IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 4, 2020, accepted August 13, 2020, date of publication August 17, 2020, date of current version August 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3017088

Research on Security Detection Technology for Internet of Things Terminal Based on Firmware Code Genes

XINBING ZHU, QINGBAO LI, ZHIFENG CHEN, GUIMIN ZHANG, AND PENG SHAN
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

Corresponding author: Zhifeng Chen (catcheverysecond@sina.com)

This work was supported by the Project of the National Natural Science Foundation of China under Grant 61802432.

ABSTRACT Internet of Things (IoT) terminals have firmware with heterogeneous, closed-source, and heavy business but light security characteristics, whereas on the edge, there are limited resources and a high code reuse rate. Once there are security risks at the firmware level, these risks are difficult to detect and discover, and the resulting impact quickly spreads over a wide range. Therefore, a similarity and homology analysis of firmware codes in an IoT terminal will be helpful for further research on firmware malicious code detection, vulnerability mining, backdoor discovery and copyright protection. Inspired by biological genes, this paper attempts to break away from the traditional feature-centered approach and focuses on code classification and the qualitative description of code features to discuss the idea of code similarity and homology analysis. Additionally, the proposed approach is information-centric, focusing on the informativeness (essentiality, stability, antivariability, and heritability) of the firmware code genes and the quantitative analysis of firmware code similarity and homology by discussing common methods and mechanisms. This paper presents security detection technology for IoT terminal firmware by measuring the gene distance between the codes. A prototype firmware security detection system (FSDS) for IoT terminals based on firmware code genes is designed and implemented. The experimental results show that this method has a good search matching effect and has certain advantages over traditional firmware security detection methods based on similarity theory.

INDEX TERMS The IoT, the IoT terminal, firmware, code gene, gene distance, similarity.

I. INTRODUCTION

With the advent of the Internet of Everything era, the global Internet of Things (IoT) has entered a new wave of development that has been driven by the upgrading of traditional industry and the large-scale consumption market [1]. By 2025, the number of personal intelligent terminals will reach 40 billion, and the total number of global connections will reach 100 billion [2]. These networked terminals are widely used in public utilities, transportation, manufacturing, medical treatment, agriculture, finance and other fields, which greatly change the social operation and way of life of individuals [1]–[5].

A. MOTIVATION

IoT terminals generally have the following characteristics:

The associate editor coordinating the review of this manuscript and approving it for publication was Vyasa Sai.

(1) Heterogeneity and closed source [1], [3]–[6]. The firmware of an IoT terminal is deployed in various architectures, with different instruction sets, registers, addressing modes, stack management, calling conventions, storage management models, etc. Most firmware has closed-source code, is unable to obtain the source code, and lacks symbol debugging information. As a result, the security detection objects of terminal firmware are not unified, and detection is difficult [3]–[6].

(2) Limited resources [3]–[6]. Most IoT terminals belong to the category of embedded devices, with limited storage and computing resources, and many terminals have high requirements for power consumption and real-time performance. Therefore, it is difficult for the terminal itself to deploy antiviral, intrusion detection and other security protection measures. Additionally, it is difficult to adopt underlying monitoring, probing of early warnings and other security monitoring means. As a result, the execution environment

of terminal firmware is not safe, and dynamic detection is difficult [3]–[6].

(3) Business is more important than security [3]. At the beginning of IoT terminal design, due to the design level, cycle, cost and other reasons, the design focus is often on the realization of a business, while the security defense ability of the product itself is ignored. Additionally, some products reserve backdoor or hardcoded information for the convenience of maintenance and management [7]. As a result, the design of terminal firmware has more business-oriented than security-oriented characteristics [3].

(4) High code reuse rate [3], [4], [6]. IoT terminal firmware generally adopts commercial off-the-shelf (COTS) technology and uses a large number of third-party libraries. Although this approach shortens the design cycle and reduces the design cost, once there are security defects in the reused code, the security of more terminals will be threatened. As a result, security defects quickly spread and have a wide range of influence [3], [4].

(5) On the edge [1]–[3]. An IoT terminal is at the intersection of the physical world and the cyber world, which realizes the perception control of the physical world and completes the conversion and interaction between analog state information and digital state information. Therefore, once attacked, both the information security of virtual cyberspace and the physical security of the real world will be affected. It can be said that the location of the IoT terminal makes the attack move forward and expand, and the impact has changed from ''seeking wealth'' in the traditional Internet domain to ''killing'' in the current IoT domain. Consequently, the dimension of the security impact is wider and deeper [3].

From the security events exposed in recent years [1], [2] [7]–[10], attacks against these massive terminals have been shown to emerge endlessly, which has seriously threatened national security, economic development, social life and other aspects. The firmware of these terminals is not only the carrier of the terminal business but is also the main target of attackers. Therefore, whether there is a real security risk or a long-term top-level design, there is an urgent need to study the security detection of IoT terminal firmware.

B. CONTRIBUTIONS

In this paper, we make the following contributions:

(1) We introduce gene theory into the field of security detection for IoT terminal firmware. It is helpful to make use of the information of firmware code genes to conduct more accurate research on firmware security detection.

(2) We attempt to break away from the traditional feature-centered approach and focus on code classification and the qualitative description of code features to discuss the idea of code homology and similarity analysis. Instead, our approach focuses on the information of firmware code genes and performs a quantitative analysis of firmware code similarity and homology by discussing common methods and mechanisms.

(3) We extract firmware code genes from three levels: call relationship between functions, control flow information within functions and basic block statistics information, which provide a material basis for semantic equivalence between firmware binary codes.

(4) We propose a two-stage measurement method of gene distance between firmware codes, which quantitatively describes the similarity between them from two dimensions, improving the accuracy and efficiency.

(5) We design and implement a prototype firmware security detection system (FSDS), which verifies the feasibility of the application of firmware code genes in the field of IoT terminal security detection in practice.

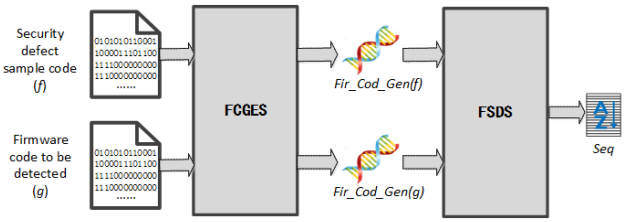


FIGURE 1. The workflow map.

C. WORKFLOW

Based on the research of reference [3], the information (essentiality, stability, antivariability and heritability) of firmware code genes is taken as the logical starting point in this paper. By measuring the gene distance between the known security defect sample code and the firmware code to be detected, the similarity between them was determined to carry out the security detection of the IoT terminal firmware. As shown in Fig. 1, the firmware code gene extraction system (FCGES) has been introduced in detail in reference [3], and this paper only supplements and improves it. We will focus on the design and implementation of the FSDS.

D. OUTLINE

The remainder of this paper is structured as follows: Section II discusses the related works; Section III introduces firmware code gene extraction; Section IV presents a method to measure the firmware code gene distance; Section V designs and implements the FSDS; Section VI presents the experimental steps and results and verifies the effectiveness and robustness of our method; and Section VII discusses the deficiencies of this paper and the plans for follow-up studies.

II. RELATED WORKS

Traditional code analysis technologies are mainly divided into dynamic and static analyses. However, when these technologies are combined with firmware code, some problems occur.

A. DYNAMIC ANALYSIS

Dynamic analysis refers to a program analysis method that records the relevant information of program execution by using various analysis technologies and displays it to

analysts in various ways by executing the analyzed samples in a controllable operation environment [5], [11]–[14]. This method can directly track and record the running behavior of the program and provides the most direct support for security detection. However, since the operation of firmware is highly dependent on the physical hardware, IoT terminals have limited resources. The first issue to be solved is how to make the firmware run in a controllable environment and how to track and record the operation behavior when the traditional dynamic detection technology is applied to the IoT terminal firmware. Therefore, the dynamic detection of the IoT terminal firmware can be carried out either in a real physical device or in a virtual environment simulation [5].

Avatar [11], [12] performs dynamic analysis by partially offloading the execution of the firmware to actual hardware. However, running on real hardware is both expensive and effective only for specific devices, with poor universality and scalability. *FIRMADYNE*[5] relies on software-based full system emulation with an instrumented kernel to achieve the scalability necessary to automatically analyze thousands of firmware binaries. Costin *et al.* [13] performed full system emulation to achieve the execution of firmware images in a software-only environment, i.e., without involving any physical embedded devices.

Deployment on real physical devices is expensive, and the problem domain is only for the device itself. In a virtual environment simulation, we also solve the problem of how to simulate the interaction with the hardware. Even a small interactive simulation failure will cause the running code to crash. From the current research situation, it can be seen that when the traditional code dynamic analysis technology is applied to the IoT terminal firmware, its applicability or generality can be greatly challenged.

B. STATIC ANALYSIS

Static analysis [15]–[27] refers to code analysis technology that uses relevant analysis tools to analyze lexical, grammatical, control flow, data flow and other information without executing the analyzed samples. Although this technology does not need actual running code, there are several limitations when the analysis tools are applied to the firmware. First, these tools are mostly aimed at the source level, which is contradictory to the closed source of the IoT terminal firmware [17]. Second, these tools are often aimed only at a single architecture, especially x86, which is contradictory to the cross-platform deployment of IoT terminal firmware [18]. Third, these analysis techniques have limited capabilities and are often targeted at specific problem domains, such as C, PHP, Java or the corresponding binary code, which is contradictory to the fact that IoT terminal firmware is often a mixture [19]. More importantly, static analysis techniques cannot solve the problem of the interaction between the firmware and hardware.

