and *VI*. Section *IV* introduces the manner in which to select function features, section *V* discusses the generation of firmware function embedding, and section *VI* introduces the function local call graph (LCG) similarity analysis. An evaluation of our method is presented in section *VII*, related work is discussed in section *VIII*, and section *IX* concludes the paper.

Our main contributions are as follows.

- We propose a novel staged firmware vulnerability detection method. The first stage uses a residual network to embed the function features into the high-dimensional space for efficient analysis of function similarity, while the second stage uses the hierarchical weighted bipartite graph to calculate the function LCG similarity. The combination of the two stages provides higher similarity analysis accuracy compared with existing methods.
- We design a method using the genetic algorithm to optimize function features. In addition to CFG features, the call flow graph features of the function are considered, and the genetic algorithm is applied to obtain an optimal weighted feature combination, which improves the function similarity accuracy.
- We implement a prototype system and verify its effectiveness. The experiments demonstrate that the prototype yields an AUC of 0.981, which is higher than the AUC of 0.971 achieved by the state-of-the-art methods, and obtains the best result of 26 true positive samples in the top 30 suspicious vulnerable functions in real-world testing. Furthermore, it can complete the retraining process in 1 h and can thus be reconstructed rapidly, according to new datasets and purposes.

II. BACKGROUND

A. FUNCTION SIMILARITY ANALYSIS

Feature matching is often used in vulnerability detection, providing high efficiency and accuracy. The main concept of feature matching is to extract the vulnerability feature, and then verify whether the target binary has any code similar thereto. For example, [13] used strings as feature to detect the firmware security. Function similarity analysis is one of the methods of feature matching, and extracts the syntactic features of functions to compare the differences among them.

Existing function similarity analysis methods are mainly suitable for functions with the same processer architecture; however, multiple architectures exist for firmware functions. The architecture of the same product firmware may differ from model to model; thus, significant syntactic diversity exists among firmware functions. However, there is little semantic difference in homologous functions, because semantics primarily express functional attributes, which would not vary with a change in architecture. By comparing a large number of CFGs and call flow graphs of cross-platform binaries compiled from the same source code, it can be found that the graph structure is exactly the same. The call flow graph of the function represents the invoke relation among functions; if the invocation order changes, so does the program action. Therefore, regardless of whether the binary architecture, operating system, compiler, and compile compilation options are the same, the call flow graph of the programs should be the same if they are compiled from the same source code. The CFG describes the internal characteristics of the function, and expresses the jump path of the basic block. Owing to the differences in the registers, address offsets, and instruction opcode among architectures, the order of the basic blocks of the homologous functions will change slightly, but the overall structure of the CFG will not vary substantially. Therefore, the call flow graph and CFG are appropriate features for the cross-architecture similarity analysis of functions. Reference [10] proposed a method for cross-architecture function similarity analysis based on maximum common subgraph matching, which achieved high accuracy. However, the experiments demonstrated that direct graph matching is very expensive, and is not suitable for large-scale security detection of firmware functions.

B. NETWORK REPRESENTATION LEARNING OF FUNCTION

Network representation learning (NRL) [18] uses low-dimensional, dense, and real-valued vectors to represent network nodes; that is, for mapping to k-dimensional hidden space. NRL introduces the concept of embedding [19], [20], which can be represented as illustrated in Figure 1, in which node u is mapped to d-dimensional space by a kernel function and transformed into a vector. NRL offers two main advantages: 1. nodes of networks with similar structures will have

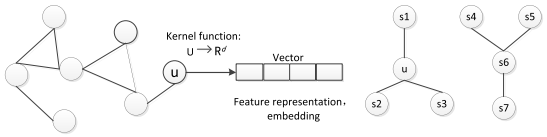


FIGURE 1. Node embedding schematic.

similar embedding; and 2. homophilic network nodes will have similar embedding. For example, nodes u and $s6$ have similar embedding in Figure 1. The property of NRL can be used to analyze the similarity of firmware functions, because homologous functions should have similar embedding, even if the processor architecture, operating system, and compiler differ. Furthermore, the computational efficiency of vector similarity is higher than that of graph similarity.

III. APPROACH OVERVIEW

A. OVERVIEW

To realize the similarity analysis among firmware functions more efficiently and obtain the potential relations among firmware function features, which can be embedded in high-dimensional space. Following embedding, a firmware function is represented as a numerical vector, which can be used directly to analyze function similarity without accessing the original functions. Any action on the function can achieve the same effect on the function embedding vector by means of certain operations, but vector analysis is more efficient than raw function analysis. An overview of our method is presented in Figure 2. We implement a staged firmware function similarity analysis method, which can not only satisfy large-scale analysis, but also ensure accuracy. Moreover, two databases are created to contain the embeddings of the firmware and vulnerability functions, respectively: FirmwareDB and VulnerabilityDB, which can be used to detect the firmware security rapidly.

In Figure 2, the inputs are firmware binary code and vulnerability functions. The commercial disassembly tool IDA Pro [21] was used to extract the function features, including intra-function CFG features and function call graph features. To represent the function more accurately, a feature selection procedure, the genetic algorithm, was used to filter out the low-impact and redundant features and generate a weighted function feature combination (section *IV*). A neural network was designed to embed the function in the high-dimensional space, where the function could be represented as a numerical vector. Finally, the two-stage code similarity analysis was performed.

B. TWO STAGES ANALYSIS

First stage: Function embedding similarity analysis. This was achieved by calculating the cosine angle of two embeddings to obtain the similarity, which is suitable for large-scale firmware security inspection. Further details on function embedding similarity analysis are provided in section *V*.

Second stage: LCG similarity analysis. The LCGs of the firmware and vulnerability functions form a multi-layer weighted bipartite graph, and the similarity of nodes on both sides of the graph could be calculated using the algorithm during the first stage. The invocations among functions were considered in this stage, which could improve the accuracy of the vulnerability similarity analysis but sacrifice efficiency. Further details on LCG similarity are provided in section *VI*.

The reason for performing two stages of code similarity analysis is to satisfy different analysis demands. If one only wishes to determine whether the firmware contains suspicious vulnerability functions; that is, whether or not the firmware is secure, stage one can satisfy this requirement, as the firmware is probably insecure if it contains many suspicious vulnerability functions and the similarity scores are high. If an accurate result is desired, especially to determine whether the function is really a vulnerability function, both stages one and two should be performed. The LCG contains function invoke relations that are important for bug triggering. Furthermore, the function invoke relations are more robust to heterogeneity of the architecture than CFG features.

IV. FEATURE SELECTION

A. FIRMWARE FUNCTION FEATURE EXTRACTION

Multiple processor architectures and operating systems exist in firmware; for example, the architecture of NETGEAR firmware may be ARM or MIPS, among others; therefore, we should select features that are weakly related to architecture to analyze the firmware function similarity [14], [16], [17]. The same functions under different architectures have similar semantics [10], such as function invoke relations and function structures. Therefore, the CFG features and certain call flow graph features of the function are selected as the features for analyzing the function similarity. The selected features can be described as $FuncFeature = ACFG + ACG = \{V, E, \psi\}$, where $ACFG$ represents the attributes of the CFG, ACG represents the attributes of the call flow graph, V represents the basic block in the CFG, E represents the paths among basic blocks, and ψ represents the function features.

Three feature types are selected from the CFG and call flow graph of the function, namely statistical, structure, and function call features. The statistical feature mainly includes the number and proportion of each instruction type, the number of basic blocks and edges of the CFG, constant information, and string information. The structure feature expresses the CFG structure, including the basic block depth, relations among basic blocks, and CFG breadth and depth. The function call feature represents the call flow graph feature, including the numbers of calls and times called. Finally, 50 function features are selected. However, embedding all features directly requires excessive time; thus, to improve the efficiency of function embedding, certain unimportant features should be filtered out.

