IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 21, 2019, accepted December 7, 2019, date of publication December 12, 2019,
date of current version December 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2959089

A Firmware Code Gene Extraction Technology for IoT Terminal

XINBING ZHU, QINGBAO LI, PING ZHANG, AND ZHIFENG CHEN
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China
Corresponding author: Xinbing Zhu (catcheverysecond@sina.com)

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

ABSTRACT With the development of the IoT technology, an unprecedented number of IoT terminals are connected to various networks. Commercial-off-the-shelf (COTS) technology is widely used in the IoT terminal firmware, which results in high code reuse rates. Such firmware is always heterogeneous and closed-source. It is so difficult to detect and investigate the security risks at the firmware level that their impacts are faster and broader. In recent years, some firmware security detection technologies based on similarity are gradually becoming a research hotspot. However, in these studies, the basic issue regarding whether these foundations comprise an essential basis for comparison and their utility as similarity measures has not been addressed theoretically. Inspired by biological genes, this paper attempts to supplement a foundation for cross-platform firmware binary code homology and similarity analysis by mining firmware code genes that can essentially identify code and exhibit stability, antivariability and heritability. The firmware code gene extract system(FCGES) is designed and implemented in this paper. FCGES first extracts the features of firmware code, then numericizes and normalizes them, and finally sublimates them to firmware code genes by the hypothesis margin. The experimental results show that the firmware code gene extracted by FCGES has essentiality, stability, antivariability and heritability on different platforms.

INDEX TERMS IoT, IoT terminal, firmware, code gene, similarity, hypothesis margin.

I. INTRODUCTION

With the continuous development of the IoT technology, the global IoT has entered a new round driven by the upgrading of traditional industry and the large-scale consumer market. An unprecedented numbers of IoT terminals are connected to various networks [1], [2]. These terminals have firmware that is heterogeneous and closed-source, which make it difficult to detect and investigate the security risks at the firmware level. And most firmware use commercial-off-the-shelf(COTS) technology; thus, code reuse is more common [3], [4]. In recent years, numerous security incidents regarding IoT terminals have occurred [2], [5], [6]. It can be seen from these incidents the same vulnerabilities or malicious code exists in many products or within multiple manufacturers to varying degrees. Even the long-standing vulnerabilities that have been exposed still do not have remedial measures [2], [3], [7]–[9]. Therefore, the homology and similarity analysis of the firmware code of IoT terminals will be helpful for further research on malicious code detection, vulnerability mining, backdoor discovery and copyright protection.

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

A. MOTIVATION

Due to the firmware is heterogeneous and closed-source, and closely related to the hardware, it is difficult to carry out universal and cross-platform firmware level security detection [3], [4], [7]–[9], [15]–[20]. In recent years, some security detection technologies based on similarity are gradually becoming a research hotspot [15]–[20]. However, in these studies, the basic issue regarding whether these foundations comprise an essential basis for comparison and their utility as similarity measures has not been addressed theoretically. Instead, these foundations are directly applied to the actual detection, which in turn verifies the initial hypothesis. Inspired by biological genes [25], we attempt to address the issue by mining firmware code genes that can essentially identify code and exhibit stability, antivariability and heritability. The goal of this paper is also to extract such firmware code genes and verify their existence, supplementing a basis link for cross-platform firmware binary code homology and similarity analysis.

This paper proposes a firmware code gene extraction technology based on the idea of the hypothesis margin, and a firmware code gene extraction system(FCGES) is designed and implemented. The experimental results show that the firmware code genes extracted by FCGES are intrinsic and exhibit stability, antivariability and heritability.

B. CONTRIBUTIONS

In this paper, we make the following contributions:

(1) We introduce gene theory into the field of firmware. By defining the firmware code genes from two aspects of materiality and informativeness, we solve the dilemma that code genes exist widely in firmware, software and even network space but cannot be accurately described.

(2) We propose a unified and extensible framework for firmware code gene extraction. More importantly, it exposes the essential differences between features and genes.

(3) We design and implement FCGES and verify the existence of the firmware code genes.

(4) Our work provides a theoretical foundation for cross-platform firmware binary code homology and similarity analysis.

C. OUTLINE

The rest part of this paper is structured as follows: Section II shows the related works; Section III introduces firmware code genes and the workflow of FCGES; Section IV proposes a feature extraction method for the firmware code; Section V provides a numericalization and normalization method for the feature values; Section VI sublimates the firmware code features extracted in the previous section to firmware code genes; Section VII presents the experimental steps, scenarios and results, and verifies the existence of the firmware code genes; and Section VIII discusses the deficiencies of this paper and the plans for follow-up works.

II. RELATED WORKS

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

A. DYNAMIC ANALYSIS

For dynamic code analysis techniques [4], [7]–[13], they need to run code in a controlled environment and record the running state and execution results. However, since the firmware is closely related to the hardware, there are generally two ways to achieve dynamic analysis of firmware: one is to carry out on the real hardware; the second is simulation in the virtual environment. *Avatar* [8], [9] performs dynamic analysis by partially offloading execution of firmware to actual hardware. However, running on real hardware is not only expensive, but effective for specific devices, with poor versatility and scalability. *FIRMADYNE* [4] relies on software-based full system emulation with an instrumented kernel to achieve the scalability necessary to analyze thousands of firmware binaries automatically. Danese *et al.* [10] performs full system emulation to achieve the execution of firmware images in a software-only environment, i.e., without involving any physical embedded devices. There are other dynamic studies [11], [12] that simulate the operation of the firmware in different forms. However, the first problem to be solved in dynamic firmware analysis is how to simulate the interaction with the hardware. Even a small interactive simulation failure will cause code running crash, thus the universality and scalability is still not guaranteed.

B. STATIC ANALYSIS

Although static analysis technologies [14]–[24] do not need actually run code, there are several limitations when they are applied to firmware. First, they are mostly aimed at the source level, which is contradictory to the closed-source of IoT terminal firmware [14]. Second, they are often only aimed at a single architecture, especially x86, which is contradictory to the cross-platform deployment of IoT terminal firmware [15]. Third, they have limited capabilities and is often targeted at specific problem domains, such as C, PHP, Java or corresponding binary code, which is contradictory to the fact that IoT terminal firmware is often a mixture [16]. More importantly, static code analysis techniques cannot solve the problem of interaction between 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 [3], [22], [23], similarity analysis based on intermediate representation [17], [18], similarity analysis based on the graph (or tree) structure [19]–[21], [24], similarity analysis based on machine learning [3], [20], [21], etc. However, there are three common questions in these studies: first, whether the attribute information, such as the comparative features and graph structure, is essential, that is, whether the information can identify the code itself in terms of grammar and semantics; second, whether the comparative attribute information is stable and antivariability, that is, whether the information is intrinsic and the code can be identified on different platforms; and third, whether the attribute information compared is heritable, that is, whether it exists stably in the same series or similar code.

There are also some studies on software genes [26]–[29] which try to address the above problems. However, all of these studies have not answered three questions. First, what is the composition of a software gene? The lack of an answer to this question means that they have not completed the mapping of biological gene composition to software space. Thus, when defining a software gene, they only use a ''binary fragment carrying functional information'' [26]–[28]. Second, the difference between a software gene and feature has not been identified; that is, the sublimation of software features to software genes has not been completed, and thus, 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 mostly focus on the search and matching of specific text information rather than numerical calculation.