In recent years, a number of security detection technologies for IoT terminal firmware based on similarity have emerged—for example, similarity analysis based on features [4], [25], [26], on intermediate representation [20], [21], on graph (or tree) structures [22]–[24], [27], and on machine learning [4], [23], [24]. However, there are three common questions in these studies: First, is attribute information, such as the comparative features and graph structure, essential? This is determined by whether the information can identify the code itself in terms of grammar and semantics. Second, is the comparative attribute information stable and antivariable? This is determined by whether the information is intrinsic and the code can be identified on different platforms. Third, is the attribute information compared heritable? This is determined by whether the information exists stably in the same series or in similar code.

Inspired by biological genes [28], in reference [3], the concept of firmware code genes is proposed, and its extraction with the idea of the hypothesis margin [34] is realized, which make up the basic part of the previous research in theory. Based on the research of reference [3], this paper focuses on the information of firmware code genes and measures the gene distance between firmware codes to realize the security detection of IoT terminal firmware.

III. EXTRACTION OF FIRMWARE CODE GENES

In this section, we first give a brief overview of firmware code genes and then extract the firmware code genes from the three levels of function call relationship information, function internal control flow information and basic block statistics information based on reference [3].

A. FIRMWARE CODE GENE OVERVIEW

The concept of genes is not new to the field of code security detection [29]–[32]. In software similarity and homology analysis, there have been several relevant studies. Before that, there were studies on software features, software fingerprints and software birthmarks [33]. Software features are based on statistical information and the behavior information of code, but sometimes they are not common and stable when used as the basis of a similarity measurement. Although software fingerprints and software birthmarks compensate for the lack of software features to some extent, they pay more attention to the identification of individuals and ignore the association of the same series and family. Software genes [29]–[31] attempt to sublimate the previous research but also fail to resolve three issues: First, the composition of a software gene remains unknown. The lack of an answer to this question means that they have not completed the mapping of the biological gene composition to the software space. Thus, when defining a software gene, only a ''binary fragment carrying functional information'' is used [29]. Second, the difference between a software gene and a feature has not been identified; that is, the sublimation of software features to software genes has not been completed, so the software gene has become a well-known concept that cannot be clearly defined. Third, the universal processing of numericalization and normalization of software genes has not been investigated; that is, the transformation from concrete to general has not been

accomplished. Thus, prior studies have mostly focused on the search and matching of specific text information rather than the numerical calculation.

Reference [3] starts from the composition of the firmware code genes, completes the structural mapping from biological genes to firmware code genes, and expounds its materiality. With the idea of the hypothesis margin, the sublimation from the feature of firmware code to the gene of firmware code is completed, and the essential difference between the feature and gene is answered. Through the numerical and normalized processing of different data types, the differences of the different features in the data types, the practical significance and the value range are shielded, and the specific to general transformation is completed, which enhances the universality and scalability.

B. FIRMWARE CODE GENE EXTRACTION

Reference [3] takes a large number of common codes in the IoT terminal firmware as the sample space, constructs a positive sample dataset and a negative sample dataset, and completes the sublimation from features to genes through the FCGES. Through experiments, we can see that in different architectures and datasets, the extracted firmware code genes are essential, stable, antivariable and heritable, and the gene vectors are highly coincident in dimensions. These factors provide a theoretical basis for the security detection of IoT terminals based on firmware code genes. This section will further improve and optimize the composition of firmware code genes.

1) EXTRACTION OF THE ORIGINAL FEATURES

From the perspective of syntax, the function is the basic component of the firmware code; from the perspective of semantics, the function and the call relationship between functions can well reflect the semantic information. In fact, the function of the IoT terminal firmware code also depends on the library function it calls to a large extent. Therefore, compared with the original feature vector in reference [3], this study adds the information of the interfunction call relationship in the function call graph.

The control flow information and basic block code statistics within the function are introduced in detail in reference [3] and will not be covered. The new interfunction call relationship information mainly refers to the interfunction call relationship centered on the corresponding nodes in the function call graph, as shown in Table 1. *Called_ Num* is the number of times the function is called, that is, the input degree of the node. *Call_Num* is the number of calls to the function, that is, the output degree of the node. *DCalled_ Num* is the deduplicate number of times the function is called, and *DCall_Num* is the deduplicate number of times the function calls. *In_ Num* is the number of input parameters of the function. *Out_ Num* is the number of output parameters of the function. *InOut_ Num* is the number of two-way parameters of the function. When two-way parameters are encountered, *In_ Num* and *Out_ Num* each add 1, and the total number of parameters of the function is *In_ Num* +*Out_ Num-InOut_ Num*. *RetValue_Typ* is the return value type of the function, and its data type is a string. *Node_ Clu* is the node clustering coefficient, calculated according to formula (1):

TABLE 1. The extended features of the firmware code.

Name	Type	Name	Type	Name	Type
Called_Num	1	*Dcall_Num*	1	*Inout_Num*	1
Call_Num	1	*In_Num*	1	*Retvalue_Typ*	4
Dcalled_Num	1	*Out_Num*	1	*Node_Clu*	1
Remarks	numerical type:1; string type:4.				

$$Node\_Clu = \frac{2c}{d\,(d-1)} \tag{1}$$

where c is the edge number of the undirected function call subgraph composed of all adjacent nodes of the node, and d is the degree of the undirected graph of the node.

2) NUMERICALIZATION AND NORMALIZATION

Although reference [3] analyzes only three types of original features, such as the numerical type, set-valued type and sequential type, it also notes that according to the actual application, it is not limited to these three data types. If there are better original features that can more accurately reflect the syntactic or semantic similarity between codes, as long as they can be numericalized and normalized, then the method in reference [3] can still be used to extract the firmware code genes. Therefore, FCGES is universal and scalable.

In this paper, we add attribute information that can reflect the call relationship between functions. For *RetValue_Typ*, we add the string data type. For numericalization and normalization, it is calculated according to formula (2):

$$Sim\,(V_1[i], V_2[j]) = \begin{cases} 1, & if\ V_1[i] = V_2[j], \\ 0, & else \end{cases} \tag{2}$$

For functions with no return value, we consider the return value to be null.

3) EXTRACTION OF FIRMWARE CODE GENES

We place the original feature vector into the FCGES and extract the firmware code gene. By using the sample space constructed by the tools and libraries commonly used in multiple firmware, we calculate the discrimination ability [36] of the close samples in each dimension of the original feature vector and obtain similar results to those of reference [3]:

1) The firmware code gene vectors obtained from different datasets have a high degree of coincidence in terms of the dimensions.

2) For the node-centered interfunction call relationship information, *Called_Num, Call_Num, DCalled_Num* and *DCall_Num* have strong essentiality, stability, antivariability and heritability when identifying firmware code similarity in different platforms.

3) For the attribute information of the control flow and the code statistical information of the basic block in the function, the result is consistent with reference [3].

Notably, in the original feature vector extraction stage, 9 dimensions of features are added to describe the call relationship. After FCGES processing, only *Called_Num, Call_Num, DCalled_Num* and DCall_*Num* have a strong ability to distinguish the close samples, which is consistent with our intuitive analysis. The function call relationship has strong robustness. *Called_Num* and *Call_Num* of the similar functions have strong identification ability for distinguishing the function itself in their respective function call graph. In particular, *DCalled_Num* and DCall_*Num* have more advantages in identifying functions. For example, if function A is called 10 times, but all of them are called by the same function B, then the deduplicated number of times called and calls are more significant for identifying function A and function B.

Therefore, this study extracts the firmware code gene from the IoT terminal firmware *Fir_Cod_Gen* = (*Sta_Spa*, *Stafra_Num*, *CmpIns_Num*, *JumIns_Num*, *CmpIns_Rat*, *JumIns_Rat*, *Str_Num*, *Str_Set*, *Con_Num*, *Con_Set*, *Nod_Num*, *Edg_Num*, *Gra_Den*, *Deg_Ave*, *Indeg_Max*, *Deg_Max*, *PatLen_Ave*, *PatDia*, *Indeg_AscLis*, *Outdeg_AscLis*, *Deg_AscLis*, *Pat_AscLis*, *Called_Num*, *Call_Num*, *DCalled_Num*, *DCall_Num*). When the firmware code gene is extracted from a function f, it is expressed as *Fir_Cod_Gen(f)*. In the following sections, the gene vector is divided into two subgene vectors: One is the function-centered firmware code gene on the function call graph *CG_Cod_Gen* = (*Sta_Spa*, *Stafra_Num*, *CmpIns_Num*, *JumIns_Num*, *CmpIns_Rat*, *JumIns_Rat*, *Str_Num*, *Str_Set*, *Con_Num*, *Con_Set*, *Called_Num*, *Call_Num*, *DCalled_Num*, *DCall_Num*). The other is the firmware code gene on the function internal control flow graph *CFG_Cod_Gen* = (*Nod_Num*, *Edg_Num*, *Gra_Den*, *Deg_Ave*, *Indeg_Max*, *Deg_Max*, *PatLen_Ave*, *PatDia*, *Indeg_AscLis*, *Outdeg_AscLis*, *Deg_AscLis*, *Pat_AscLis*). When the firmware code gene on the function call graph is extracted for a function f, it is expressed as *CG_Cod_Gen(f)*; when the firmware code gene on the internal control flow graph of a function is extracted, it is expressed as *CFG_Cod_Gen(f)*.

IV. FIRMWARE CODE GENE DISTANCE MEASUREMENT

In this section, we first define the concept of the firmware code gene distance and then use the gene distance to measure the similarity between firmware codes, which provides the basis for the approximate equivalent judgment of the semantic similarity between the firmware codes.