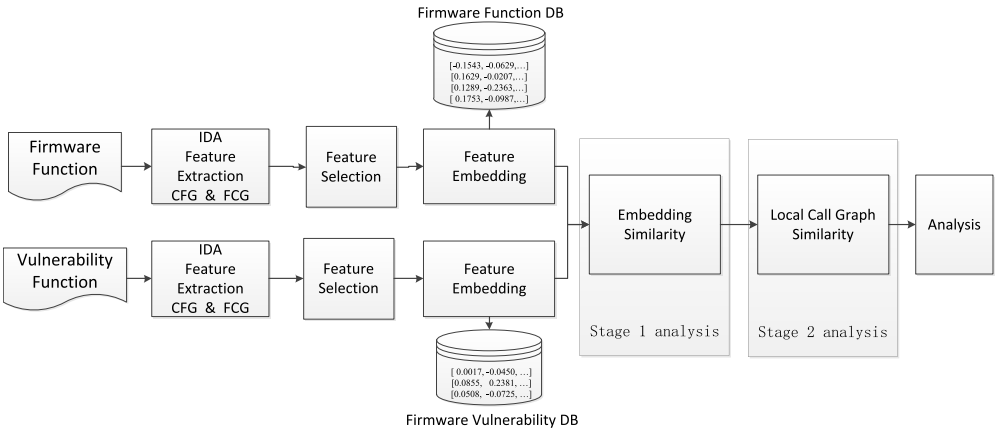


FIGURE 2. Approach overview.

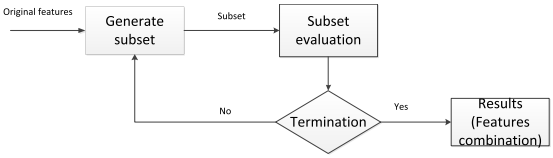


FIGURE 3. Feature selection process.

B. GENETIC ALGORITHM OF FEATURE SELECTION

Numerous feature selection algorithms are available [22]; the process of feature selection is illustrated in Figure 3. Firstly, the feature subset is generated from the original features, following which the feature subset is evaluated by the evaluation function. If the evaluation meets the criteria, the selection is complete; if not, the feature subset is updated and evaluated again.

The genetic algorithm is considered to be more suitable for the above requirements than others [23], and provides a method to search for the optimal solution by simulating the natural evolution process. To make the feature combination suitable for firmware function similarity analysis, sample selection is very important. OpenSSL is common in firmware, so OpenSSL with different architectures (ARM, MIPS, X86) were selected as samples, and IDA was used to extract features from them. The function is represented by the selected features and transformed into the embedding; thus, the accuracy of the function embedding can be used to evaluate the selected features. Section V presents the details of function embedding and how to obtain it. The Euclidean metric between two function embeddings was selected as the objective function, which can be expressed as

$$\begin{aligned} F_{obj}(func) &= \rho(func_{arc1}, func_{arc2}) \\ &= (\sum\nolimits_{i\in[1,n]} |\mu(func_{arc1})_i - \mu(func_{arc2})_i|^2)^{1\backslash 2} \end{aligned} \tag{1}$$

where $func_{arc}$ represents the function $func$ under arc architecture, $\mu(func)$ represents the embedding of function $func$, and n represents the number of vectors of the function embedding. Genetic algorithms generally evolve in the direction in which the fitness value increases; thus, the fitness and objective functions should be converted appropriately. The Euclidean metric between two function embeddings is a non-zero positive number; therefore, the derivative of the objective function can be used as the fitness function, which can be represented as

$$F_{fit}(func) = \frac{M}{F_{obj}(func)} \tag{2}$$

where M is a coefficient less than 1 and is used to prevent the fitness value from becoming too large. The selection operation in the genetic algorithm is used to filter out the unimportant features, and the selection principle is as follows: The best feature is directly entered into the next generation, while the remaining features are converted into selection probabilities according to fitness values. A large fitness value corresponds to a high selection probability, while a small fitness value corresponds to a low selection probability. The selection process of firmware function features can be represented by Algorithm 1.

In Algorithm 1, the inputs are samples and raw features, where arc may be ARM, MIPS or X86. A sample is a function pair and a batch contains 500 samples. In line 1, the weights and the maximal fitness are initialized to 0. The maximum generation T is set to 50. In line 4, each feature is evaluated by the fitness function and the average fitness value $AverVal_i$ is obtained; $func(v_i)$ is the i^{th} feature of function $func$, and N is the number of samples in one batch. The feature weight w_i is calculated according to the fitness value in line 5. The feature combination is used to calculate the fitness value of the function in line 6. If $AverVal(t)$ is greater than $MaxFitness$, update the value of $MaxFitness$ in line 7. In line 11, the selection operation is performed to select the genetic features. Line 13 can obtain the feature combination with the maximum fitness value.

Algorithm 1 Function Feature Selection Algorithm

Input: Samples $P = (p_i, i = 1, 2, ..., 3000)$
$p = (func_{arc1}, func_{arc2})$
raw feature set $V = \{v_i, I = 1, 2, ..., 50\}$
Output: Selected feature combination *FinalSet* and M

```
Initialize MaxFitness = 0, M, T
for t in T do
    Generate samples randomly p(t)
    Val_i = F_fit(func(v_i)) and
    AverVal_i = (1/N) Σ_{j∈[1,N]} Val_j
    w_i(t) = f(AverVal_i) and
    W_(t) = {w_i(t), i = 1, 2,..., 50}
    Val(t) = F_fit(func(W · V)) and
    AverVal(t) = (1/N) Σ_{j∈[1,N]} Val(t)
    if MaxFitness < AverVal(t) then
        MaxFitness = AverVal(t)
        FinalW = W_(t) and FinalSet = feat_set(t)
    end if
    Select p(t + 1) from p(t)
end for
return FinalSet and FinalW
```

C. FEATURE SELECTION FOR FIRMWARE FUNCTION SIMILARITY ANALYSIS

Algorithm 1 filters out the low-impact features from the 50-dimensional features and obtains a 10-dimensional feature combination with weights. The feature combination is displayed in Table 1.

TABLE 1. Selected feature combination.

Feature type	Feature name	Weight
Function call feature	No. of calls	0.0922
	No. of called	0.1053
Statistical feature	No. of instructions	0.0866
	No. of degrees (indegree, outdegree)	0.0561
	No. of arithmetic instructions	0.0940
	No. of nodes	0.1237
	No. of edges	0.1323
Structure feature	Base block depth	0.0620
	Base block average depth	0.1401
	Base block maximum depth	0.1076

The experiment demonstrated that two feature types were filtered out by the genetic algorithm: low-impact and redundant features. Low-impact features mainly include instruction distribution and CFG branch structure features; for example, the instruction ratio and number of offspring. The reason that these features are not important is that the instruction distributions of different functions exhibit few differences. The redundant features are mainly the same functional features or may be derived from one another; for example, the number of basic blocks and number of transfer instructions of a function are both important, but the number of transfer instructions can be inferred from the number of basic blocks, so one of these can be discarded.

Following feature selection, the staged code similarity analysis is performed.

V. THE FIRST STAGE - FIRMWARE FUNCTION EMBEDDING

A. FUNCTION EMBEDDING GENERATION

A common approach for learning deep characters from structured data is the kernel method, and the kernel function is designed according to the demand. The kernel function is used to transform structured data into feature representation, which can be used to achieve the same function effect without accessing the raw data. Inspired by [24], graph theory was used to embed the function features into high-dimensional space, and a multidimensional numerical vector $\mu(f)$ was generated to represent the function. Furthermore, $\mu(f)$ can be used to calculate the function similarity by means of formula $Sim(f_1, f_2) = Sim(\mu(f_1), \mu(f_2))$.