III. OVERVIEW OF THE FIRMWARE CODE GENE

In this section, after analyzing the composition of biological gene, the structural mapping of the gene concept to firmware is established, and the definition of a firmware code gene is given. Then, we present the workflow of FCGES.

A. BIOGENE

It is well known that the concept of a gene in biology did not exist at the very beginning. From Mendel's discovery of a ''factor'' in pea flowers, which can transfer from generation to generation in a discrete state and determine the external traits or phenotypes of organisms, to Watson, Crick, Wilkins and Franklin's ultimate solution to the mystery of the double helix structure of DNA, more than a hundred years transpired. Even today, the understanding of genes is constantly developing and improving.

A gene refers to a DNA fragment carrying genetic information [25]. DNA is composed of a skeleton of glycosyl and phosphoric acid. Four bases–A, T, G and C–are arranged in accordance with certain rules on the skeleton. A gene is a long DNA fragment that contains other information regarding how to construct proteins. When your body needs a protein, it reads the corresponding nucleotide sequence to synthesize it. The next level is the chromosome. There are 23 pairs of chromosomes in humans. Long strands of DNA are wrapped around a skeleton called histone, which forms the chromosome. Chromosomes contain many genes, which are separated by long segments of DNA.

We can see that a gene is composed of basic structural units, including letters, vocabularies, syntax and grammar. There are only four letters in the ''alphabet'' of a gene. They are the four bases(A, C, G and T). Vocabularies are triplet codes. As shown in Fig. 1, three linked bases can encode an amino acid in a protein. ACT codes threonine, CAT codes histidine, and GGT codes glycine. Proteins are similar to ''sentences'' encoded by a gene and can chain letters together. For example, ACT-CAT-GGT encodes threonine-histidine-glycine. Gene regulation creates rich contexts for these sentences. The regulatory sequence attached to the gene can be considered as the grammar within the gene [25].

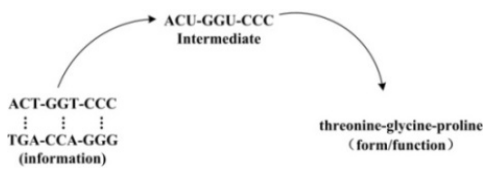

FIGURE 1. The triple cryptographic map.

B. FIRMWARE CODE GENE

Similar to genetic languages in biology, firmware code also has such basic structural units. As shown in Fig. 2, there are only two letters in the ''alphabet'' of firmware code, i.e., 0 and 1. Vocabulary is a byte in an instruction consisting of an octet cipher. There are 256 encoding cases. Instructions are ''sentences'' encoded by a firmware code gene and can chain letters together. In different instruction architectures, instructions have different lengths, compositions and meanings. The control flow of firmware code is the regulation rule of a firmware code gene, which creates rich contextual connotations for these instructions and can also be regarded as the grammar within the firmware code gene.

As we know, not every DNA fragment in biology is a gene. There are many other DNA fragments to separate genes. Only those DNA fragments carrying genetic information are genes. This paper argues that firmware code genes are similar to biological genes.

Definition 1: A *firmware code gene* refers to a binary code fragment that carries genetic information in the firmware code. The fragment satisfies the following properties:

(1) The genetic information contained in such a binary fragment is extractable and representable.

(2) If two or more different binary fragments are generated by the same source code, the genetic information contained in these binary fragments is stable and consistent.

(3) If two or more different binary fragments are generated by similar source codes, the genetic information contained in these binary fragments is also stable and consistent.

As seen from Definition1, Property (1) shows that a firmware code gene has a real physical existence, which reflects the material side. Properties (2) and (3) show that a firmware code gene is intrinsic and stable, which reflects the informative side. In addition, Property (2) also reflects the antivariability, and Property (3) also reflects the heritability. The following sections will focus on the definition and properties to discuss the extraction and verification of the firmware code genes.

However, emphasis on the following is necessary:

(1) Biological genes and firmware code genes belong to different domains. They are two different concepts. Their existence forms and action mechanisms are not exactly the same. There is no one-to-one mapping relationship between them, either in the existence form or in the action mechanism.

(2) The mapping established in this paper is from the point of view that both biological genes and firmware code genes are composed of basic structural units. According to the expression and mechanism of biological genes, the mapping is carried out at the levels of alphabet, vocabulary, sentence and grammar. This structural mapping better reflects the nature, stability, antivariability and heritability of firmware code genes and better supports the similarity and homology analysis than the traditional methods.

C. WORKFLOW

In this paper, FCGES is designed and implemented. As shown in Fig. 3, the system consists of three modules:

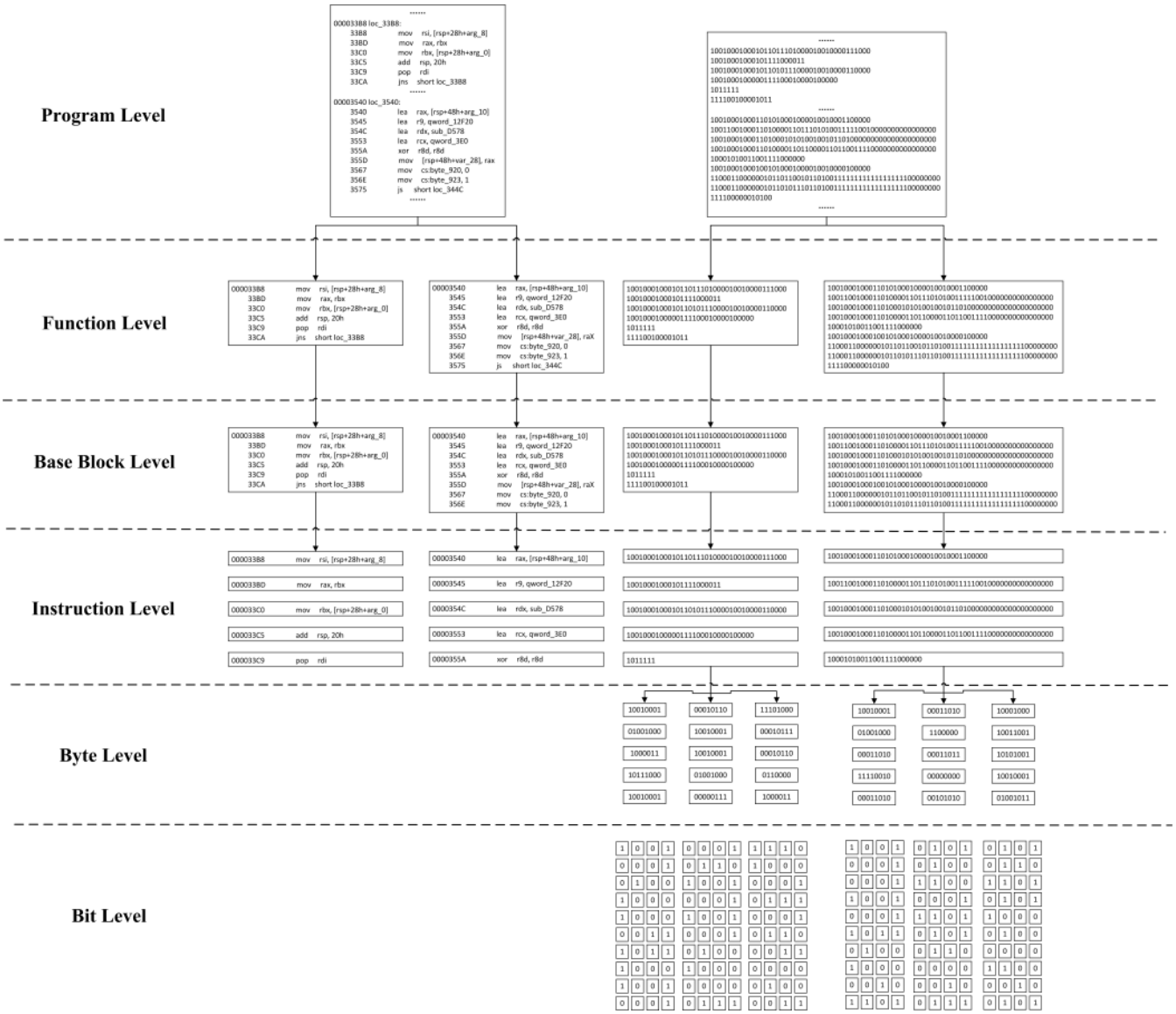

FIGURE 2. The bit-byte-instruction-baseblock-function map.

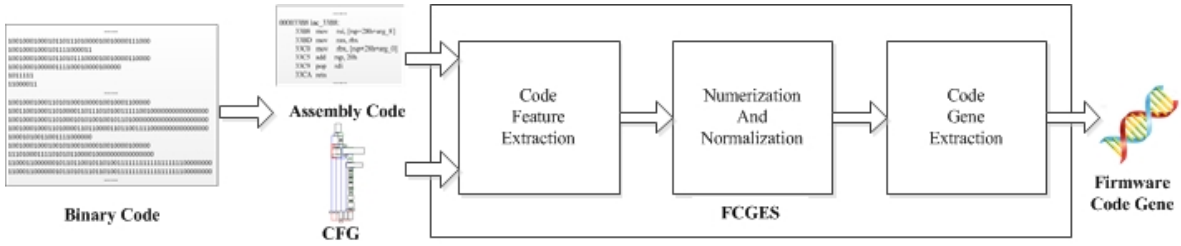

FIGURE 3. The workflow of FCGES.

(1) Firmware code feature extraction module. This part corresponds to Section IV. It mainly completes the extraction of the original features of firmware binary code.

(2) Numericalization and normalization module. This part corresponds to Section V, which mainly deals with the numericalization and normalization of the original feature.

(3) Firmware code gene extraction module. This part corresponds to Section VI. It mainly uses the idea of the hypothesis margin to sublimate the features to the genes.

IV. CROSS-PLATFORM FIRMWARE CODE FEATURE EXTRACTION

In this section, we introduce the differences of three main instruction set architectures(ISA). Then, we specify the heterogeneous instruction sets into five categories according to their basic functions. We also propose a feature extraction method without uniform intermediate representation(IR).

A. IDEAS FOR FEATURE EXTRACTION

ARM, MIPS and x86 processors belong to different ISA. x86 is a typical CISC [30]. MIPS is a typical RISC [31]. As suggested by its name(advanced RISC machine) [32], ARM is an RISC, but also exhibits some advantages of CISC, such as the emergence of multi-register load/store instructions and conditional execution. The instruction sets of these three processors are quite different. Some studies unify the three instruction sets into an IR [17] to bridge the gap between them. However, there are two disadvantages:

(1) It is not easy to unify instruction sets of various architectures into a unified IR. If we want to do so, it is necessary to abstractly unify the instruction set, register organization, addressing mode, stack management, call convention, storage management model and so on. It is difficult to create an abstract unification of these aspects. In fact, the IR often unifies some attributes in an abstract way that incurs some tradeoffs in terms of syntax and grammar, as well as semantics.

(2) Even if the instruction sets of various architectures are expressed as a unified IR, the same code representation in syntax or grammar cannot be obtained. Due to the differences in register organization, addressing mode, calling convention, stack management and so on, instructions of different architectures do not have one-to-one or one-to-more mapping relationships. Consequently, when we translate the code of different architectures into the IR, the intermediate code is also different.

Because of these shortcomings, we do not use the idea of ''binary code–assembly language–IR'', which has appeared in previous studies, but instead directly use ''binary code–assembly language'' to extract firmware code genes. We attempt to design a scientific and reasonable mechanism to evaluate the binary code similarity, which is deployed in different architectures. To achieve this goal, two contradictions need to be addressed: first, Contradiction between the different binary representations and the grammar or syntactic similarity measurement cross-platform; second, Contradiction between the different binary representations and the semantic similarity measurement cross-platform.

However, it is essential that the semantic similarity can faithfully reflect the code similarity, and not all grammatical or syntactic similarity can reflect. This is why many code similarity studies focus on the semantic similarity but not grammatical or syntactic similarity.

According to the function of instructions, we classify the common instructions of these three architectures into five categories: Data Transfer Class Instructions, Arithmetic Operational Class Instructions, Logical Operational Class Instructions, Comparing Test Class Instructions, and Branch Jump Class Instructions. In fact, we have completed a type of ''implicit'' unity without uniform IR. This type of ''implicit'' unity also has certain preferences.

B. CROSS-PLATFORM FIRMWARE CODE FEATURE EXTRACTION

Research has revealed that some basic attribute information and structural attribute information remain stable when the same source code is compiled to different platforms. In particular, we also find that the IoT terminal firmware is strongly indicative of the associated professional business and strongly interacts with the real physical world. Regardless of the platform that the same or similar firmware code is compiled to, some configuration information and specific operations in the code remain stable. Based on these analyses, we extract the basic attribute information of firmware binary code and the structural attribute information of CFG in different architectures without uniform IR.

1) BASIC ATTRIBUTE INFORMATION OF CODE

These information can reflect the code's grammatical or syntactic similarity and even semantic similarity partly.

a: STACK ATTRIBUTE INFORMATION

A stack is often used as a storage structure, regardless of whether there are special stack operation instructions. In this paper, the stack space size *Sta_Spa* and the number of like-stack frame operations *Stafra_Num* are counted.

b: INSTRUCTION NUMBER AND PROPORTION INFORMATION

Instructions are the most basic units of the firmware code. Ranging from every business to every operation, all require a series of instructions. Because of the characteristics of the IoT terminal, the firmware of the same series products or similar functional products may have similar operations. These similarities will eventually be reflected in the instructions. Consequently, the following are counted: the instruction number *Ins_Num,* the data transfer instruction number *MovIns_Num*, the arithmetic operation instruction number *AriIns_Num*, the logic operation instruction number *LogIns_Num*, the comparison test instruction number *CmpIns_Num*, the branch jump instruction number *JumIns_Num*, the data transfer instruction proportion *MovIns_Rat*, the arithmetic operation instruction proportion *AriIns_Rat*, the logic operation instruction proportion *LogIns_Rat*, the comparison test instruction proportion *CmpIns_Rat* and the branch jump instruction proportion *JumIns_Rat*.

c: INSTRUCTION ENTROPY INFORMATION

Same as information entropy, instruction entropy reflects the instruction occurrence probability. Generally, the higher the probability that a class of instructions appears, the higher

the execution probability of the operations represented by such instructions. Consequently, the following are counted: the instruction entropy *Ins_Ept*, the data transmission instruction entropy *MovIns_Ept*, the arithmetic operation instruction entropy *AriIns_Ept*, the logic operation instruction entropy *LogIns_Ept*, the comparative test instruction entropy *CmpIns_Ept* and the branch jump instruction entropy *JumIns_Ept*.