A. OVERVIEW OF THE FIRMWARE CODE GENE DISTANCE

Generally, in the field of biological genes [28], different species, different families and different individuals have different genes, but the degree of gene similarity between the same species is higher than that between different species, and the degree of gene similarity between the same family in the same species is higher than that of different families. Therefore, in biology, the similarity of genes is often used to classify individuals and cluster families.

In fact, firmware code genes also have this feature. Due to different instruction architectures, compiler tools and optimization options, different target binary codes generated by the same or similar source code have different forms [3], [37]–[39]. However, the firmware code genes hidden in these binary codes show stability, antivariability and heritability, which can essentially identify the code itself. In reference [3], this theory is described in detail. On this basis, this study will carry out research on security detection based on firmware code genes with the help of the concept of the gene distance.

Definition 1 (Firmware Code Gene Space): Each function in the firmware binary code corresponds to a firmware code gene vector. These gene vectors form a nonempty set Ω, which we call the firmware code gene space.

It can be seen from reference [3] that the material carrier of the firmware code gene can be flexibly selected according to different application scenarios. In this study, function level granularity is chosen.

Definition 2 (Firmware Code Gene Distance): The firmware code gene distance refers to the quantitative measurement of the degree of differentiation between the firmware code genes. For any two points α and β in Ω, there is a real number *Dis* (α, β), which reflects the similarity between firmware code genes α and β. We call $dis(\alpha, \beta)$ a distance in Ω—that is, the firmware code gene distance between α and β.

This study uses a two-stage search algorithm to measure the distance of the firmware code gene on the function call graph *Dis_CG* and the distance of the firmware code gene on the function CFG *Dis_CFG*. *Dis_CG* in this paper is represented by the cosine distance, while the cosine distance satisfies only the positive qualitative properties and symmetry, not trigonometric inequality. Therefore, we do not use the classical definition of the distance space when defining the gene distance of the firmware code [35], [36], [40].

B. FIRMWARE CODE GENE DISTANCE ON THE CG

For *Dis_CG*, we use the cosine distance [35], [36], [40] to measure it. The cosine distance is a measure mechanism that uses the cosine value of the angle between two vectors in multidimensional space to measure the difference between two individuals. The more similar the two individuals are, the smaller the angle between their vectors is, the higher the cosine similarity is, and the smaller the cosine distance is.

Let the functions to be compared be f and g. Let α = *CG_Cod_Gen(f)*, and β = *CG_Cod_Gen (g). Then*, *Dis_CG* is calculated as follows the equation can be derived, as shown at the bottom of next page:

In this paper, *CG_Cod_Gen* is an m-dimensional vector. If the function f is more similar to g, then *CG_Cod_Gen(f)* and *CG_Cod_Gen(g)* are more similar,

the cosine angle θ of the vector is smaller, the cosine similarity cos(*CG_Cod_Gen(f), CG_Cod_Gen(g)*) is higher, and the cosine distance *Dis_CG (CG_Cod_Gen(f),CG_Cod_Gen(g))* is smaller.

C. FIRMWARE CODE GENE DISTANCE ON THE CFG

To measure *Dis_CFG*, the Euclidean distance [35], [36], [40] is used. The Euclidean distance refers to the real distance between two points in Euclidean space. The more similar the two individuals are, the closer they are in the Euclidean space, and the smaller the Euclidean distance is.

Let the functions to be compared be f and g. Let α = *CFG_Cod_Gen(f)* and β= *CFG_Cod_Gen (g)*. Then, *Dis_CFG* is calculated as follows:

$$\begin{aligned} &Dis\_CFG(\alpha, \beta) \\ &= Dis\_CFG\left(CFG\_Cod\_Gen(f), CFG\_Cod\_Gen(g)\right) \\ &= \sqrt{\sum_{k=1}^{n} (CFG\_Cod\_Gen(f)[k] - CFG\_Cod\_Gen(g)[k])^2} \end{aligned}$$

In this paper, *CFG_Cod_Gen* is an n-dimensional vector. If the function f is more similar to the function g, then *CFG_Cod_Gen(f)* and *CFG_Cod_Gen(g)* are more similar, and the European distance *Dis_CFG (CFG_Cod_Gen(f), CFG_Cod_Gen(g))* is smaller.

As shown in Fig. 2, the Euclidean distance represents the absolute difference in the numerical value and is not sensitive to the spatial direction, while the cosine distance represents the relative difference in direction and is not sensitive to the absolute value. At the same time, these two distances are used to correct the problem that the user space measurement standards may not be uniform. In this study, the cosine distance is used to measure the similarity of the firmware code genes on the CG, and then the Euclidean distance is used to measure the similarity of firmware code genes on the CFG. After reducing the influence of the direction difference on the similarity of the firmware code, the absolute distance between the firmware code genes is evaluated, which can not only improve the accuracy of firmware code similarity measurement but also improve the space-time efficiency of the overall security detection.

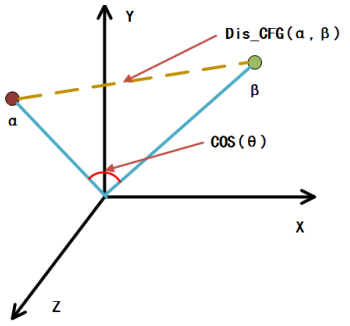


FIGURE 2. The firmware code gene distance map.

V. SECURITY DETECTION BASED ON THE FIRMWARE CODE GENE

In this section, the information of the firmware code gene is taken as the logical starting point, and *Dis_CG* and *Dis_CFG* are taken as the basis to complete the firmware security detection.

A. IDEAS FOR SECURITY DETECTION

In this study, firmware code genes are extracted from known security defect sample codes, and then FSDS designed in this study is used to search for the existence of the same or similar codes in the firmware to be detected. The ''security defect'' here refers to the security defect in a broad sense, including vulnerabilities, malicious code and backdoors. A defect sample is the firmware code that contains the vulnerability function, malicious code, or backdoor core function itself and its associated context. Obviously, the sample code itself should have a strong ability to distinguish it from other codes. For the convenience of the narration, security defects such as vulnerabilities, malicious code and backdoors are collectively referred to as vulnerabilities in subsequent sections.

According to the analysis of section IV, we adopt a two-stage search strategy to implement the FSDS. In the first stage, we first measured *Dis_CG* between the security defect sample code and the code to be detected and then use the CGDS algorithm designed in this study to calculate the overall similarity driven by the local sub-graph of the function call graph. In the second stage, we measure *Dis_CFG* between codes. The specific steps are shown in Fig. 3.

$$\begin{aligned} Dis\_CG(\alpha, \beta) &= 1 - cos\theta = 1 - \cos(\alpha, \beta) \\ &= 1 - \frac{\alpha \cdot \beta}{\|\alpha\| \, \|\beta\|} \\ &= 1 - \frac{CG\_Cod\_Gen(f) \cdot CG\_Cod\_Gen(g)}{\|CG\_Cod\_Gen(f)\| \, \|CG\_Cod\_Gen(g)\|} \\ &= 1 - \frac{\sum_{k=1}^{m} CG\_Cod\_Gen(f)[k] \, CG\_Cod\_Gen(g)[k]}{\sqrt{\sum_{k=1}^{m} (CG\_Cod\_Gen(f)[k])^2} \sqrt{\sum_{k=1}^{m} (CG\_Cod\_Gen(g)[k])^2}} \end{aligned}$$

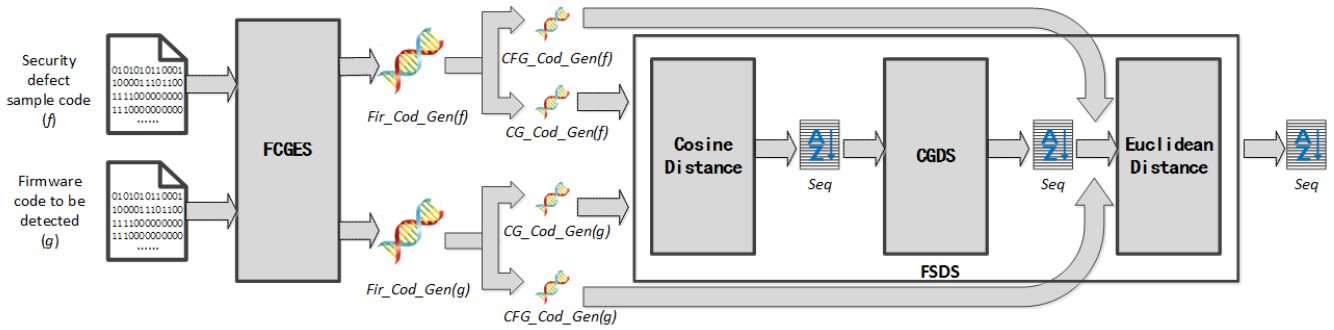


FIGURE 3. The two-stage search strategy map.

B. DESIGN AND IMPLEMENTATION

(1) Extract the firmware code gene from the vulnerability function f in the known security defect sample *Fir_Cod_Gen(f)*, and split the gene into two subgene vectors: *CG_Cod_Gen* (f) and *CFG_Cod_Gen(f)*.

(2) Generate the function call graph [40], [41] of the firmware code to be detected, traverse each node in the graph, and generate their *CG_Cod_Gen*. In this step, we obtain a function call graph with attribute information $CG = (V, E, \Psi, P)$.

Among them, V is the vertex set, and each vertex is a function. E is the edge set, $E \subseteq V \times V$. Ψ is the correlation function between the vertex and the edge, indicating the call relationship between the functions. P is the attribute set of the node, indicating the firmware code gene on the function call graph of the node. $\forall e \in E$, then $\exists v_i, v_j \in V$. For $(v_i, v_j) \in V \times V$ and $\Psi(e) = (v_i, v_j)$, where e is called the edge from v_i to v_j, and $\Psi(e)$ represents the calling relationship between v_i and v_j. That is, v_i calls v_j, and v_j is called by v_i. $\forall v_i \in V$, then $\rho_i \in P$, and $\rho_i = CG\_Cod\_Gen(v_i)$.