Mapping function features to Hilbert space is equivalent to mapping to infinite space features [25]; that is,

$$\mu_X := E_X[\Phi(X)] = \int_{\chi} \Phi(x)p(x)dx; P \to F, \chi = R^d \quad (3)$$

where X represents a set of variables, x represents an instance of X, $\Phi(x)$ represents the feature mapping, and $p(x)$ represents the probability of x. Following mapping, the variables are embedded into high-dimensional space and exhibit abundant representational abilities. The mapping in Hilbert space has an injective property; that is, $p(x)$ and $\mu(x)$ exhibit one-to-one correspondence. Any operation on $p(x)$ can obtain the same result by performing certain operations on $\mu(x)$. We can design a kernel function that embeds the function features in the high-dimensional space. As opposed to using expensive function feature matching, high-dimensional representation is efficient for function similarity analysis.

The literature [24] proposed a method for feature embedding based on graph inference, which finally achieved the goal of image classification. The kernel function used was:

$$\mu_i = \Gamma(x_i, \{\mu_j\}_{j \in N(i)}, \{x_j\}_{j \in N(i)}) \quad (4)$$

where $N_{(i)}$ represents the adjacent nodes to node i. The relations of nodes and edges in the graph are mostly nonlinear, so Γ is a nonlinear function. The kernel function is used to obtain the embedding of image features through iteration, and then compared with the real label of the image. However, this method is mainly applied to classification and is not applicable to firmware function similarity analysis. Instead of obtaining the classification results, function embedding is sufficient for analyzing function similarity.

Equation 4 is modified to fit into the function embedding generation; that is

$$\mu_i = F(v_i, \mu) = reLu(W_1 v_i + \sigma(W_2 \sum_{j \in N(i)} \mu_j + W_3 \sum_{j \in N(i)} v_j)) \quad (5)$$

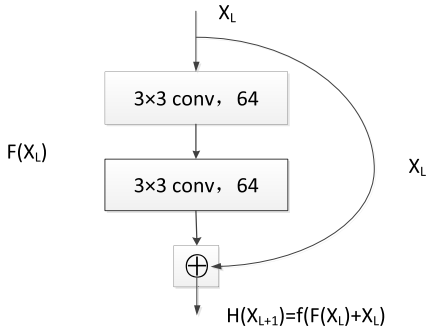


FIGURE 4. Residual network.

where v_i is the d-dimensional eigenvector of the basic block node i in the CFG of the function, $W = \{W_1, W_2, W_3\}$ is a matrix with d rows, and σ is an n-layer neural network. To learn the deep relations of functions, the deep residual network [26] is used to generate function embeddings, and the structure is illustrated in Figure 4. The function embedding can be implemented by Algorithm 2, and the data preprocessing includes function feature extraction and an invalid function filter. Functions with less than five basic blocks are specified as invalid, because a function with fewer basic blocks has a lower probability of vulnerability.

Algorithm 2 Firmware Function Embedding Algorithm

Input: Parameter $W = \{W_1, W_2, W_3\}$, *FuncFeature*
Output: Function embedding $\mu(f)$
1: Initialize $\mu_b^0 = 0$, for all $b \in V$, T
2: Data pre−processing Input_Message
3: **for** $t = 1$ to T **do**
4: **for** $b \in V$ **do**
5: $z_b = \sum_{j \in N(b)} \mu_j^{t-1}$
6: $\mu_b^t = reLu(W_1\, v_b + \sigma(W_2\, z_b + W_3 \sum_{j \in N(b)} v_j))$
7: **end for**
8: **end for**{fixed point equation update}
9: **return** $\mu(f) := \sum_{b \in V}\ \mu_b^T$

In Algorithm 2, the matrix parameter W is obtained by means of training. The training datasets are OpenSSL and BusyBox binaries under different architectures (ARM, MIPS, and X86), which are compiled by different compilers. The formats of the training dataset are $\{fi, fi', 1\}$ and $\{fi, fj, -1\}$, where fi and fi' are labeled as similar functions, indicated by 1, and fj and fi are labeled as non-similar functions, denoted by -1. Non-similar functions refer to functions with different source code, while similar functions refer to functions with the same source code and different architectures, compilers or compile optimization, which can be represented as

$$Sim\_function := \{same\_source \&\& (diff\_architecture \,|diff\_compilier\,|diff\_optimize)\} \quad (6)$$

FuncFeature is the feature of function f, which can also be represented as $\{V,\ E, \psi\}$. The basic block embeddings are initialized to 0 for every basic block in line 1. The variate T in line 3 is the number of iterations, which represent the distance a vertex propagates. A larger T means the vertex travels further, and μ_b^t obtains additional information. Lines 5 and 6 are equal to Equation 5, which can determine the basic block embedding of the t^{th} iteration. The final function embedding $\mu(f)$ is obtained in line 9.

B. PARAMETER TRAINING

In the training process, the loss function is expressed as the absolute difference between the function similarity and label. The purpose of training is to reduce the loss function value to the minimum. Our loss function is expressed in Equation 7, where W is the required parameter.

$$Loss(Sim, label) = \min_W \sum\nolimits_N (Sim(\mu\,(f_i)\,, \mu\,(f_j)) - label)^2 \quad (7)$$

The function similarity is calculated by the angle between cosines.

$$Sim(f_i, f_j) = \cos(\mu(f_i), \mu(f_j)) = \frac{<\mu(f_i), \mu(f_j)>}{||\mu(f_i)|| \bullet ||\mu(f_j)||} \quad (8)$$

The stochastic gradient descent algorithm was used to optimize the parameters. First, the function embedding $fi \rightarrow \mu(fi)$ was obtained; then, the function similarity was calculated according to the embeddings $Sim(\mu(fi), \mu(fj))$. By minimizing the square loss, the optimal value of the parameter W was learned. During training, each function will have different features and embeddings, but all functions share the parameter W. Algorithm 3 was used to implement the training process.

Algorithm 3 Parameter Training Algorithm

Input: Parameter
$DataSet = \{< f_i, f_j >, label | i \in [1, N], j \in [1, N]\}$,
loss function $Loss(Sim, label)$
Output: Parameter W
1: Initialize W randomly, T
2: Data pre−processing
3: **for** $t = 1$ to T **do**
4: //Get samples from *DataSet* randomly
$DataSet(t) = \{< f_i, f_j >, label\}$
5: Get the embeddings of functions by algorithm 2 with W^t: $\mu(f)$
6: //Calculate the function similarity
$Sim_t = Sim(\mu(f_i), \mu(f_j))$
7: Update $S^t = S^{t-1} + \lambda_t \nabla_{S^{t-1}} Loss(Sim_t, Lable)$
8: **end for**
9: **return** W^T

This section has discussed the implementation of the function similarity analysis, which is the first stage of our method. Firmware function embeddings and vulnerability function embeddings are stored in FirmwareDB and VulnerabilityDB, respectively. When firmware is required to conduct security

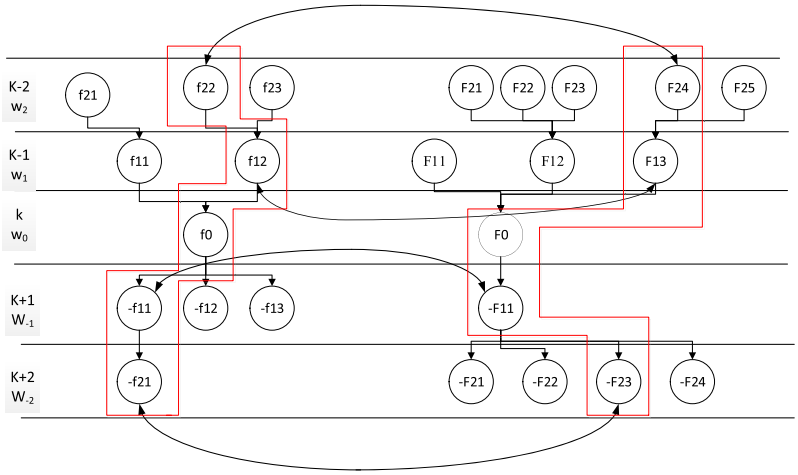


FIGURE 5. LCG similarity analysis.

detection, it should first be disassembled, and the firmware functions are transformed into function embeddings. Thereafter, the locality-sensitive hashing (LSH) algorithm is used to search VulnerabilityDB to determine whether there are any embeddings of vulnerability functions similar to the firmware function embeddings. Because the efficiency of LSH is high [17], [27], it is possible to implement large-scale firmware security inspection.