The calculation of instruction entropy is shown in formula (1). P_k is the proportion of the class k instruction.

$$Ins\_Ept = -\sum_{k=1}^{n} P_k \log_2 P_k \tag{1}$$

The specific instruction entropy is calculated according to the above formula and the specific situation. For example, *JumIns_Ept* of the x86 is calculated as in formula (2):

$$JumIns\_Ept = -\sum_{k=1}^{n} P_k \log_2 P_k \tag{2}$$

given that this paper only counts 33 branch jump instructions of x86, $k = 1, 2, 3, \ldots 33$.

d: CONSTANT, VARIABLE, AND STRING INFORMATION

Constants and strings in firmware reflect environmental parameters and business configurations. Variables reflect the process information, such as data processing. They are helpful for analyzing code similarity. In this paper, the string number *Str_Num*, the string set *Str_Set*, the variable number *Var_Num*, the constant number *Con_Num* and the constant set *Con_Set* are counted.

2) STRUCTURAL ATTRIBUTE INFORMATION OF CFG

For code similarity, the semantic is more advantage than the grammar or syntax. Although complete semantic comparison is impossible, approximate equivalence is possible. As the regulation rule of the firmware code gene, the control flow creates rich contextual connotations. This paper argues that the structural attribute information of CFG can reflect the semantic similarity to a certain extent.

a: NODE ATTRIBUTE INFORMATION OF CFG

Each node in CFG is a basic block composed of a sequence of uniformly executed instructions. The control flow information mainly reflects the jump transfer between nodes. Consequently, the following are counted: the node number *Nod_Num*, the average input degree of the node *Indeg_Ave*, the average output degree of the node *Outdeg_Ave*, the average undirected degree of the node *Deg_Ave*, the max input degree of the node *Indeg_Max*, the max output degree of the node *Outdeg_Max*, the max undirected degree of the node *Deg_Max*,the input degree ascending sequence *Indeg_AscLis*, the output degree ascending sequence *Outdeg_AscLis*, and the undirected degree ascending sequence *Deg_AscLis*.

b: OVERALL STRUCTURAL ATTRIBUTE INFORMATION OF CFG

In addition to the jump transfer centered on independent node, the basic block execution sequence in CFG, i.e., edges and paths also reflects the control flow information from a larger perspective. Sometimes they seems to be more advantageous than the node-centered local information in reflecting code similarity. Consequently, the following are counted: the CFG edge number *Edg_Num*, the CFG graph density *Gra_Den*, the CFG undirected graph clustering coefficient *Gra_Clu*, the CFG average path length *PatLen_Ave*, the CFG graph diameter(maximum path length) *PatDia*, the CFG graph link efficiency *PatEff* and the CFG shortest path ascending sequence *Pat_AscLis*.

The calculation of *Gra_Den* is shown in formula (3):

$$Gra\_Den = \frac{2 \times Edge\_Num}{Node\_Num\,(Node\_Num - 1)} \tag{3}$$

The calculation of *Gra_Clu* is shown in formula (4):

$$Gra\_Clu = \frac{1}{Node\_Num} \sum_{k=1}^{Node\_Num} \frac{2c}{d_k\,(d_k - 1)} \tag{4}$$

c denotes the edge number of the undirected CFG subgraph composed of node k and all its neighbor nodes, and d_k denotes the undirected degree of node k.

The calculation of *PatEff* is shown in formula (5):

$$PatEff = \frac{Edg\_Num - PatLen\_Ave}{Edg\_Num} \tag{5}$$

As shown in Table 1, the original firmware code features in different ISA are listed.

TABLE 1. The features of the firmware code.

Name	Type	Name	Type	Name	Type
Sta_Spa	1	*MovIns_Ept*	1	*Outdeg_Ave*	1
Stafra_Num	1	*AriIns_ Ept*	1	*Deg_Ave*	1
Ins_Num	1	*LogIns_ Ept*	1	*Indeg_Max*	1
MovIns_Num	1	*CmpIns_ Ept*	1	*Outdeg_Max*	1
AriIns_Num	1	*JumIns_ Ept*	1	*Deg_Max*	1
LogIns_Num	1	*Str_Num*	1	*Gra_Clu*	1
CmpIns_Num	1	*Str_Set*	2	*PatLen_Ave*	1
JumIns_Num	1	*Var_Num*	1	*PatDia*	1
MovIns_Rat	1	*Con_Num*	1	*PatEff*	1
AriIns_Rat	1	*Con_Set*	2	*Indeg_AscLis*	3
LogIns_Rat	1	*Nod_Num*	1	*Outdeg_AscLis*	3
CmpIns_Rat	1	*Edg_Num*	1	*Deg_AscLis*	3
JumIns_Rat	1	*Gra_Den*	1	*Pat_AscLis*	3
Ins_Ept	1	*Indeg_Ave*	1		
Remarks	numerical type:1; set-valued type:2; sequential type:3.				

V. NUMERICALIZATION AND NORMALIZATION

In this section, we numericize and normalize the feature values. Suppose that the two binary codes to be compared are T_1 and T_2. According to the feature extraction method, the feature vectors V_1 and V_2 aregenerated. Obviously $V_1[i]$ and $V_2[j](1 \leq i, j \leq \mathrm{n})$ may only be of three types: numerical type, set-valued type and sequential type.

A. NUMERICAL TYPE FEATURE

For a numerical feature, we use formula (6) to measure the similarity. It can be seen that in European space, the closer the distance between the two values is, the smaller the absolute value of the difference, and the less obvious the difference in the denominator as the greater of the two is, the smaller the ratio and the higher the similarity. Moreover, the farther the distance between the two values is, the greater the absolute value of the difference, and the more obvious the difference in the denominator as the greater of the two is, the greater the ratio and the lower the similarity. $0 \leq Sim\,(V_1[i], V_2[j]) \leq 1$, C is a constant, and $0 \leq \mathrm{C} \leq 1$.

$$Sim\,(V_1[i], V_2[j]) = \begin{cases} \mathrm{C}, & \text{if } V_1[i] = 0 \text{ and } V_2[j] = 0, \\ 1 - \dfrac{|V_1[i] - V_2[j]|}{max\,(V_1[i], V_2[j])}, & \text{else} \end{cases} \tag{6}$$

B. SET-VALUED TYPE FEATURE

For a set-valued feature, the Jaccard coefficient is used to measure the similarity, and the calculation method is shown in formula (7). It can be seen that the more intersecting elements of two sets there are, the greater the Jaccard coefficient and the higher the similarity; moreover, the fewer intersecting elements of two sets there are, the smaller the Jaccard coefficient and the lower the similarity. $0 \leq Sim\,(V_1[i], V_2[j]) \leq 1$, C is a constant, and $0 \leq \mathrm{C} \leq 1$.

$$Sim\,(V_1[i], V_2[j]) = \begin{cases} \mathrm{C}, & \text{if } V_1[i] = \emptyset \text{ and } V_2[j] = \emptyset, \\ \dfrac{|V_1[i] \cap V_2[j]|}{|V_1[i] \cup V_2[j]|}, & \text{else} \end{cases} \tag{7}$$