(3) Traverse *CG* and calculate the firmware code gene similarity between the known sample vulnerability function and each node in the *CG*—that is, the firmware code gene distance *Dis_CG*. Then, generate a function similarity descending list *Seq* according to the firmware code gene distance *Dis_CG* of every node.

(4) CGDS, a function call graph-driven local similarity measurement algorithm, is used to calculate the overall similarity of each function in the function list *Seq* in the local function call subgraph (LCG) [40], [41]. Based on this, the function similarity descending list *Seq* is updated.

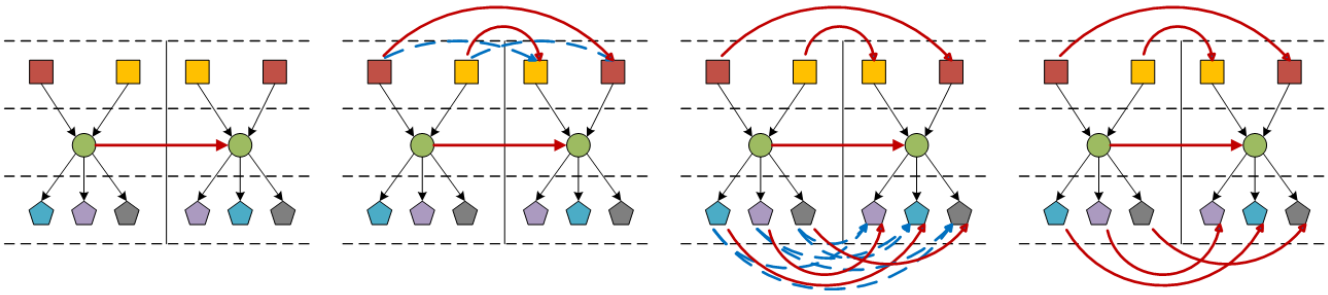
Previous studies have revealed that the function call graph has strong robustness to different instruction architectures and different optimization options [3], [4], [20]. Therefore, this study uses CGDS to update the similarity of each function in the function list *Seq* generated in the previous step. The specific calculation process is shown in Fig. 4. In the function call subgraph composed of the node to be matched and its adjacent nodes, the weighted average similarity of the function to be matched and all the functions with their calling and called relationships with the function to be matched is calculated. This weighted average similarity is the overall similarity of the function to be matched on the function call graph.

It can be seen in Fig. 4 that the local function call subgraph is a subgraph composed of the node to be matched, its adjacent nodes with edge connections, and the edges between them. That is, the function call subgraph is composed of the functions to be matched and the functions with their calling and called relations as the node set, and the calling and called relations between them as the edge set. In fact, two local function call subgraphs centered on the function pair to be matched constitute a weighted complete bipartite graph [40]–[43], and the function call graph firmware code gene distance between node pairs in the same layer is the weight of the edge. In this study, we use the Kuhn-Munkres algorithm [41]–[43] to calculate the maximum weight matching of the weighted complete bipartite graph as the similarity between the nodes of the graph and then calculate the weighted average of the similarity of all matching nodes as the overall similarity of the function pairs to be compared. In this way, the descending function similarity ranking list *Seq* is updated.

The algorithm is as follows.

In algorithm 1, Line 2 corresponding to (a) in Fig. 4, computes *Dis_CG* of the function to be matched. Lines 3 to 7, corresponding to (b) in Fig. 4, generate a bipartite graph composed of the parent node of the function to be matched and *Dis_CG* between the parent nodes and solve the maximum matching of the bipartite graph. Lines 10 to 14, corresponding to (c) in Fig. 4, generate a bipartite graph composed of child nodes of the function to be matched and Dis_CG between the child nodes and solve the maximum matching of the bipartite graph. Lines 17 to 20, corresponding to (d) in Fig. 4, calculate the overall similarity of the function to be matched on the local function call subgraph.

(5) The Euclidean distance is used to measure the similarity of the firmware code gene on the function CFG between the known sample vulnerability function and the functions in the function list *Seq*. This similarity is the firmware code gene distance of the function CFG. The descending function similarity ranking list



(a) Measurement of the similarity between the nodes to be matched

(b) Measurement of the similarity between the parent nodes to be matched

(c) Measurement of the similarity between the child nodes to be matched

(d) Measurement of The global similarity between the nodes to be matched on LCG

FIGURE 4. The CGDS algorithm map.

Algorithm 1 The CGDS Algorithm

Input: Function f and g to be matched
Local function call subgraph centered on f, $LCG(f)$
Local function call subgraph centered on g, $LCG(g)$

Output: Global similarity driven by local function call subgraph between f and g, $Sim\_LCG(f, g)$

Procedure

(1) $PQ = \emptyset$;
(2) $Sim\_LCG(f, g) = Dis\_CG(CG\_Cod\_Gen(f), CG\_Cod\_Gen(f))$;
(3) Generate the parent node set Vpf of f in LCG (f);
(4) Generate the parent node set Vpg of g in LCG (g);
(5) $Simp = \{Sim_{ij} = Dis\_CG(CG\_Cod\_Gen(vpf_i), CG\_Cod\_Gen(vpg_j)) | \forall\ vpf_i \in Vpf, vpg_j \in Vpg\ \}$;
(6) Take Vpf and Vpg as the vertex sets, and $Simp$ as the weighted edge set to generate bipartite graph $Gp = (Vpf, Vpg, Simp)$;
(7) Using Kuhn-Munkres algorithm to generate the maximum matching of bipartite graph Gp, $Mp =$ Kuhn-Munkres (Gp);
(8) For each $(vpf_i, vpg_j) \in Mp$
(9) PQ.push((vpf_i, vpg_j), Sim_{ij}) ;
(10) Generate the child node set Vsf of f in LCG (f);
(11) Generate the child node set Vsg of g in LCG (g);
(12) $Sims = \{Sim_{ij} = Dis\_CG(CG\_Cod\_Gen(vsf_i), CG\_Cod\_Gen(vsg_j)) | \forall\ vsf_i \in Vsf, vsg_j \in Vsg\ \}$;
(13) Take Vsf and Vsg as the vertex sets, and $Sims$ as the weighted edge set to generate bipartite graph $Gs = (Vsf, Vsg, Sims)$;
(14) Using Kuhn-Munkres algorithm to generate the maximum matching of bipartite graph Gs, $Ms =$ Kuhn-Munkres (Gs);
(15) For each $(vsf_i, vsg_j) \in Ms$
(16) PQ.push((vsf_i, vsg_j), Sim_{ij}) ;
(17) while (PQ != $\emptyset$)
(18) $m = PQ$.pop;
(19) $Sim\_LCG(f, g) = Sim\_LCG(f, g) + m.Sim_{ij}$;
(20) $Sim\_LCG(f, g) = Sim\_LCG(f, g)$ / (PQ.length + 1);
(21) Return $Sim\_LCG(f, g)$;

Seq is updated according to the firmware code gene distance *Dis_CFG*.

In this study, we did not measure the CFG similarity inside the function until this step, partly because of the practical need to further improve the function matching accuracy. On the other hand, and more importantly, after the local similarity measurement based on the function call graph, it can effectively mask the influence of the CFG heterogeneity within the function caused by different instruction architectures, different compilers and different optimization options on the similarity measurement. Moreover, the firmware code gene on the function CFG extracted in this study is not based on each basic block inside the CFG but on the structured CFG information extracted from the global perspective; that is, the global and structured CFG similarity is measured.

VI. EXPERIMENT AND ANALYSIS

In this section, we will take the common codes and real firmware as the experimental dataset and verify the effectiveness and robustness of our method in firmware security detection by analyzing the influence of different platforms, compilers and optimization options on function search and matching in detail.

A. EXPERIMENTAL THOUGHTS

Because of the detection algorithm in this study, the final result is a list of descending functions. Therefore, in the experimental part, we will mainly focus on whether there is code similar to the sample in the firmware code to be detected, as well as their similarity ranking and distribution. It should be emphasized that the descending similarity list given by the detection algorithm reflects only the similarity between the sample and the code in the firmware to be detected, and it cannot be determined that the function with the highest ranking or the top ranking is the vulnerability function. In fact, the verification of the vulnerability function is an independent research field. However, providing highly suspected objects to analysts for further analysis is very meaningful work, which can greatly improve the efficiency of security detection.

In this section, we will use the following indicators to evaluate the experimental results.

Definition 3 (Perfect Match and the Proportion of Perfect Match): A function f in code F matches all the functions in code G to obtain a ranking list of descending similarity. The ranking of function g in the list that truly matches f is the *Rank*. If $Rank = 1$, the match is called a perfect match. The proportion of perfect match refers to the ratio of the number of perfect matching functions to the total number of functions in F, which is recorded as *Top1*.

Definition 4 (Approximate Match and the Proportion of Approximate Match): If the function g in the list that truly matches f is ranked within 10—that is, $Rank \leq 10$—then the match is called an approximate match. The proportion of the approximate match refers to the ratio of the number of all approximate matching functions to the total number of functions in F, which is recorded as *Top10*.

Definition 5 (Reference Match and the Proportion of Reference Match): If the function g in the list that truly matches f is ranked within 100—that is, $Rank \leq 100$—then the match is called a reference match. The proportion of reference match refers to the ratio of the number of all reference matching functions to the total number of functions in F, which is recorded as *Top100*.