VI. THE SECOND STAGE - LCG SIMILARITY ANALYSIS

A. FUNCTION INVOKE RELATIONS

Section *V* discussed the implementation of the firmware function embedding similarity analysis, which can be used to determine whether the firmware contains suspicious vulnerability functions efficiently. However, bug triggering is not the effect of a single function, but the result of a series of function invocations. Therefore, the similarity of a single function cannot infer the vulnerability similarity.

This section analyzes the similarity of LCG, which considers the invoke relations among functions, to improve the firmware vulnerability detection accuracy. A function call flow graph is a structured representation of the program, which can accurately describe the invoke relations among functions therein. The call flow graph of the function can be obtained by static analysis, where the nodes in the graph represent functions and the edges represent the invoke relations among functions. The call flow graph exhibits strong robustness to the program confusion and distortion of the program; therefore, it offers the following advantages in vulnerability similarity analysis compared with the CFG.

1) It considers the invoke relations among functions.
2) It is more robust to the heterogeneity of processor architectures, operating systems, and compilers than CFG features, and critical to function similarity analysis.
3) It is more consistent with the vulnerability triggering mechanism.

B. LCG SIMILARITY ANALYSIS ALGORITHM

The function LCG can be represented as $G_{local} = \{\cdots, f_{i-2}, f_{i-1}, f_i, f_{i+1}, f_{i+2}, \cdots\}^N_{1<i<N}$, where i represents the layer of function f, N represents the number of layers of the LCG, and $F_{i+k}(-N/2 < k < N/2)$ represents the set of functions in the $i+k$ layer that have an invoke relation with f_i. The LCG similarity between the firmware function and vulnerability function $Sim(G_f, G_{vul})$ is analyzed to determine whether function f exhibits vulnerability. The similarity analysis of the function LCG is illustrated in Figure 5.

The two LCGs to be compared constitute a multilevel bipartite graph [28], as indicated in Figure 4. On the left is the LCG of the firmware function, and on the right is the LCG of the vulnerability function. Different layers have different weights, and in principle, the weight is higher closer to the central function. A greedy extension algorithm is used to extend the LCG layer, as indicated in Algorithm 4. The left graph is represented by g and the right graph is represented by g_{vul}, and the functions in layer k are the firmware and vulnerability functions. The LCG similarity between two graphs can be calculated by Equation 9, where g represents the LCG, W_k is the weight of layer k, and fg_k represents the functions of g in layer k.

$$Sim(g, g_{vul}) = \sum\nolimits_{i\in[-M,M]} W_i Sim(fg_{k+i}, fg_{vul(k+i)}) \quad (9)$$

In Algorithm 4, the weight W is obtained experimentally, and fg and fg_{vul} are the functions in g and g_{vul}. In line 1, the layers M are initialized to 2; in theory, additional layers of the LCG mean that more features are included, and the similarity accuracy rate should be higher. However, the increase in layers will lead to the accumulation of errors and a sacrifice of efficiency. Therefore, the scale of the LCG is not as large as possible, and the optimal number of layers should be obtained by training. The similarity of functions in layer k is first calculated based on the method in section ćõ, and the

Algorithm 4 LCG Similarity Analysis Algorithm

Input: $g, g_{vul}, fg_k, fg_{vul(k)}$
Output: Similarity of two LCGs $Sim(LCGg, LCGg')$
1: Initialize layers $M = 2$, $W = \{w_2, w_1, w_0, w_{-1}, w_{-2}\}$
2: Use g and g_{vul} to build multilevel bipartite graph
3: Calculate function similarity of layer k based on method in section V, $Sim(fg_k, fg_{vul(k)})$
4: **for** $m = 1$ to M **do**
5: $FuncPair_{\max} = Sim_{\max}(fg_{(k+m)i}, fg_{vul(k+m)j})$ $|i \le ng_{(k+m)}, j \le ng_{vul(k+m)}$
6: Extend to next layer based on function pair $(fg_{(k+m)i'}, fg_{vul(k+m)j'})$
7: **end for**
8: //determine layer $k-$ function similarity
9: **for** $m = -1$ to $-M$ step -1 **do**
10: Repeat step 7 and step 8
11: **end for**
12: //Calculate similarity weighted sum of all layers $SimSum = \sum_{l\in[-M,M]} W_l \underset{\max}{Sim}(fg_{(k+l)}, fg_{vul(k+l)})$
13: **return** $Sim(LCG_g, LCG_{g'}) = SimSum$

similarity can be represented by $Sim(fg_k, fg_{vul(k)})$. Then, this is extended to layer $k+1$ in line 4. The similarity between the functions in g and those in g_{vul} is calculated to obtain the most similar function pair $FuncPair_{max}$ in line 5, where $n_{g_{k+m}}$ represents the number of functions in layer $k+m$ of graph g. In line 6, this is extended to layer $k+2$. Only the functions that have invoke relations with $(fg_{(k+m)i}, fg_{vul(k+m)j})$ in layer $k+2$ are considered. That is, i' represents the node that has an invoke relation with i and j' represents the node that has an invoke relation with j. The same operations as in layer $k+$ are performed in layer $k-$ to obtain the function similarity in layer $k-$ in line 9. The similarity weighted sum of all layers is calculated by Equation 9 in line 12.

This section has discussed the implementation of the LCG similarity analysis, which is the second stage of our method. This stage improves the vulnerability similarity accuracy at the cost of efficiency and can be used for fine-grained firmware security detection.

VII. IMPLEMENTATION AND EVALUATION

We implement a prototype by using the method presented in this paper, and evaluate the prototype in three aspects : accuracy, efficiency, and utility. The evaluation experiments were implemented with 4.3 GHz, 128 GB memory, 2 TB SSD, and a single GPU server.

A. DATA PREPARATION

An IDA plug-in was written to extract function features and three datasets were prepared: the training, firmware, and vulnerability datasets.

Dataset 1, training dataset: To adapt the firmware cross-architecture attribute, it is necessary to select libraries that are widely available in multiple architectures as the training dataset. Thus, the open source libraries OpenSSL1.1.0f and BusyBox1.27.2 were selected as the original training datasets. The libraries were compiled into three architectures (ARM, MIPS, and X86) by the compilers gcc6.4 and clang3.9 at the three optimization levels O1, O2, and O3. The disassembler IDA7.0 was used to extract the function features. Homologous functions under different architectures, compilers, and compile optimization are marked as 1, while nonhomologous functions are marked as -1.

Dataset 2, firmware dataset: Crawlers were used to access the firmware of major manufacturers, including router firmware: NETGEAR, D-Link, TP-Link, and HUAWEI, among others. Printer firmware such as HP, Canon, and Epson, as well as APEX, provided a substantial amount of firmware. In total, 13750 firmware images were collected.

Dataset 3, vulnerability dataset: The list of CVE numbers of vulnerability was collected from the official website and 50 vulnerability functions were obtained.