C. SEQUENTIAL TYPE FEATURE

For a sequential feature, the longest common subsequence (*LCS*) ratio is used to measure the similarity, and the calculation method is shown in formula (8). Thus, the longer the *LCS* of two sequences is, the greater the ratio of the *LCS* and the higher the similarity; moreover, the shorter the *LCS* of two sequences is, the smaller the ratio of the *LCS* and the lower the similarity. $0 \leq Sim\,(V_1[i], V_2[j]) \leq 1$, C is a constant, and $0 \leq \mathrm{C} \leq 1$.

$$Sim\,(V_1[i], V_2[j]) = \begin{cases} \mathrm{C}, & \text{if } V_1[i] = \text{NULL and } V_2[j] = \text{NULL} \\ \dfrac{|LCS\,(V_1[i], V_2[j])|}{max\,(|V_1[i]|, |V_2[j]|)}, & \text{else} \end{cases} \tag{8}$$

In this paper, the value of C is 0. If the numeric feature is 0, the set-valued feature is Ø, and the sequential feature is NULL, the feature is meaningless to measure similarity. If C is not 0, we measure the similarity with a feature that does not exist, which will obviously distort the overall similarity.

After numericalization and normalization, we shield three differences in each dimension of the feature vector: First, we shield the difference of value meanings so that we do not need to consider what the specific meaning of each feature is. Second, we shield the difference of value types so that we do not need to consider whether it is numerical, set-valued or sequential. Third, we shield the difference of value range so that we do not need to consider which interval value it is or which set or sequence of space it is. More importantly, it provides a general extension method for our future research according to the actual application scenarios without the limitation of the 41-dimensional features in this paper. If there are better features, even if they cannot be expressed by these three data types, as long as we numericize and normalize them, we can still use them in our method.

VI. CROSS-PLATFORM FIRMWARE CODE GENE EXTRACTION

In this section, we use the idea of hypothesis margin [33]–[35] to sublimate the firmware code features to genes.

A. IDEAS FOR GENE EXTRACTION

Although this paper extracts as many features as possible, their importance is different for different code and different application environment.

The firmware code gene extraction algorithm is a feature weighting algorithm that assigns different weights to features according to the correlation of each feature and category. A feature whose weight is less than a certain threshold will be removed; that is, the correlation between features and categories in the algorithm is based on the ability to distinguish features from close samples. The algorithm randomly selects a sample X from training set D and then searches for the nearest neighbor sample from the same class, called *Near-Hit*, and searches for the nearest neighbor sample from the different class, called *Near-Miss*. The weight of each feature is updated according to the following rules: if the distance between X and *Near-Hit* on a feature is less than the distance between X and *Near-Miss*, it indicates that the feature is beneficial for distinguishing the nearest neighbors of different classes and increases the weight of the feature. In contrast, if the distance between X and *Near-Hit* is greater than the distance between X and *Near-Miss*, it indicates that the feature has a negative effect on distinguishing the nearest neighbors of different classes and then reduces the weight of the feature. The above process is repeated p times, and the average weight of each feature is finally obtained. The larger the weight of the feature is, the stronger the classification ability of the feature is.

B. ALGORITHM IMPLEMENTATION

Suppose that a source code S contains m independent functional modules, $S = \{s_1, s_2, \cdots, s_i, \cdots, s_m\}$. Each module may be an independent function or library, or it may be an independent tool or command. T_1 and T_2 are two object codes derived from source code S, where:$T_1 = \{t_{11}, t_{12}, \ldots, t_{1i}, \ldots, t_{1m}\}$, $T_2 = \{t_{21}, t_{22}, \ldots, t_{2j}, \ldots, t_{2m}\}$.

According to the feature extraction method, the original feature vector sets VS_1 and VS_2 are generated,

where: $VS_1 = \{v_{11}, v_{12}, \ldots, v_{1i}, \ldots, v_{1m}\}$, $VS_2 = \{v_{21}, v_{22}, \ldots, v_{2j}, \ldots, v_{2m}\}$.

Each element v_{1i} and v_{2j} in VS_1 and VS_2 is an n-dimensional original feature vector. Then, we use a pair of original feature vectors v_{1i} and v_{2j} in VS_1 and VS_2 to measure their similarity according to the numerical and normalized methods in this paper. The measurement methodis as shown in formula (9):

$$Sim_{ij} = Sim\left(v_{1i}, v_{2j}\right) = \begin{pmatrix} Sim\left(v_{1i}[1], & v_{2j}[1]\right), \ldots, \\ Sim\left(v_{1i}[\mathrm{n}], & v_{2j}[\mathrm{n}]\right) \end{pmatrix} \quad (9)$$

In this manner, we obtain data set D, $D = \{Sim_{ij} \mid 1 \le i, j \le \mathrm{m}\}$. Each sample point in D is an n-dimensional similarity vector Sim_{ij}, which represents the similarity between the object code modules t_{1i} and t_{2j}, and each dimension in Sim_{ij} represents the similarity between the object code modules t_{1i} and t_{2j} in this dimension feature. D is divided into a positive sample set D^+ and a negative sample set D^-: $D^+ = \{Sim_{ij} \mid i = j, 1 \le i, j \le m\}$, $D^- = \{Sim_{ij} \mid i \ne j, 1 \le i, j \le m\}$. Obviously, $D = D^+ \cup D^-$, and $D^+ \cap D^- = \emptyset$. The sampling times is p, and the threshold of correlation statistics is τ. The algorithm detects and identifies those features that are statistically related to the current learning task. When the correlation statistics of these features are larger than τ, they are the effective correlation features.

The algorithm is as follows:

Where $diff\ (x_k, y_k)$ denotes the difference between the corresponding dimension features of sample points X and Y:

$$diff\ (x_k, y_k) = |x_k - y_k|.$$

As seen from the feature update function (17)-(19), the algorithm uses the idea of the hypothesis margin [36]–[38] to evaluate the classification ability of features in each dimension. After several iterations, the more sensitive to the current learning task, the greater the weight is; the more insensitive to the current learning task, the smaller the weight is. Finally, we select those features whose weights are larger than the threshold to generate the most sensitive feature subset, which is the gene vector extracted in this paper.

The firmware code gene extraction algorithm is simple and efficient. Its time complexity is $O(n \times p)$, and it is not affected by the size of the data set [33]–[35]. Even on small sample dataset, features with high correlation can be proposed.

C. APPLICATION SCENARIOS

The goal of this paper is to extract the firmware code genes. Thus, we should apply our method to different scenarios to verify whether the extracted genes have stability, antivariability and heritability.

1) OBJECT CODE GENERATED BY THE SAME SOURCE CODE

Whether the code genes have stability and antivariability will be the key to verifying the correctness of our method.

Suppose that a source code S can be divided into m independent functional modules, $S = \{s_1, s_2, \cdots, s_i, \cdots, s_m\}$. Each module may be an independent function or library or a separate tool or command. We can compile and generate object codes T_1 and T_2 by using different platforms, different tools, or different optimization options, where: $T_1 = \{t_{11}, t_{12}, \ldots, t_{1i}, \ldots, t_{1m}\}$, $T_2 = \{t_{21}, t_{22}, \ldots, t_{2j}, \ldots, t_{2m}\}$. The original feature vector VS_1 and VS_2 are generated by feature extraction. By calculating the feature similarity Sim_{ij} between any two modules, the data set D, D^+ and D^- are generated. Then, we can evaluate the relevant statistics of the features and finally extract the code gene vector.