In addition, we used *Topn* in some experiments. *Topn* refers to the proportion of functions with $Rank \leq$ n. Obviously, in the process of matching, the *Rank* value evaluates the matching effect from the perspective of individuals such that the smaller the value is, the better the matching effect. The *Top* value evaluates the matching effect from the perspective of the whole, so the larger the value is, the better the effect.

B. EXPERIMENTAL ENVIRONMENT

The experimental environment is as follows. The central processing unit (CPU) is an Intel Core i7-6700@ 3.40 GHz, and the memory is 16.0 GB of DDR4 SDRAM. The binary object code is compiled to a 32-bit x86, a 32-bit ARM and a 32-bit MIPS using GCC v4.6.2, GCC v4.8.1 and Clang v3.0, respectively. The Python programming language[44] is used to implement the FSDS. IDA Pro[45] is used to disassemble the binary code and write plug-ins to extract code features. A tool provided by MATLAB R2014b[46] is utilized for feature selection, gene sublimation and gene distance measurements.

C. EXPERIMENT

1) THE INFLUENCE OF DIFFERENT PLATFORMS ON FUNCTION MATCHING

In this experiment, we will use the same compiler and the same optimization options to compile the same source code on different platforms to test the influence of heterogeneous platforms on the detection algorithm.

We use the most commonly used compiler GCC v4.62 and the default optimization option O2 to compile OpenSSL v1.0.0 [47] on three different platforms, ARM, MIPS and x86, and obtain the object binary code with different forms. We use the method designed in this paper to match the function of the same name in these codes and calculate the distribution of the similarity ranking *Topn* (n = 1,10,20,30,40,50,60,70,80,90, 100). The experimental results are shown in Fig. 5.

In Fig. 5, the following can be seen.

1) In the six matching modes, the results of the proportion of perfect match *Top1* are in the interval (70%, 80%), the proportion of approximate match *Top10* is in the interval (80%, 90%), and the proportion of reference match *Top100* is in the interval (85%, 95%). The results of using this method are better.

2) In the six matching modes, the slope of the similarity ranking proportion curve is gradually flattened from large to small and finally tends to a straight line. That is, even if the value of n in *Topn* is increased, *Topn* will not change much, which we call Top-saturated. This is very important, as the substantial workload for subsequent analysis and detection is reduced, and the efficiency and accuracy of security detection is greatly improved.

3) In the six matching modes, ARM and x86 have better matching effects, and MIPS and x86 have worse matching effects. This is mainly because MIPS is a typical reduced instruction set computer (RISC) architecture [37]–[39]. To realize the pipeline, the instruction length is fixed, and unlike the complex instruction set computer (CISC) architecture [37]–[39], there are special instructions to complete the specific functions. What can be done with one instruction under the CISC architecture often requires multiple instructions under the RISC architecture (such as in stack and out stack). Compared with x86, the same source code compiled on the MIPS platform will have more instructions

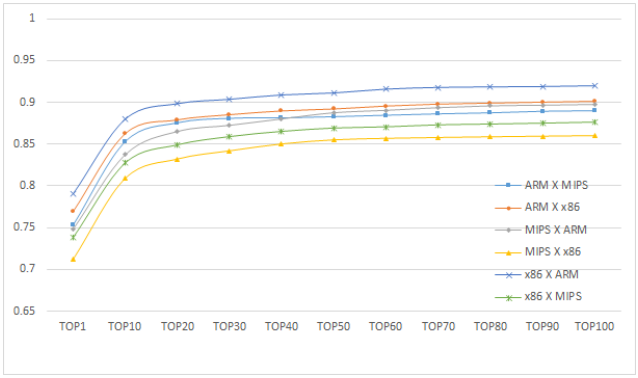


FIGURE 5. The distribution of *TOPn* in experiment I.

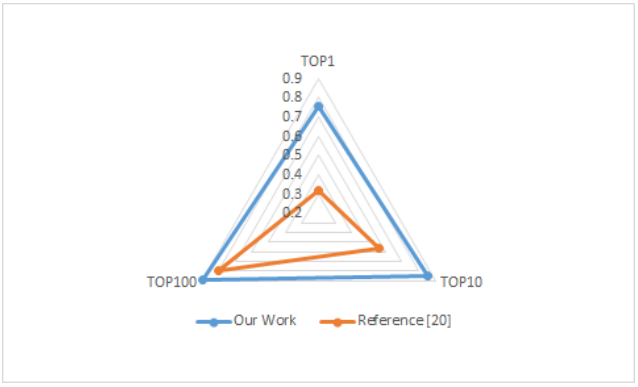


FIGURE 6. The comparison with reference [20] in experiment.

and basic blocks. Although the ARM architecture belongs to the RISC architecture, it also absorbs some advantages of CISC (such as multiregister instructions and conditional execution). When the same source code is compiled on the ARM architecture, the number of instructions and basic blocks is between MIPS and x86. The differences among the ARM, MIPS and x86 architectures will be reflected in the function call graph CG and the function CFG of the code. Therefore, when measuring the firmware code gene distance on the function call graph *Dis_CG* and the firmware code gene distance on the function CFG *Dis_CFG* belonging to the CISC architecture x86 and RISC architecture MIPS, respectively, the gene distance is larger, and the similarity match effect is poorer. However, even so, we can still see that the function call graph CG and the function CFG of the binary code generated by the same source code under different architectures still maintain sufficient similarity.

4) In the six matching modes, even in two matching patterns with the same architecture, if the matching direction is different, then the distribution of the similarity ranking proportion is not the same, and the matching results are not symmetrical. As shown in Fig. 5, in ARM × x86 mode, $Top1 = 77.01\%$, $Top10 = 86.28\%$, and $Top100 = 90.12\%$. In the opposite direction of the x86 × ARM mode, $Top1 = 79.97\%$, $Top10 = 88.01\%$, and $Top100 = 91.99\%$. This result is because when we calculate the firmware code gene distance on the function call graph *Dis_CG*, we use the whole similarity measurement driven by the function call graph. As shown in Fig. 4, this method is unidirectional and is a global measure of the firmware code gene distance between each node in the local function call subgraph centered on the function to be matched in the sample code function call graph and the corresponding node in the code to be detected. In fact, this is a scoring method for the sample code. As mentioned above, the function call graph and CFG will be more or less heterogeneous under different architectures, so the matching direction is opposite, and the matching results are not symmetrical. For the function call subgraph with fewer penalty points, the distance of the firmware code gene on the function call graph *Dis_CG* is smaller, and the matching result is better. Therefore, the x86 × ARM mode matches better than the ARM × x86 mode.

In addition, for ARM × MIPS, in this paper, $Top1 = 75.32\%$, $Top10 = 85.29\%$, and $Top100 = 88.97\%$. Under similar experimental conditions, the experimental results in reference [20] show that $Top1 = 32.1\%$, $Top10 = 56.1\%$, and $Top100 = 80\%$. As shown in Fig. 6, the two methods are close only in the direction of *Top100*, but there is a large gap in other directions, especially in the direction of *Top1*.

It can be seen that the experimental results in this paper are much better than those in reference [20]. This improvement may be due to two reasons.

One reason is that reference [20] does not provide theoretical proof of the comparability and robustness of the similarity measurement basis used. In other words, reference [20] adopts the formula I/O extraction and sampling based on the basic block, which is highly susceptible to the influence of the platform, compiler and optimization options. However, this influence is not discussed theoretically, but the rationality of the method is verified in turn by experimental results.

Second, because the BHB algorithm[20] used in the signature search phase is very sensitive to the heterogeneity of the CFG, even if the local graph structure of the CFG changes, it will increase the possibility of a mismatch, which is very unfavorable for heterogeneous platforms.

The method used in this study effectively solves these two problems.

First, the detection method in this paper is based on the firmware code gene. Reference [3] has theoretically demonstrated that firmware code genes are essential, stable, antivariable and heritable in heterogeneous environments. That is, before the similarity comparison in this paper, we theoretically answer whether the measurement basis has comparability and robustness. This answer lays a solid foundation for improving the accuracy of the matching results.

Second, the detection method in this paper is implemented by measuring two distances between firmware code genes. Both the distance measurement method and the design of the whole detection algorithm shield the influence of CFG heterogeneity under different platforms to a certain extent. This method also guarantees the accuracy of matching results from the detection process.

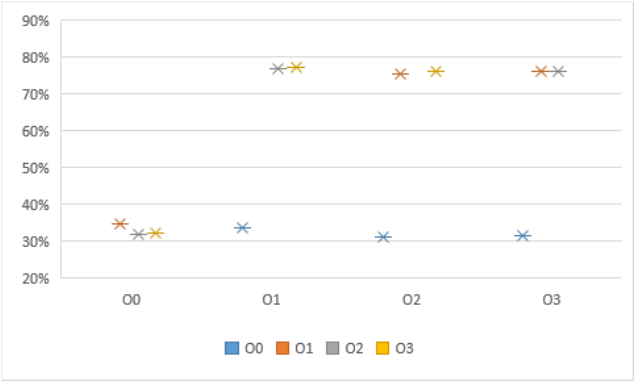


FIGURE 7. The impact of different optimization options in experiment II.

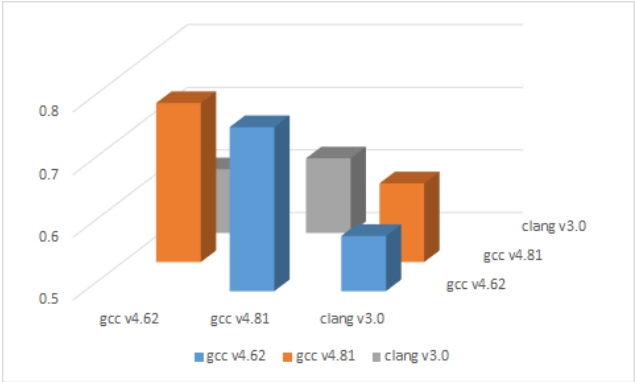


FIGURE 8. The impact of different compilers in experiment III.