The default setting for generating embedding was the same as in reference [24], with an embedding size p = 64 and embedding depth n = 1. The model ran for T = 3 iterations, and the learning rate was 0.0001.

B. HYPERPARAMETERS

This section discusses the experiments conducted on the embedding size, embedding depth, and iteration times. Our initial hyperparameter settings were the same as in [24]: embedding size: 64, embedding depth: 1, and number of iterations: 3. The hyperparameters of Gemini are: embedding size: 64, embedding depth: 2, and number of iterations: 5. The same dataset was used to compare the code similarity under our hyperparameters and those of Gemini.

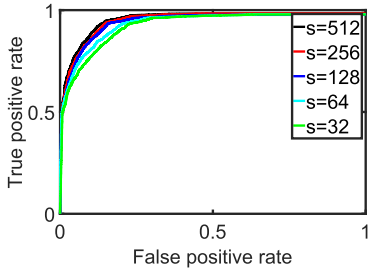


FIGURE 6. ROC for embedding size.

Embedding size: The embedding size refers to the vector length of each category and can be set for each category feature. There are no specific rules for the selection of the embedding size, which can be obtained through experiments. The embedding size selection has an impact on the embedding accuracy and performance. The experimental results in Figure 6 demonstrate that the AUC increased with the increase in embedding size, but the growth rate was reduced when the AUC was greater than 128. To reduce overheads while ensuring accuracy, our final selection was 128.

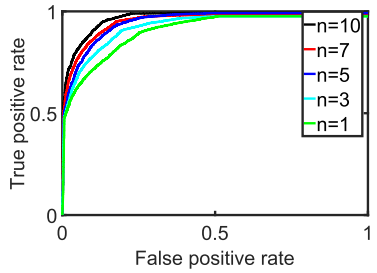


FIGURE 7. ROC for embedding depth.

Embedding depth: The embedding depth is the number of layers of the neural networks in the model, and the size of embedding depth has an impact on the embedding accuracy and performance; if the depth is too great, overfitting may occur. The depth of the residual network is used to increase the embedding depth. As illustrated in Figure 7, the increase in the embedding depth increased the AUC, and our final embedding depth was 10.

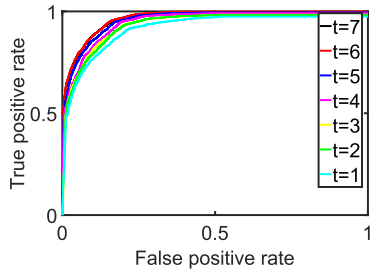


FIGURE 8. ROC for iterations.

Iterations: The number of iterations determines the node propagation distance, as illustrated in Figure 8. The experiment proved that, when the number of iterations was greater than 5, the AUC growth rate slowed; therefore, our final number of iterations was 5.

Our final hyperparameters were selected as follows: embedding size: 128, embedding depth: 10, and number of iterations: 5. Compared with Gemini, both the embedding size and depth were increased significantly. The same dataset was used to compare the performance of our hyperparameters and those of Gemini, and the results demonstrate that our hyperparameters were more accurate.

C. COMPARISON OF FEATURES

The genetic algorithm was used to select 10 effective features from 50 function features to realize function embedding. Our feature combination was compared with that of Gemini to evaluate its performance. The Gemini feature combination is {string constants, numeric constants, no. of transfer instructions, no. of calls, no. of instructions, no. of arithmetic instructions, no. of offspring, betweenness}. Compared with Gemini, we increased no. of calls, no. of called, and no. of edges, among others. The AUC curve of the experiment is illustrated in Figure 9. The experimental results demonstrated

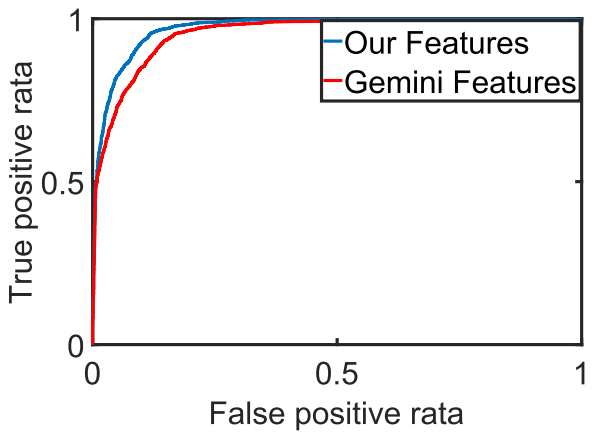


FIGURE 9. ROC curves for feature comparison.

that our feature combination is more accurate than that of Gemini under the same test data. Following the analysis, a possible reason for the superior performance of our method is that our features increased the structural features inside the function and certain invocation features among functions, which are important for function similarity.

D. COMPARISON OF ACCURACY

Our method implements a staged firmware code similarity analysis. The first stage involves calculating the similarity of firmware functions based on code embedding, which is suitable for large-scale firmware security inspection. The second stage uses LCG similarity for fine-grained analysis of firmware security. The accuracy comparison between the first and second stages is illustrated in Figure 10.

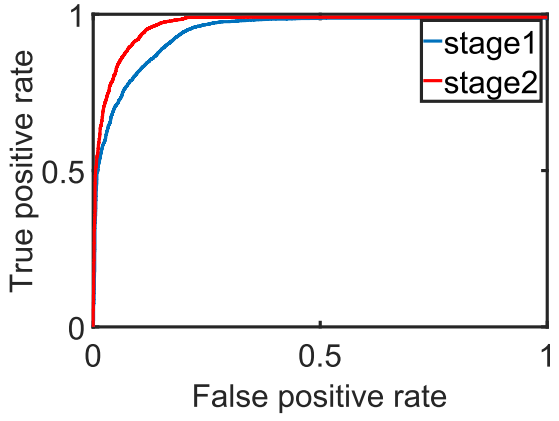


FIGURE 10. Two stages accuracy comparison.

Obviously, the accuracy of the second stage is higher than that of the first stage, while the efficiency of the second stage is lower. The reason is easy to understand: the second stage considers the revoke relations among functions and will increase the accuracy. With the exception of the target functions, the second stage has to calculate the similarity of functions that have call relations with target functions;

TABLE 2. Real-world vulnerability evaluation.

CVE Number	Vulnerability	TOP 30	Accruacy	Affected Manufactures
CVE-2014-0224	ssl3_get_server_hello	13	43.3%	D-Link, TP-Link, Netgear
CVE-2015-0204	ssl3_get_key_exchange	26	86.7%	D-Link, Fuji Xerox, Buffalo, TP-Link, Netgear
CVE-2016-2107	aesni_cbc_hmac_sha1_cipher	18	60%	D-Link, TP-Link, Netgear

therefore, the efficiency will decrease. The first stage can perform large-scale firmware security analysis, and further validation is conducted in the second stage.

E. COMPARISON OF EFFICIENCY

The time cost of our method mainly includes the embedding generation time and similarity calculation time. Real firmware in DataSet2 was used to evaluate the code embedding time. Figure 11 illustrates that our method required a longer time than Gemini, because the neural network structure of our method is more complex than that of Gemini.

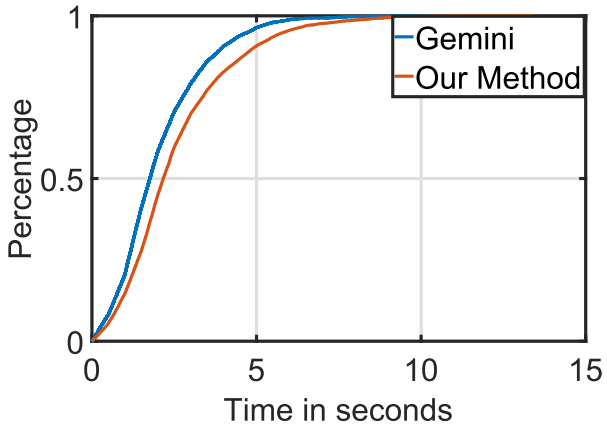


FIGURE 11. Efficiency of our method versus Gemini.