2) OBJECT CODE GENERATED BY THE SAME SERIES OR SIMILAR SOURCE CODE

Whether the code genes have stability and heritability will also be the key to verifying the correctness of our method.

Suppose that $S_1, S_2, \ldots, S_i, \ldots, S_m$ is a series of similar source codes. We can compile them and obtain the object binary codes $T_1, T_2, \ldots, T_i, \ldots, T_m$. The original feature vector sets are $VS_1, VS_2, \ldots, VS_i, \ldots, VS_m$. By calculating Sim_{ij} of the feature similarity between any two object codes, we generate the data set D, D^+ and D^-. Then, we can evaluate the relevant statistics of the features and finally generate the code gene vector.

VII. EXPERIMENT AND ANALYSIS

In this section, we use the common code in IoT terminal firmware to generate data sets and verify the stability, antivariability and heritability of the firmware code genes extracted by FCGES.

A. EXPERIMENTAL THOUGHTS

In the previous similarity-based studies, there is no direct theoretical answer to whether the basis for comparison is stable, antivariability and heritability in different architecture, but only to verify the initial hypothesis with the practical test results. The purpose of this paper is to extract the firmware code gene with stability, antivariability and heritability, which will complement a basic link for the research in this field. So it is necessary to focus on whether the extracted gene has these characteristics or not. In this paper, We abandon the reverse thinking mode of validating the original theoretical hypothesis through practical test results, and replace it with the simplest and most direct evaluation method; that is, if the object code pair is generated by the same or similar source code, the similarity of each dimension of the gene vector should be at a higher level, and the change cannot be drastic.

To highlight the comparison of the experimental results, the steps are simplified. For Scenario (1), we only consider that the same source code is compiled to different platforms with the same compilation tool and same optimization options; for Scenario (2), we only consider that the same series of source codes is compiled to the same platforms with the same compilation tool and same optimization options. Each experiment is divided into two stages: one is the extraction, and the other is the verification.

Algorithm 1 Firmware Code Gene Extraction Algorithm

Input: Object binary code T_1 and T_2
Sampling times p
Correlation statistics threshold τ

Output: Firmware code gene vector G

(1) Dividing T_1 into m independent modules, $T_1 = \{t_{11}, t_{12}, \ldots, t_{1i}, \ldots, t_{1m}\}$;
(2) Dividing T_2 into m independent modules, $T_2 = \{t_{21}, t_{22}, \ldots, t_{2j}, \ldots, t_{2m}\}$;
(3) Extracting code feature vector set VS_1 from T_1, $VS_1 = \{v_{11}, v_{12}, \ldots, v_{1i}, \ldots, v_{1m}\}$;
(4) Extracting code feature vector set VS_2 from T_2, $VS_2 = \{v_{21}, v_{22}, \ldots, v_{2j}, \ldots, v_{2m}\}$;
(5) Generating feature vector similarity data set D, $D = \{Sim_{ij} \mid 1 \le i, j \le m\}$;
(6) Dividing D into positive sample set D^+ and negative sample set D^-, $D^+ = \{Sim_{ij} \mid i = j, 1 \le i, j \le m\}$, $D^- = \{Sim_{ij} \mid i \ne j, 1 \le i, j \le m\}$;
(7) Initializing the every dimension of the weight vector to zero, $W = (0, 0, \ldots\ldots, 0)$;
(8) For $i = 1$ to p
(9) A sample X, $X \in D$, is randomly selected from data set D;
(10) A positive sample Z^+, $Z^+ \in D^+$, nearest to X is randomly selected from the positive sample set D^+;
(11) A negative sample Z^-, $Z^- \in D^-$, nearest to X is randomly selected from the negative sample set D^-;
(12) if $(X \in D^+)$ then
(13) $Near\_hit = Z^+$; $Near\_miss = Z^-$;
Procedure (14) else
(15) $Near\_hit = Z^-$; $Near\_miss = Z^+$;
(16) Update_weight (W, X, $Near\_hit$, $Near\_miss$);
(17) $Relevance = W/p$;
(18) For $i = 1$ to n
(19) if $(relevance_i \ge \tau)$ then
(20) the dimension i feature is a related feature; $f_i \rightarrow G$;
(21) else
(22) the dimension i feature isn't a related feature;
(23) Return G;

Update_weight(W, X,$Near\_hit$, $Near\_miss$)
(1) For $i = 1$ to n
(2) $w_i = w_i - diff\ (x_i, Near_{hit})^2 + diff\ (x_i, Near\_miss)^2$;

B. EXPERIMENTAL ENVIRONMENT

The experimental environment is as follows. The CPU is an Intel Core i7-6700@ 3.40 GHz, and the memory is 16.0 GB of DDR4 SDRAM. To reduce the impact of multithreading scheduling on the experimental results, the implementation of the gene extraction system adopts a single-threading approach. The binary object code is compiled to x86-32-bit, ARM-32-bit and MIPS-32-bit using GCC v4.6.2. The Python programming language [39] is used to implement FCGES. IDA Pro [40] is used to disassemble binary code and write plug-ins to extract code features. A tool provided by MATLAB R2014b [41] is utilized for feature selection and gene sublimation.

C. STABILITY AND ANTIVARIABILITY OF FIRMWARE CODE GENES

As described in Definition 1, a firmware code gene extracted from the same source code should be stable and resistant to variability, regardless of the form of the binary code; that is, such firmware binary code pairs should have high similarity in each dimension of the code gene vector, and the change cannot be drastic.

1) EXPERIMENT I

In this experiment, the data set is OpenSSL [42] v1.0.0, which is compiled to x86-32-bit, ARM-32-bit and MIPS-32-bit platforms using GCC v4.6.2 with O2. In the firmware code gene extraction stage, two sub-experiments are included. One is the evaluation of the binary code from x86 and ARM, written as x86 × ARM. The other is ARM × MIPS. In each sub-experiment, we can easily distinguish commands, tools and libraries from OpenSSL source code, so we can construct the data set D, positive sample set D^+ and negative sample set D^-. The sampling times $p = 20$, and the weights of each dimension feature in the two sub-experiments are calculated as shown in Table 2. We set the threshold value $\tau = 0.0150$. It can be seen that the weights of the features *Ins_Num*, *MovIns_Num*, *MovIns_Rat*, *AriIns_Rat*, *LogIns_Rat*, *Ins_Ept*, *MovIns_Ept*, *AriIns_Ept*, *LogIns_Ept*, *CmpIns_Ept*, *JumIns_Ept*, *Var_Num*, *Indeg_Ave*, *Outdeg_Ave*, *Gra_Clu*, and *PatEff* are lower than the threshold

TABLE 2. The weights of features in Experiment I.