2) THE INFLUENCE OF DIFFERENT OPTIMIZATION OPTIONS ON FUNCTION MATCHING

In this experiment, we will use the same compiler and different optimization options to compile the same source code on the same platform to test the influence of different optimization options on the detection algorithm.

Using the compiler GCC v4.62, we compile BusyBox v1.21.1[48] on the ARM platform, choosing four different optimization options (O0-O3), resulting in four different binary target codes. As in experiment 1, we use our method to match the functions in them and calculate the proportion of perfect matches. The experimental results are shown in Fig. 7 and are described as follows.

1) In the 12 matching modes, the value range of *Top1* is between 30% and 80%. Additionally, the value range of *Top1* is between 70% and 80%, except for match modes involving the O0 optimization option. Thus, the method presented in this paper is also robust to different optimization options.

2) In the 12 matching modes, *Top1* ranges from 30% to 40% regardless of the matching direction when the O0 optimization option is involved. Compared with other matching modes, the matching result is not satisfactory. This result shows that code with no optimization option has a great impact on the matching results, but this kind of matching still has some reference significance, and in the real firmware code space, there are fewer codes with no optimization.

3) In the 12 matching modes, the value range of *Top1* is between 70% and 80% regardless of the matching direction, except for the other six matching modes involving the O0 optimization options. This finding means that functions with optimization options match, and the optimization options have little effect on the matching results, which may be related to the compatibility of the optimization strategies for high-level options with those for low-level options.

3) THE INFLUENCE OF DIFFERENT COMPILERS ON FUNCTION MATCHING

In this experiment, we will use different compilers and the same optimization options to compile the same source code on the same platform to test the impact of different compilers on the detection algorithm.

BusyBox v1.21.1 is taken as the experimental object and is compiled on the MIPS platform using three common compilers: GCC v4.62, GCC V4.81 and Clang v3.0. The optimization option is O2, resulting in three different binary target codes. As in experiment 2, two pairs of their functions are matched using the method presented in this paper to calculate *Top1*. The experimental results are shown in Fig. 8 and are described as follows.

1) In the six matching modes, regardless of the function match between the target code generated by the two compilers and the direction of matching, the value range of *Top1* is between 50% and 80%. This result shows that the method in this paper has strong robustness to different compilers.

2) In the six matching modes, the function match between the target code generated by the same series of compilers GCC v4.62 and GCC v4.81, regardless of the matching direction, yields a value range of *Top1* between 70% and 80%. This finding indicates that the function matching between the series compilers is good and is related to the similar compiling and optimization techniques used by the same compiler series.

3) In the six matching modes, function matching between the target codes generated by different series of compilers, regardless of the matching direction, yields a value of *Top1* that is lower than that of the same series, but the matching results are still meaningful.

4) THE INFLUENCE OF SIMILAR CODE ON FUNCTION MATCHING

In this experiment, we will use the same compiler and the same optimization options to compile similar source codes to the same platform to verify the effectiveness of this method for function matching between similar codes.

We compiled BusyBox v1.20.0 and BusyBox v1.21.1 on the x86 platform using GCC v4.62, the most commonly used compiler, and the optimization option O2. Using the method designed in this study, we matched the functions with the same name in these codes and calculated the distribution of similarity ranking *Topn* (n = 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100). The experimental results are shown in Fig. 9.

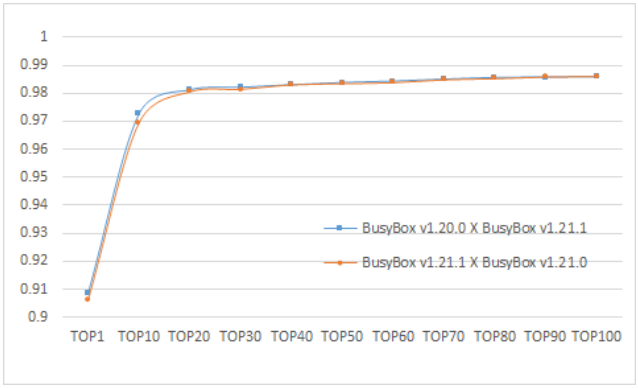


FIGURE 9. The distribution of *TOPn* in experiment IV.

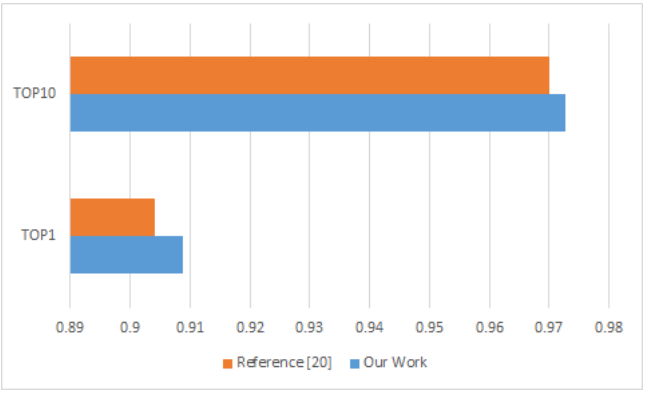


FIGURE 10. The comparison with reference [ ] in experiment IV.

1) In the two matching modes, the result of *Top1* exceeds 90%, which indicates that the result is good when the method designed in this paper is used to match the function with the same name between BusyBox V1.20.0 and BusyBox V1.21.1.

2) In the two matching modes, the slope of the similarity ranking proportion curve is similar to that of experiment 1; that is, the slope changes from large to small, gradually flattens, and finally tends to a straight line. However, the difference is that the initial slope of the curve in this experiment is larger, and the curve flattening is faster; that is, the saturation speed of the *Top* value is faster.

3) In the two matching modes, the matching results in both directions have a higher degree of coincidence than in experiment 1. Especially after *Top20*, the results are basically coincident. Even *Top1* and *Top10* are not very different.

From these three points, we can see that this experiment and Experiment 1 have both the same place and a different place. Similarly, the existence of firmware code genes is verified in practice, which verifies the validity and robustness of the method designed in this study. The difference is as follows: First, the similarity ranking in Experiment 2 has a higher starting point, a larger initial slope, and a faster curve flattening speed. Second, the results of the two matching directions in Experiment 2 are symmetrical.

This symmetry may be due to three reasons: First, the version between BusyBox V1.20.0 and BusyBox V1.21.1 is relatively close, and the code itself is quite similar. Second, Experiment 1 was carried out on a heterogeneous platform, while Experiment 2 was carried out on the same platform. In other words, the matching accuracy under the same architecture was higher than that under different architectures. Third, Experiment 2 not only verified the essentiality, stability and antivariability of the firmware code gene verified in Experiment 1 but also verified the heritability.

In addition, in the BusyBox v1.20.0 × BusyBox v1.21.1 matching mode, $Top1 = 90.87\%$, and $Top10 = 97.26\%$. In similar experimental environments, the results in reference [20] show that $Top1 = 90.4\%$ and $Top10 = 97\%$. As shown in Fig. 10, the method in this paper is slightly better than that in reference [20]. However, considering that this experiment uses the same compiler and the same optimization options to compile similar source code on the same platform, the proportion of the perfect match *Top1* has exceeded 90%, so this advantage does not seem very large, but under different platforms, different compilers, and different optimization options, the advantages may be more obvious.

TABLE 2. The rank value of experiment V.

	V1.1.0-ARM	V1.1.0-MIPS	V1.1.0-x86
V1.1.0-ARM	--	1	1
V1.1.0-MIPS	1	--	2
V1.1.0-x86	1	1	--

In the above four experiments, the validity and robustness of the method designed in this study for function matching are investigated and analyzed from the platform, compiler, optimization options and matching direction, taking the target code generated by the same source code and similar source code as the research object. The experimental results not only verify the experimental purpose but also provide a practical basis and a benchmark for subsequent experiments. More importantly, in line with reference [3], the existence of firmware code genes has been proved in practice, along with the essentiality, stability, antivariability and heritability of the firmware code genes.

5) OpenSSL VULNERABILITY CVE-2016-6305

In version 1.1.0 before OpenSSL V1.1.0a, there is a security vulnerability CVE-2016-6305 in the ssl3_read_bytes function in the file record/rec_layer_s3.c. This vulnerability allows remote attackers to cause a denial of service (infinite loop) by triggering a zero-length record in an SSL_peek call.

We use GCC v4.62 and optimization option O2 to compile OpenSSL V1.1.0 on the ARM, MIPS and x86 platforms and extract the firmware code genes of the function SSL3_read_bytes in the three platforms. The method in this paper is used to search for matches in the binaries on other platforms, and the match rankings are shown in Table 2. From these results, we can see that the matching ranking is more ideal.

TABLE 3. The rank value of experiment VI.

	V1.1.0 -ARM	V1.1.0 -MIPS	V1.1.0 -x86	V1.1.3 -ARM	V1.2.0 -MIPS	V1.3.0 -x86
V1.1.0 -ARM	1	2	2	1	2	4
V1.1.0 -MIPS	2	1	2	2	1	5
V1.1.0 -x86	1	2	1	1	3	1

The experimental results show the following.

1) In the six matching modes, the matching result is ideal, *Rank* $\leq$2.

2) The worst ranking (*Rank* = 2) appears in the MIPS $\times$ x86 matching pattern, which is consistent with the analysis in Experiment 1.