F. THE RESULT OF EMBEDDING ACCURACY

We used function embedding for the firmware function and vulnerability similarity analyses. Therefore, the function embedding accuracy determines the function similarity analysis accuracy. In this experiment, 10 functions were selected for comparison. The 10 functions were compiled using different compilers (GCC and clang) and different compilation options (O1, O2, and O3) to obtain the binary functions of different architectures (ARM, MIPS, and X86). Our method was applied to generate the function embeddings, and t-SNE [29] was used to map the code embeddings to a two-dimensional plane. The mapping results are illustrated in Figure 12. The same functions have the same color, and the mapping result indicates that the embeddings of the same functions will be clustered together with a closer distance, while different functions have a larger distance.

G. APPLICATION OF REAL-WORLD VULNERABILITIES

To verify the ability for detecting real-world vulnerabilities, three real OpenSSl vulnerabilities were used to

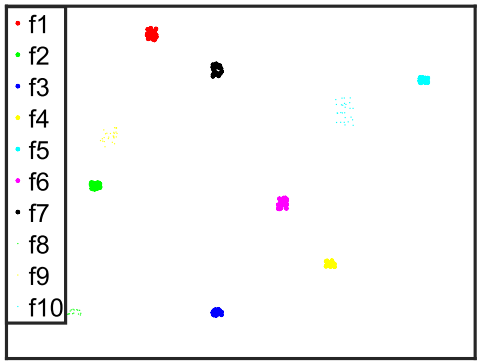


FIGURE 12. Visualizing function embeddings.

detect 3,619,200 firmware functions in dataset 2. Using our method to embed the vulnerability functions, following which LSH [27] was applied for query searching in FirmwareDB. After determining the suspicious vulnerability function, it was necessary to verify whether it is a real vulnerability function manually, which is a time-consuming work. The higher the similarity, the higher the probability that the suspicious vulnerability function is real. Owing the workload, only the suspicious vulnerability functions of the top 30 similarity were verified. So, the sample size of firmware function is 3,619,200 and the sample size of suspicious vulnerability function is 30. The result of real-world case evaluation experiment is shown in table 2.

In the top 30 similarity of vulnerability CVE-2014-0224 [30], 13 true positive vulnerability functions affecting three manufacturers were verified.

In the top 30 similarity of vulnerability CVE-2015-0204 [31], 26 true positive vulnerability functions affecting five manufacturers were verified.

In the top 30 similarity of vulnerability CVE-2016-2107 [32], 18 true positive vulnerability functions affecting three manufacturers were verified.

The experimental results demonstrated that the accuracy of our method could reach up to 86% in real cases. The average accuracy rate of the three experiments was greater than 60%. In the real case, the accuracy rate of our method would be unstable owing to the different mechanisms of vulnerability composition.

The vulnerability function of CVE-2014-0224 is *ssl3_get_server_hello*, which may be exploited by a Man-in-the-middle (MITM) attack where the attacker can decrypt and modify traffic from the attacked client and server. Getting the vulerability function from OpenSSl 1.0.1f. Finding a suspicious vulnerability function in a firmware of NetGear by the

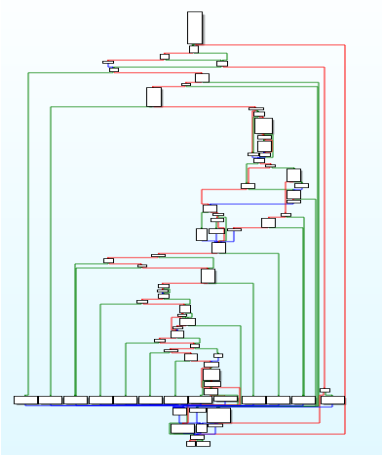


FIGURE 13. CFG of vulnerabiltiy function.

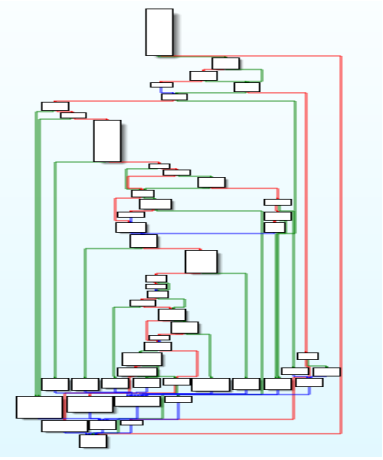


FIGURE 14. CFG of suspicious vulnerability function.

first stage of our method. The CFGs of *ssl3_get_server_hello* and suspicious vulnerability function are shown in Figure 13 and Figure 14. The overview of two graphs are similar, but the order of some basic blocks and some instructions are different. The similarity of the two graphs is 0.71 calculated by the first stage of our method. The LCGs of fucntion *ssl3_get_server_hello* and suspicious vulnerability function are shown in Figure 15 and Figure 16. The similarity of the two LCG graphs is 0.92 calculated by the second stage of our method. The result of the experiment proves that the second stage of our method can improve the accuracy of vulnerability detection effectively.

We determined a 0 day vulnerability in the Kyocera duplicator of a certain type, and similar vulnerabilities were found in other types of Kyocera duplicators by using our method.

H. DISCUSSION

As we discuss in Section *I*, the existing methods of firmware vulnerability detection share at least one of the following shortcomings: incomplete features, high overheads and poor extensibility. Our method takes the invoke relations among functions into account, and uses genetic algorithm to get an optimal weighted feature combination, which improves the function similarity accuracy. Experiment *C* confirm the effect of our feature combination. To reduce the overhead, the first stage of our method uses function embedding for function similarity analysis. Experiment *E* confirm the efficiency of our method. The accuracy of our method is confirmed in experiments *D* and *F*.

During the training process of the embedding model, tag data (similar and non-similar function pairs) are relatively expensive, while the numbers and types of embedded firmware in the IoT are very large; thus, it is difficult to embed all firmware code accurately. Therefore, it is often necessary to retrain the embedded model. Existing methods for firmware security inspection are very costly to retrain, requiring days or even weeks. Our embedded model was generated based on the neural network, so the training efficiency was high. On our equipment, the average retraining time of our method was within 1 h. Thus, our method has a good extensibility. To adapt to the embedding requirements of the new firmware, we changed the composition structure of the training data to form a new training set with a ratio of 20:1, in which the amount of new data was 20 times that of the old data, and the new model was trained with the new data.

VIII. RELATED WORK

Closely related work has been discussed earlier in the paper, and this section briefly introduces the related work of other authors.

A. CODE SIMILARITY ANALYSIS

Bindiff [33], [34] is a commonly used static analysis tool that can be employed to compare changes in binary files. Pewny *et al.* [16] proposed a method to detect binary program security based on code similarity and designed the prototype system TEDEM to verify the method effectiveness. Based on the literature [16], Eschweiler *et al.* [35] designed and implemented the system discovRE, which supports four instruction set architectures (x86, x64, ARM, and MIPS). Compared with TEDEM, discovRE adds a set of filters to improve efficiency significantly. Genius [17] and Gemini [14] have already been introduced. BinHunt [10] uses symbolic and theorem proving to determine the differences among binary programs. iBinHunt [36] extends the work of BinHunt, using deep taint and automatic input generation to determine the semantic differences. *αDiff* [37] uses the inter-function and inter-module features to detect similarities among cross-version binaries. VMPBL [38] can effectively identify the vulnerability function type with the aid of patch files.