Name	Weight1	Weight2	Name	Weight1	Weight2	Name	Weight1	Weight2
Sta_Spa	0.0501	0.0503	*MovIns_Ept*	**0.0028**	**0.0032**	*Outdeg_Ave*	**0.0089**	**0.0092**
Stafra_Num	0.0522	0.0527	*AriIns_Ept*	**0.0073**	**0.0096**	*Deg_Ave*	0.0163	0.0197
Ins_Num	**0.0043**	**0.0056**	*LogIns_Ept*	**0.0076**	**0.0099**	*Indeg_Max*	0.0336	0.0380
MovIns_Num	**0.0097**	**0.0101**	*CmpIns_Ept*	**0.0126**	**0.0137**	*Outdeg_Max*	**0.0085**	**0.0099**
AriIns_Num	0.0292	0.0311	*JumIns_Ept*	**0.0127**	**0.0136**	*Deg_Max*	0.0313	0.0351
LogIns_Num	0.0294	0.0341	*Str_Num*	0.0233	0.0246	*Gra_Clu*	**0.0064**	**0.0076**
CmpIns_Num	0.0454	0.0480	*Str_Set*	0.0368	0.0371	*PatLen_Ave*	0.0329	0.0358
JumIns_Num	0.0451	0.0479	*Var_Num*	**0.0072**	**0.0102**	*PatDia*	0.0399	0.0453
MovIns_Rat	**0.0063**	**0.0071**	*Con_Num*	0.0246	0.0256	*PatEff*	**0.0076**	**0.0087**
AriIns_Rat	**0.0102**	**0.0136**	*Con_Set*	0.0378	0.0389	*Indeg_AscLis*	0.0346	0.0391
LogIns_Rat	**0.0100**	**0.0146**	*Nod_Num*	0.0314	0.0399	*Outdeg_AscLis*	0.0221	0.0269
CmpIns_Rat	0.0162	0.0188	*Edg_Num*	0.0321	0.0411	*Deg_AscLis*	0.0298	0.0314
JumIns_Rat	0.0164	0.0187	*Gra_Den*	0.0198	0.0256	*Pat_AscLis*	0.0439	0.0469
Ins_Ept	**0.0049**	**0.0056**	*Indeg_Ave*	**0.0089**	**0.0102**			

value τ in both sub-experiments, and their ability to distinguish close samples is weak and unstable. So we believe that the code genes of OpenSSL v1.0.0 compiled to x86 $\times$ ARM and ARM $\times$ MIPS are composed of the remaining dimensions.

Intuitively, different architectures have different instruction sets, and the number of instructions required to complete the same operation varies. Thus, the ability of *Ins_Num* to reflect code similarity should not be strong. Data transfer class instructions are used widely in various codes, and their ability should be weak. Comparing test class instructions and branch jump class instructions reflects the jump transfer between basic blocks within the code. It has certain semantic information, so their instruction number ability should be relatively strong. However, because the total number of instructions is considerable, the ability of the ratio of these two types of instructions will be reduced. From Table 2, we can see that the results obtained by our method are basically consistent with the intuitive analysis.

In the firmware code gene verification stage, we also carry out two sub-experiments to verify the validity of the method more accurately. In each sub-experiment, we randomly select three samples from D^+ and D^-, and each sample is different. According to the gene vector generated by FCGES, we calculate the similarity of the gene vector in each dimension. The results of x86 $\times$ ARM are shown in Fig. 4, and the results of ARM $\times$ MIPS are shown in Fig. 5. In the two figures, the black curves represent positive samples, and the red curves represent negative samples in the corresponding data set.

From the experimental results, we can see that the similarity curve of each selected sample is more stable if it belongs to D^+, and the value of each dimension changes between 0.7 and 1, whereas the similarity curve of each selected sample that belongs to D^- has intense oscillation and its value is unstable. It can be seen that when the same source code is compiled to different platforms, even though their binary forms have changed greatly, some attributes hidden behind the code still remain stable and antivariable.

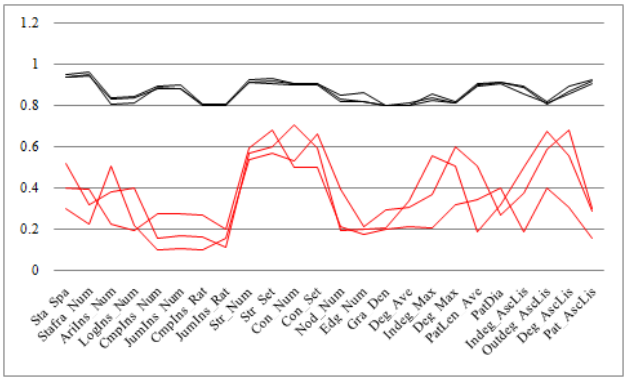

FIGURE 4. The similarity of gene vector in Experiment I-X86 $\times$ ARM.

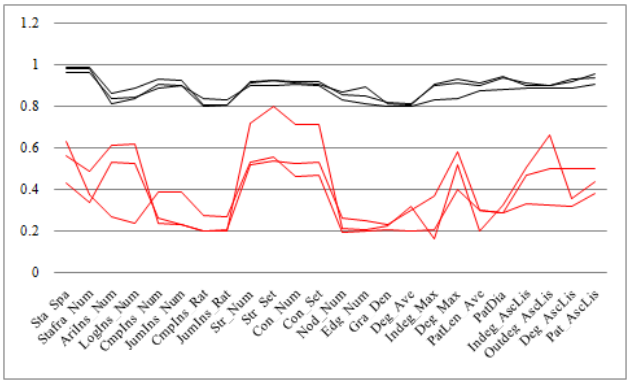

FIGURE 5. The similarity of gene vector in Experiment I-ARM $\times$ MIPS.

2) EXPERIMENT II

In this experiment, the data set is Coreutils [43] v1.0.0, which is compiled to x86-32-bit, ARM-32-bit and MIPS-32-bit platforms using GCC v4.6.2 with O2. In the firmware code gene extraction stage, two sub-experiments are also included. One is x86 $\times$ MIPS. The other is ARM $\times$ MIPS. In each sub-experiment, we randomly select 30 common commands from Coreutils to construct data set D, positive sample set D^+ and negative sample set D^-. The sampling times $p = 20$ and the weights of each dimension feature in the two sub-experiments

TABLE 3. The weights of features in Experiment II.

Name	Weight3	Weight4	Name	Weight3	Weight4	Name	Weight3	Weight4
Sta_Spa	0.0503	0.0504	*MovIns_Ept*	**0.0031**	**0.0033**	*Outdeg_Ave*	**0.0091**	**0.0092**
Stafra_Num	0.0520	0.0523	*AriIns_Ept*	**0.0046**	**0.0043**	*Deg_Ave*	0.0169	0.0199
Ins_Num	**0.0044**	**0.0046**	*LogIns_Ept*	**0.0041**	**0.0049**	*Indeg_Max*	0.0341	0.0378
MovIns_Num	**0.0087**	**0.0091**	*CmpIns_Ept*	**0.0125**	**0.0133**	*Outdeg_Max*	**0.0083**	**0.0089**
AriIns_Num	**0.0101**	**0.0102**	*JumIns_Ept*	**0.0129**	**0.0137**	*Deg_Max*	0.0319	0.0346
LogIns_Num	**0.0102**	**0.0101**	*Str_Num*	0.0256	0.0255	*Gra_Clu*	**0.0071**	**0.0078**
CmpIns_Num	0.0461	0.0476	*Str_Set*	0.0370	0.0376	*PatLen_Ave*	0.0341	0.0362
JumIns_Num	0.0459	0.0462	*Var_Num*	**0.0087**	**0.0095**	*PatDia*	0.0411	0.0454
MovIns_Rat	**0.0064**	**0.0068**	*Con_Num*	0.0253	0.0261	*PatEff*	**0.0077**	**0.0081**
AriIns_Rat	**0.0082**	**0.0083**	*Con_Set*	0.0381	0.0390	*Indeg_AscLis*	0.0354	0.0385
LogIns_Rat	**0.0085**	**0.0083**	*Nod_Num*	0.0387	0.0399	*Outdeg_AscLis*	0.0226	0.0251
CmpIns_Rat	0.0167	0.0178	*Edg_Num*	0.0396	0.0407	*Deg_AscLis*	0.0302	0.0319
JumIns_Rat	0.0163	0.0179	*Gra_Den*	0.0212	0.0253	*Pat_AscLis*	0.0468	0.0476
Ins_Ept	**0.0050**	**0.0053**	*Indeg_Ave*	**0.0091**	**0.0100**			