6) BusyBox VULNERABILITY CVE-2018-20679

An issue (Vulnerability CVE-2018-20679) was discovered in BusyBox before 1.30.0. An out-of-bounds read in the udhcp component (consumed by the DHCP server, client, and relay) allows a remote attacker to leak sensitive information from the stack by sending a crafted DHCP message. This issue is related to verification in udhcp_get_option() in networking/udhcp/common.c that 4-byte options are indeed 4 bytes.

We obtained the binary code of BusyBox V1.1.0 on the ARM, MIPS and x86 platforms, from which we extracted the firmware code gene of the function udhcp_get_option In the case that the compiler and optimization options are unknown, the binary code of BusyBox V1.1.0-arm, V1.1.0-MIPS, V1.1.0-x86, V1.1.3-ARM, V1.2.0-MIPS and V1.3.0-x86 are taken as experimental objects to search and match the vulnerability function using the method designed in this study. The experimental results are shown in Table 3 and are described as follows.

1) Overall, the matching rank value is not as ideal as Experiment 5, but it is still relatively advanced at *Rank* $\leq$5, which is related to the unknown compiler and optimization options.

2) Compared with other match modes, the matching rank in MIPS $\times$ x86 mode is still lower.

3) Matching under the same platform, with an unknown compiler and optimization options, the *Rank* value is 1, which may be due to the code pairs being similar or the same.

7) VULNERABILITY OF THE TP-LINK ROUTER CVE-2017-16957

Command injection vulnerability CVE-2017-16957 exists in the firmware of multiple TP-Link routers (TL-WVR, TL-WAR, TL-ER, TL-R, etc.). This vulnerability allows remote authenticated users to execute arbitrary commands via shell metacharacters in the iface field of an admin/diagnostic command to cgi-bin/luci, related to the zone_get_effect_devices function in/usr/lib/lua/luci/controller/admin/diagnostic.lua in uhttpd.

TABLE 4. The rank value of experiment VII.

	TL-WVR450 L	TL-WAR302	TL-ER3210G	TL-R473G
TL-WVR450 L	--	1	2	1
TL-WAR302	1	--	3	2
TL-ER3210G	3	2	--	3
TL-R473G	1	1	3	--

We retrieved four versions of TP-Link router firmware binaries containing CVE-2017-16957 vulnerabilities—namely, TL-WVR450 L, TL-WAR302, TL-ER3210G and TL-R473G—from which the firmware code genes of the vulnerability function zone_get_effect_devices were extracted. The method designed in this study was used to search and match the vulnerability function in four versions of the firmware binary code. The matching results are shown in Table 4.

Unlike the previous experiments, this experiment is conducted with real closed-source firmware code. That is, we no longer focus closely on the platform of code deployment, the compiler used, and the optimization options selected. It employs a direct search match under unknown circumstances, which can better test the effectiveness and robustness of the firmware code gene and the method designed in this study. From the experimental results, we can see that the overall matching result is ideal, at *Rank* $\leq$3. This result may be related to the selected firmware, which is designed and produced by the same manufacturer in the same period. Generally, the design tools, design strategies, third-party libraries, and even the core code and hardware environment that they use are quite similar.

8) BACKDOOR OF THE D-LINK ROUTER

Some D-Link routers have backdoors. With this backdoor, an attacker can access the Web control interface without a username and password. The reason for this backdoor concerns the alpha_auth_check function of the firmware application Web server (/bin/Webs). There is a backdoor password "xmlset_roodkcableoj28840 ybtide" detection in this function; if true, then it directly allows access to the login status—that is, without any authentication to access the Web control interface. The backdoor exists in several types of router firmware for D-Link and Planex routers.

We obtained four firmware binaries containing the backdoor, DIR-100, DI-524UP and DI-604S for the D-Link router and BRL-04UR for Planex, from which we extracted the firmware code genes of the function alpha_auth_check. The method designed in this study was used to perform a function matching search among them. The experimental results are shown in Table 5.

In addition to the vulnerabilities, the backdoor is also an important research topic of firmware security detection. Similarly, this experiment is a direct search match in real closed-source firmware binary code. From the experimental results, the overall matching effect is good, at *Rank* $\leq$2. This

TABLE 5. The rank value of experiment VIII.

	DIR-100	DI-524UP	DI-604S	BRL-04UR
DIR-100	--	1	1	2
DI-524UP	1	--	2	2
DI-604S	1	1	--	1
BRL-04UR	2	2	1	--

TABLE 6. The time overhead of experiment V.

	V1.1.0-ARM	V1.1.0-MIPS	V1.1.0-x86
V1.1.0-ARM	--	3.1s	2.9s
V1.1.0-MIPS	3.2s	--	3.3s
V1.1.0-x86	2.9s	3.0s	--

finding also verifies the validity of the firmware code gene and the design method in this paper for firmware backdoor detection in practice.

D. EXPERIMENTAL PERFORMANCE

Since the extraction of firmware code genes can be completed before the search, and can be performed only once, this paper calculates only the time overhead of search matching. We take Experiment 5 as an example because it allows us to focus on performance overhead without considering the compiler, optimization options, and source code implications. In the experiment, the length of the *Seq* list is set to 200, and the experimental results are shown in Table 6.

As seen in the table, code similar to the security defect sample was searched from OpenSSL in an acceptable amount of time. Due to the time overhead of the first-stage search, it is mainly focused on the *CGDS* algorithm, which is linearly dependent on the size of the function call graph of the code to be detected. While the same source code is compiled on different platforms using the same compiler and the same optimization options, the function call graph retains enough similarity (if inline functions are not considered) that the time overhead for different platforms is relatively close. As for the overhead of the second-stage search, since the size of the *Seq* list has been set to 200, the impact is also a constant.

Therefore, we can also see that the two-stage search strategy adopted in this study, on the one hand, is to meet the needs of algorithm design; on the other hand, it also provides countermeasures for the balance of precision and cost. The length of the *Seq* list will directly determine the size of the second-stage search. In this experiment, it is set as 200, mainly because the TOP curve tends to saturation after $n = 100$.

E. EXPERIMENTAL REVELATION

The above experiments can be divided into three parts.

(1) Full-function search matching with codes commonly used in the firmware of IoT terminals as experimental objects, including Experiments 1, 2, 3, and 4. This section mainly analyzes the impact of different platforms, different compilers, different optimization options and different matching directions on function search matching.

(2) The vulnerability function search matching, which takes real vulnerabilities and codes commonly used in the firmware of IoT terminals as subjects, contains Experiments 5 and 6. Based on the previous part, the validity of this method to search and match real vulnerability functions is verified.

(3) Vulnerability function search matching with real vulnerability (backdoor) and real closed-source terminal firmware binary code of the IoT includes Experiments 7 and 8. Based on the first two parts, this part further verifies the validity of this method in searching and matching of the real vulnerability functions in real firmware.

The three parts are closely related and progressive. The first part provides the analysis basis and comparison benchmarks for the latter two parts, and the second part makes practical preparations for the third part. By combining the experiments, we can see the following.

(1) The detection algorithm designed in this study has a good search and match effect on functions and has certain advantages over traditional methods in the field of IoT terminal security detection.

(2) Using the method designed in this study for full-function search matching, the *Top* value saturation speed is fast. When we encounter a large scale of function call graph nodes to be detected, this method will help us effectively reduce the workload of subsequent inspection measurements, which will greatly improve the efficiency of IoT terminal firmware security detection.

(3) In different architectures, although the result of function search matching is different, it is still ideal in general. This ideal result is critical for IoT terminal firmware that is widely deployed on heterogeneous platforms.

(4) In different matching directions, the results are sometimes asymmetric, but the overall difference is only quantitative, not enough to cause qualitative changes. This is useful for the security detection of IoT terminal firmware because sometimes we cannot specify a matching direction.

(5) Function matching without optimization options is not ideal, but it has some reference value, and few codes do not use optimization strategies in reality.

(6) In binary code generated by the same or the same family of compilers, function searching and matching perform better than in cross-family compilers. However, COTS technology is widely used in the design of IoT terminal firmware, and when the same manufacturer designs the same series of products, it generally uses the same compiler, which has little effect in reality.

VII. SUMMARY

In this paper, based on reference [3], an IoT terminal security detection method based on firmware code genes is proposed, and the FSDS is designed and implemented. This method uses the essentiality, stability, antivariability and heritability of the firmware code genes of the IoT terminals, measures the gene distances of the firmware codes and evaluates the similarity between codes to achieve security detection. The results show that the FSDS has a good function search matching

effect on the firmware binary code under different platforms, compilers, optimization options and matching directions and has high efficiency and robustness for IoT terminal firmware security detection. However, the research on firmware code genes is still in its infancy, related research is still incomplete, and some work still must be undertaken in the future.

(1) The purpose of this paper is to complete the search matching of known vulnerabilities in the firmware to be detected. The result is a descending ranking list reflecting the similarity with the sample vulnerability function. Determining whether the function that ranks first or at the top of this list is a vulnerability function that is beyond the scope of this paper. In fact, the validation of vulnerabilities is a research field in itself.

(2) This paper presents a security detection method based on firmware code genes for IoT terminals. This method completes security detection by measuring the distance between firmware code genes. The firmware code gene distance on the function call graph, the firmware code gene distance on the function CFG and the two-stage search algorithm are used to measure the overall similarity in this study. However, if there is a better way to describe the firmware code gene distance or a more precise algorithm to complete the search matching between functions, then they can still be applied to the system designed in this study. In other words, the detection system designed in this study is open and scalable.

(3) This paper corresponds to reference [3], proves the existence of the firmware code gene of the IoT terminal in practice, and applies it to the security detection of the IoT terminal firmware. However, our experiments focus on vulnerability function matching. Research studies on malicious code detection, backdoor discovery, copyright protection, homology analysis, etc., must be further strengthened.