B. FIRMWARE SECURITY ANALYSIS

Firmalice [39] uses symbolic execution and program slicing techniques to analyze firmware security. Cui and Stolfo [40] used Nmap tools to identify weak password vulnerabilities in approximately 540,000 embedded devices. Heninger *et al.* [41] used the ZMap [42] network scanner

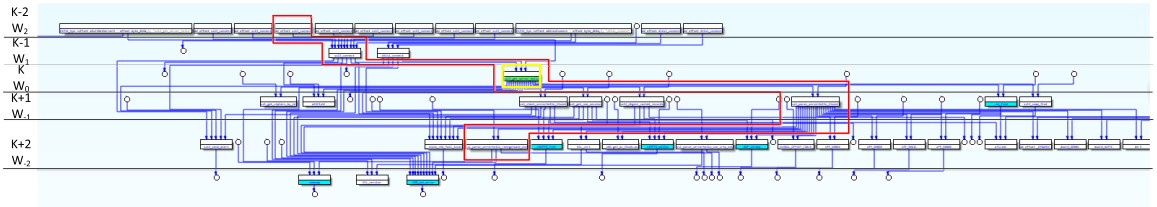


FIGURE 15. LCG of vulnerabiltiy function.

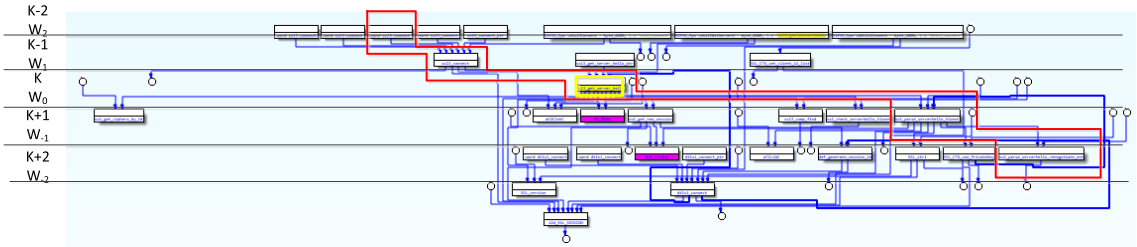


FIGURE 16. LCG of suspicious vulnerability function.

to demonstrate that embedded devices are also affected by entropy problems. Shirani*et al.* [43] proposed a firmware vulnerability detection method for intelligent electronic devices with ARM architecture, and designed BINARM, a multi-stage detection engine, which can inspect the security of IED firmware efficiently. Li *et al.* [44] used natural language processing technology to generate fine-grained firmware fingerprints, which is effective in identifying firmware manufacturers and versions.

In 2014, the University of Vienna, Austria, developed an embedded device dynamic analysis system, PROSPECT, which provides a fully virtualized execution environment for embedded devices. In 2014, the EURECOM research center in France developed the embedded device dynamic analysis platform Avatar [11], which can run a firmware program alternately between the virtual machine and a real device, and uses S2E [45] to perform symbolic execution and stain analysis of the code. However, at present, Avatar can only dynamically analyze peripheral simple equipment. In 2016, Chen et al. developed the automatic dynamic analysis system Firmadyne [12] for embedded firmware, which can realize full system simulation of embedded device programs. The EURECOM research center implemented Avatar2 [46] in 2018. In addition to the above technologies, several vulnerability detection technologies based on source code [29], [47], [48] are available. As the number of open source firmware is small, such methods are not suitable for firmware security analysis in general.

C. GRAPH EMBEDDING

Yan *et al.* [49] proposed the graph embedding framework, which defines two different graphs to describe the dataset characteristics. The intrinsic graph describes the statistical characteristics that need to be enhanced, while the penalty graph describes the statistical characteristics that need to be suppressed. Reference [49] introduced a general framework for graph embedding, and a new algorithm, margin fisher analysis, was proposed based on this framework. Chen *et al.* [50] proposed local discriminant embedding (LDE), kernel LDE, and two-dimensional LDE. Based on LDE, Dornaika and Bosaghzadeh [51] proposed parameterless LDE (ELDE), while Cheng *et al.* [52] proposed incremental LDE (ILDE). Many classical feature extraction algorithms, such as PCA, LDA, LE, LPP, and NPE, can be classified as graph embedding algorithms, and the main difference among these algorithms is the definition of the intrinsic and penalty graphs [53].

IX. CONCLUSIONS

In this paper, we have proposed a staged code similarity analysis method for firmware vulnerability detection. The first stage designs a residual network to embed the function features into the high-dimensional space for efficient analysis of function similarity, moreover, two databases are created to contain the embeddings of the firmware and vulnerability functions, respectively: FirmwareDB and VulnerabilityDB, which can be used to detect the firmware security rapidly. The second stage takes the invoke relations among functions into account and uses the hierarchical weighted bipartite graph to calculate the similarity of function LCG . The combination of the two stages provides higher similarity analysis accuracy compared with existing methods. Moreover, the genetic algorithm is used to optimize our function feature combination which proved very effective. We designed a prototype system and compared it with state-of-the-art methods. The experiments demonstrated that our method achieved superior accuracy to the state-of-art methods, obtained the best result of 26 true positive samples in the top 30 suspicious vulnerable functions in real-world testing, and could complete the retraining process in 1 h.

TABLE 3. Function features.

Feature type	Feature name	Weight
Function call feature	No. of calls	-
	No. of called	-
	No. of indirect calls	-
	No. of lib functions	-
Instruction feature	No. of instructions	-
	No. of arguments	-
	No. of local variate	-
	No. of Basic Block indegree	-
	No. of Basic Block outdegree	-
	No. of Arithmetic instructions	-
	No. of Bit Manipulation instructions	-
	No. of Data Transfer instructions	-
	No. of String instructions	-
	No. of Processor Control instructions	-
	No. of Iteration Control instructions	-
	No. of Interrupt instructions	-
	No. of Load instructions	-
	No. of Store instructions	-
	No. of Call instructions	-
	Strings	-
	No. of strings	-
	Constants	-
	No. of constants	-
Statistical feature	Entropy of instructions	-
	Entropy of Arithmetic instructions	-
	Entropy of Bit Manipulation instructions	-
	Entropy of Data Transfer instructions	-
	Entropy of Execution Transfer instructions	-
	Entropy of String instructions	-
	Entropy of Processor Control instructions	-
	Entropy of Iteration Control instructions	-
	Entropy of Interrupt instructions	-
	Skewness of instructions	-
	Kurtosis of instructions	-
	Standard deviation of instructions	-
	Mean of instructions	-
	Variance of instructions	-
	Z-score of instructions	-
Structure feature	Stack	-
	No. of nodes	-
	No. of edges	-
	No. of paths	-
	No. of nodes in shortest path	-
	No. of loops	-
	Base block depth	-
	Base block average depth	-
	Base block maximum depth	-
	Base block average breadth	-
	Base block maximum breadth	-
	Density of graph	-

However, several limitations exist in our research. At present, we select several features of the function CFG and function call flow graph, which can accurately express a function in most cases. However, the literature [54] has proven that it is not always accurate to judge function similarity only by CFG and LCG features. Therefore, the next step will be to increase the code similarity accuracy by adding function features, including data flow features. The application of machine learning in the field of IoT security remains in the exploratory stage. In this study, we used the residual network to generate firmware code embedding, which is limited by the number of firmware features and datasets, and the hyperparameters obtained by training are not optimal. The next step will involve attempting to use additional machine learning techniques for IoT security. The number of vulnerability function samples relative to the number of IoT devices is insufficient, the next step will collect more types of firmware vulnerabilities.

APPENDIX

See Table 3.

REFERENCES

[1] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal.*, 2009, pp. 117–128.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. NDSS*, vol. 9, 2009, pp. 8–11.