(4) The object of this research is firmware binary code, but to highlight vulnerability function search matching and firmware security detection, the ''binary code'' in this study is the binary code that was unpacked, with a peeled file system, not the firmware binary image before unpacking.

(5) From the method designed in this study, we can describe the code similarity by measuring the firmware code gene distance on the function call graph and the firmware code gene distance on the function CFG. This description requires that sufficient similarity be maintained between the code to be detected and the sample on both the function call graph CG and the function CFG. Therefore, the validity and robustness of this method for codes with confusion, encryption and other techniques remain to be studied.

(6) In real firmware code, once a manufacturer finds a security defect, it usually solves it by patching. Sometimes, the difference between patched and unpatched code is not so great; it is just a judgment statement or a few lines of code. This situation reflects that the differences in the firmware code genes and gene distances are much less obvious. Therefore, the validity and robustness of the methods presented in this paper for the vulnerability function and vulnerability function patches remain to be studied.

REFERENCES

[1] *White Paper on Internet of Things Security*, China Inst. Inf. Commun., Beijing, China, 2018.
[2] (May 27, 2020). *In-depth Analysis Report on Internet of Things Security and Applications in 2018*. [Online]. Available: https://wenku.baidu.com/view/40b64e8329ea81c758f5f61fb7360b4c2e3f2ad8.html
[3] X. Zhu, Q. Li, P. Zhang, and Z. Chen, ''A firmware code gene extraction technology for IoT terminal,'' *IEEE Access*, vol. 7, pp. 179591–179604, 2019.
[4] C. Qing, L. Zhongjin, and W. Mengtao, ''VNDS: An algorithm for cross-platform vulnerability searching in binary firmware,'' *J. Comput. Res. Develop.*, vol. 53, no. 10, pp. 2288–2298, 2016.
[5] D. D. Chen, M. Egele, M. Woo, and D. Brumley, ''Towards automated dynamic analysis for Linux-based embedded firmware,'' in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.
[6] F. I. Khan and S. Hameed, ''Understanding security requirements and challenges in Internet of Things (IoTs): A review,'' *J. Comput. Netw. Commun.*, vol. 2019, pp. 9629381:1–9629381:14, Jan. 2019.
[7] (May 7, 2019). *Talking About the Safety of Camera*. [Online]. Available: https://mp.weixin.qq.com/s/xY6W-zq2dzgeH4N6t6-ouQ
[8] S. Hilt, V. Kropotov, F. Merces, M. Rosario, and D. Sancho, ''The Internet of Things in the cybercrime underground,'' Trend Micro Res., Tokyo, Japan, Tech. Rep., 2019. [Online]. Available: https://documents.trendmicro.com/assets/white_papers/wp-the-internet-of-things-in-the-cybercrime-underground.pdf
[9] N. Saeed, M.-S. Alouini, and T. Y. Al-Naffouri, ''Toward the Internet of underground things: A systematic survey,'' *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3443–3466, 4th Quart., 2019.
[10] (May 7, 2019). *Thoughts on a Vulnerability in Industrial Control System*. [Online]. Available: https://mp.weixin.qq.com/s/LZnvDQ9lSqgfd8LvKKgteA
[11] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, ''Avatar: A framework to support dynamic security analysis of embedded systems' firmwares,'' in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–16.
[12] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, ''Avatar2: A multi-target orchestration platform,'' in *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.)*, vol. 11, Feb. 2018, pp. 1–11.
[13] A. Costin, A. Zarras, and A. Francillon, ''Automated dynamic firmware analysis at scale: A case study on embedded Web interfaces,'' in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur. (ASIA CCS)*, 2016, pp. 437–448.
[14] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, ''Dynamic malware analysis in the modern era—A state of the art survey,'' *ACM Comput. Surv.*, vol. 52, no. 5, pp. 1–48, 2019.
[15] A. Danese, G. Pravadelli, and V. Bertacco, ''Work-in-progress: DOVE: Pinpointing firmware security vulnerabilities via symbolic control flow assertion mining,'' in *Proc. 12th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth. Companion (CODES)*, Oct. 2017, pp. 1–12.
[16] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. B. Butler, ''FirmUSB: Vetting USB device firmware using domain informed symbolic execution,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2245–2262.
[17] F. Gauthier, T. Lavoie, and E. Merlo, ''Uncovering access control weaknesses and flaws with security-discordant software clones,'' in *Proc. 29th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2013, pp. 209–218.
[18] S. H. H. Ding, B. C. M. Fung, and P. Charland, ''Asm2 Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.
[19] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, ''A large-scale analysis of the security of embedded firmwares,'' in *Proc. Usenix Secur. Symp.*, 2014, pp. 95–110.
[20] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, ''Cross-architecture bug search in binary executables,'' in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.
[21] Y. David, N. Partush, and E. Yahav, ''Similarity of binaries through re-optimization,'' in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2017, pp. 79–94.
[22] 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.*, Oct. 2016, pp. 480–491.
[23] D. Zhao, H. Lin, L. Ran, M. Han, J. Tian, L. Lu, S. Xiong, and J. Xiang, ''CVSkSA: Cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph,'' *Softw. Qual. J.*, vol. 27, pp. 1045–1068, Feb. 2019.

[24] 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.*, Oct. 2017, pp. 363–376.
[25] P. Shirani, L. Wang, and M. Debbabi, "BinShape: Scalable and robust binary library function identification using function shape," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerabil. Assessment*. Cham, Switzerland: Springer, 2017, pp. 301–324.
[26] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "BinSign: Fingerprinting binary functions to support automated analysis of code executables," in *Proc. IFIP Int. Conf. ICT Syst. Secur. Privacy Protection*. Cham, Switzerland: Springer, 2017, pp. 341–355.
[27] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proc. ACM Asia Conf. Comput. Commun. Secur.* New York, NY, USA: ACM, Apr. 2017, pp. 155–166.
[28] S. Mukherjee, *The Gene: An Intimate History*, X. Ma, Ed. Beijing, China: CITIC Press Corporation, 2018.
[29] H. Jin, S. Zheng, and Z. Binglin, "Detection and classification of Android malware based on malware gene," *Appl. Res. Comput.*, vol. 36, no. 6, pp. 1813–1818, 2019.
[30] (Jun. 7, 2019). *Written After the Sub-Forum of 'Software Genetics Technology' (I)*. [Online]. Available: https://www.sohu.com/a/228476725_468696
[31] (Jun. 7, 2019). *Written After the Sub-Forum of 'Software Genetics Technology' (II)*. [Online]. Available:http://www.sohu.com/a/228946546_468696
[32] (Jun. 7, 2019). *CCF Held Seminar on Software Gene in ZhengZhou*. [Online]. Available: https://www.ccf.org.cn/c/2018-12-06/657463.shtml
[33] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 491–511, May 2018, doi: 10.1109/TSE.2017.2688383.
[34] M. Yang and P. Yang, "Hypothesis-margin model incorporating structure information for feature selection," in *Proc. 2nd Int. Symp. Electron. Commerce Secur.*, May 2009, pp. 634–639.
[35] G. Edgar and M. Parmenter, *Discrete Mathematics With Graph Theory*, 3rd ed. Beijing, China: China Machine Press, 2020.
[36] Z. Zhihua, *Machine Learning*. Beijing, China: Tsinghua Univ. Press, 2016.
[37] K. R. Irvine and L. B. Das, *Assembly Language for X86 Processors*. Upper Saddle River, NJ, USA: Prentice-Hall, 2011.
[38] D. Sweetman, *See MIPS Run Linux*. Amsterdam, The Netherlands: Elsevier, 2010.
[39] G. Lei, *Development of Embedded Linux System Based on ARM*. Beijing, China: Tsinghua Univ. Press, 2014.
[40] D. B. West, *Introduction to Graph Theory (Classic)*, 2nd ed. London, U.K.: Pearson, 2018.
[41] G. Suixiang, *Graph Theory and Network Flow Theory*. Beijing, China: Higher Education Press, 2009.
[42] B. Korte, "Combinatorial optimization: Theory and algorithms," *Algorithms Combinatorics*, vol. 146, no. 1, pp. 120–122, 2002.
[43] H. Zhu, M. Zhou, and R. Alkins, "Group role assignment via a Kuhn–Munkres algorithm-based solution," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 42, no. 3, pp. 739–750, May 2012, doi: 10.1109/TSMCA.2011.2170414.
[44] A. Downey, P. Wentworth, and J. Elkner, *How To Think Like a Computer Scientist: Learning With Python*, 2nd ed. Beijing, China: Posts and Telecom Press, 2016.
[45] C. Eagle, *The IDA Pro Book*, S. Huayao and D. Guiju, Eds. Beijing, China: Posts Telecom Press, 2012.
[46] W. Xin, *MATLAB R2014a From Entry to Mastery*. Beijing, China: Publishing House Electronics Industry, 2015.
[47] W. Zhihai, T. Xinhai, and S. Hanhui, *OpenSSL and Network Information Security: Foundation, Structure and Instructions*. Beijing, China: Tsinghua Univ. Press, 2007.
[48] C. Hallinan, *Using BusyBox (Digital Short Cut)*. London, U.K.: Pearson Education, 2006.

XINBING ZHU is currently pursuing the Ph.D. degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include the IoT and information security.

QINGBAO LI is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include information security and trusted computing.

ZHIFENG CHEN is currently pursuing the Ph.D. degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include networks and information security.

GUIMIN ZHANG is currently pursuing the Ph.D. with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include networks and information security.

PENG SHAN is currently pursuing the master's degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include networks and information security.