[3] 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.

[4] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation,"Ĭ in *Proc. 20th Annu. Netw. Distrib. Syst. Secur. Symp.* Reston, VA, USA: Internet Society, 2013.

[5] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares," in *Proc. USENIX Secur. Symp.*, 2014, pp. 95–110.

[6] M. A. Khan and K. Salah, "IoT security: Review, blockchain solutions, and open challenges," *Future Gener. Comput. Syst.*, vol. 82, pp. 395–411, Mar. 2018.

[7] 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.

[8] C. Wronka and J. Kotas, "Embedded software debug in simulation and emulation environments for interface IP," in *Embedded Software Verification and Debugging*. New York, NY, USA: Springer, 2017, pp. 19–45.

[9] W. Zhou, Y. Jia, A. Peng, Y. Zhang, and P. Liu, "The effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved," *IEEE Internet Things J.*, to be published.

[10] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proc. Int. Conf. Inf. Commun. Secur.* Berlin, Germany: Springer, 2008, pp. 238–255.

[11] 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.

[12] 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.

[13] Y. Chen, H. Li, W. Zhao, L. Zhang, Z. Liu, and Z. Shi, "IHB: A scalable and efficient scheme to identify homologous binaries in IoT firmwares," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.

[14] 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. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.

[15] C. Kuang, Q. Miao, and H. Chen, "Analysis of software vulnerability," in *Proc. 5th Int. Conf. Inf. Secur. Privacy. World Sci. Eng. Acad. Soc. (WSEAS)*, 2006, pp. 218–223.

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

[17] Q. Feng, 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.

[18] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information," in *Proc. IJCAI*, 2015, pp. 2111–2117.

[19] H. Chen, B. Perozzi, R. Al-Rfou, and S. Skiena. (2018) "A tutorial on network embeddings." [Online]. Available: https://arxiv.org/abs/1808.02590

[20] P. Cui, X. Wang, J. Pei, and W. Zhu, "A survey on network embedding," *IEEE Trans. Knowl. Data Eng.*, 2017. [Online]. Available: https://arxiv.org/abs/1711.08752

[21] C. Eagle, *The IDA Pro Book*. San Francisco, CA, USA: No Starch Press, 2011.

[22] Y. Saeys, I. Inza, and P. Larrañaga, "A review of feature selection techniques in bioinformatics," *Bioinformatics*, vol. 23, no. 19, pp. 2507–2517, 2007.

[23] 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.

[24] 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.

[25] A. Smola, A. Gretton, L. Song, and B. Schölkopf, "A Hilbert space embedding for distributions," in *Proc. Int. Conf. Algorithmic Learn. Theory*. Berlin, Germany: Springer, 2007, pp. 13–31.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[27] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Proc. 47th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*, Oct. 2006, pp. 459–468.

[28] F. Serratosa, "Fast computation of bipartite graph matching," *Pattern Recognit. Lett.*, vol. 45, pp. 244–250, Aug. 2014.

[29] L. van der Maaten, "Accelerating t-SNE using tree-based algorithms," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3221–3245, Oct. 2014.

[30] *CVE-2014-0224*. Accessed: Jan. 22, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0224

[31] *CVE-2015-0204*. Accessed: Jan. 22, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0204

[32] *CVE-2016-2107*. Accessed: Jan. 22, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2107

[33] *Zynamics BinDiff*. Accessed: Jan. 22, 2019. [Online]. Available: https://www.zynamics.com/bindiff.html

[34] D. Thomas and R. Rolles, "Graph-based comparison of executable objects," in *Proc. Symp. sur la Securite des Technol. de l'Inf. et des Commun.*, 2005. Accessed: May 2010. [Online]. Available: http://actes.sstic.org/SSTIC05/Analysedifferentielledebinaires/

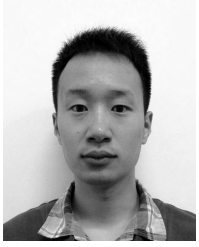
[35] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. NDSS*, 2016, pp. 1–15.

[36] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proc. Int. Conf. Inf. Secur. Cryptol.* Berlin, Germany: Springer, 2012, pp. 92–109.

[37] B. Liu *et al.*, "αDiff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 667–678.

[38] D. Liu, Y. Li, Y. Tang, B. Wang, and W. Xie, "VMPBL: Identifying vulnerable functions based on machine learning combining patched information and binary comparison technique by LCS," in *Proc. 17th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./ 12th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2018, pp. 800–807.

[39] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proc. NDSS*, 2015, pp. 1–15.
[40] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan," in *Proc. 26th Annu. Comput. Secur. Appl. Conf.*, 2010, pp. 97–106.
[41] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices," in *Proc. USENIX Secur. Symp.*, vol. 8, 2012, p. 1.
[42] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *Proc. USENIX Secur. Symp.*, vol. 8, 2013, pp. 47–53.
[43] P. Shirani *et al.*, "**BINARM**: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2018, pp. 114–138.
[44] Q. Li, X. Feng, R. Wang, Z. Li, and L. Sun, "Towards fine-grained fingerprinting of firmware in online embedded devices," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2537–2545.
[45] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
[46] 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.
[47] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The bug catalog of the maven ecosystem," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 372–375.
[48] G. Denaro, M. Pezzè, and M. Vivanti, "On the right objectives of data flow testing," in *Proc. IEEE 7th Int. Conf. Softw. Test. (ICST), Verification Validation*, Mar./Apr. 2014, pp. 71–80.
[49] S. Yan, D. Xu, B. Zhang, H.-J. Zhang, Q. Yang, and S. Lin, "Graph embedding and extensions: A general framework for dimensionality reduction," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 1, pp. 40–51, Jan. 2007.
[50] H.-T. Chen, H.-W. Chang, and T.-L. Liu, "Local discriminant embedding and its variants," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 2, Jun. 2005, pp. 846–853.
[51] F. Dornaika and A. Bosaghzadeh, "Exponential local discriminant embedding and its application to face recognition," *IEEE Trans. Cybern.*, vol. 43, no. 3, pp. 921–934, Jun. 2013.
[52] M. Cheng, B. Fang, Y. Y. Tang, T. Zhang, and J. Wen, "Incremental embedding and learning in the local discriminant subspace with application to face recognition," *IEEE Trans. Syst., Man, C (Appl. Rev.)*, vol. 40, no. 5, pp. 580–591, Sep. 2010.
[53] E. Kokiopoulou, J. Chen, and Y. Saad, "Trace optimization and eigenproblems in dimension reduction methods," *Numer. Linear Algebra Appl.*, vol. 18, no. 3, pp. 565–602, 2011.
[54] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *Proc. 33rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 392–404.



YISEN WANG was born in 1990. He received the B.A. degree from Tianjin University, in 2012, and the M.S. degree in computer science and technology from Information Engineering University, in 2015. He is currently pursuing the Ph.D. degree in computer science and technology with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include computer architecture, Internet of Things security, and deep learning.

JIANJING SHEN was born in 1961. He received the Ph.D. degree in computer science and technology from Information Engineering University, in 1994. He is currently a Computer Science and Technology Professor. His research interests include artificial intelligence, computational intelligence, distributed computing, and new generation Web technology.

JIAN LIN was born in 1989. He received the M.S. degree in computer science and technology from Information Engineering University, in 2016. He is currently pursuing the Ph.D. degree in cyberspace security with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include binary program analysis and vulnerability detection and exploit.

RUI LOU was born in 1989. He received the B.A. degree from the Harbin Institute of Technology, in 2011, and the M.S. and Ph.D. degrees in computer science and technology from Information Engineering University, in 2014 and 2018, respectively. He is currently a Researcher in computer science and technology with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include computer architecture, system virtualization, and computer security.