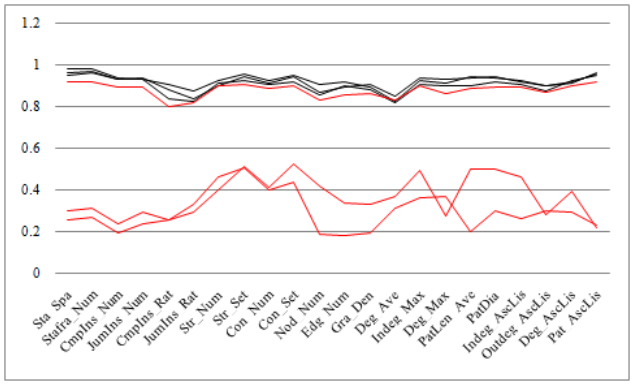

FIGURE 6. The similarity of gene vector in Experiment II-X86 × MIPS.

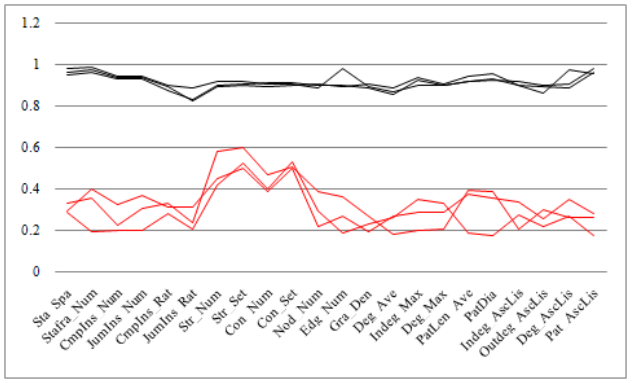

FIGURE 7. The similarity of gene vector in Experiment II-ARM × MIPS.

are calculated as shown in Table 3. We also set the threshold value $\tau = 0.0150$. It can be seen that the weights of *Ins_Num*, *MovIns_Num*, *MovIns_Rat*, *AriIns_Num*, *AriIns_Rat*, *LogIns_Num*, *LogIns_Rat*, *Ins_Ept*, *MovIns_Ept*, *AriIns_Ept*, *LogIns_Ept*, *CmpIns_Ept*, *JumIns_Ept*, *Var_Num*, *Indeg_Ave*, *Outdeg_Ave*, *Gra_Clu* and *PatEff* are lower than τ in both sub-experiments. Their ability to distinguish close samples is weak and unstable. Thus, we believe that the firmware code genes of Coreutilsv1.0.0 compiled to x86 × ARM and ARM × MIPS with GCC v4.6.2 are composed of the remaining dimensions.

From Table 3, it can be seen that, in this experiment, *AriIns_Num* and *LogIns_Num* are eliminated because their weights are lower than τ. This may be related to the different functions between OpenSSL and Coreutils. OpenSSL is an open-source software library package, which includes three parts: the SSL protocol library, application program and cryptographic algorithm library. The code contains a large number of arithmetic and logic operations, so its gene vector contains these two dimensions. Coreutils is a software package under GNU, which contains common commands of Linux. Compared with OpenSSL, the number of arithmetic and logic operations is not much different from other code.

In the firmware code gene verification state, Experiment II also includes two sub-experiments, as in Experiment I. The results of x86 × MIPS are shown in Fig. 6, and the results of the ARM × MIPS are shown in Fig. 7.

Unfortunately, the curves in Fig. 6 are not as perfect as those in Fig. 4, Fig. 5 and Fig. 7. In Fig. 6, a red curve also shows the stability that a black curve should have; that is, the experimental results of a negative sample in the data set show the same performance as that of a positive sample. After analysis, we found that this sample is generated by the commands ls and dir. In addition, ls and dir have very similar functions. However, in contrast, this false positive proves the effectiveness of our method.

D. STABILITY AND HERITABILITY OF FIRMWARE CODE GENES

As described in Definition 1, the firmware code gene extracted from similar or identical series of source code should be stable and heritable, regardless of the form of the binary code; that is, such firmware binary code pairs should have high similarity in all dimensions of the code gene vector, and the changes cannot be drastic.

1) EXPERIMENT III

In this experiment, the data set is OpenSSL v1.0.0 and v1.0.1, which are compiled to ARM-32 using GCC v4.6.2 with O2. In the firmware code gene extraction stage, we construct data set D, positive sample set D^+ and negative sample set D^- as in Experiment I. The sampling times $p = 20$, and the weights of each dimension feature are calculated, as shown in Table 4. We also set the threshold value $\tau = 0.0150$. It can be seen the weights of *Ins_Num*, *MovIns_Num*, *MovIns_Rat*,

TABLE 4. The weights of features in Experiment III.

Name	Weight5	Name	Weight5	Name	Weight5
Sta_Spa	0.0502	*MovIns_Ept*	**0.0035**	*Outdeg_Ave*	**0.0093**
Stafra_Num	0.0523	*AriIns_Ept*	**0.0092**	*Deg_Ave*	0.0199
Ins_Num	**0.0057**	*LogIns_Ept*	**0.0094**	*Indeg_Max*	0.0382
MovIns_Num	**0.0100**	*CmpIns_Ept*	**0.0133**	*Outdeg_Max*	**0.0095**
AriIns_Num	0.0310	*JumIns_Ept*	**0.0138**	*Deg_Max*	0.0353
LogIns_Num	0.0329	*Str_Num*	0.0249	*Gra_Clu*	**0.0079**
CmpIns_Num	0.0471	*Str_Set*	0.0373	*PatLen_Ave*	0.0359
JumIns_Num	0.0480	*Var_Num*	**0.0097**	*PatDia*	0.0453
MovIns_Rat	**0.0069**	*Con_Num*	0.0257	*PatEff*	**0.0081**
AriIns_Rat	**0.0134**	*Con_Set*	0.0392	*Indeg_AscLis*	0.0392
LogIns_Rat	**0.0139**	*Nod_Num*	0.0403	*Outdeg_AscLis*	0.0263
CmpIns_Rat	0.0187	*Edg_Num*	0.0409	*Deg_AscLis*	0.0319
JumIns_Rat	0.0191	*Gra_Den*	0.0255	*Pat_AscLis*	0.0476
Ins_Ept	**0.0055**	*Indeg_Ave*	**0.0099**		

AriInstruc_Rat, *LogIns_Rat*, *Ins_Ept*, *MovIns_Ept*, *AriIns_Ept*, *LogIns_Ept*, *CmpIns_Ept*, *JumIns_Ept*, *Var_Num*, *Indeg_Ave*, *Outdeg_Ave*, *Gra_Clu*, and *PatEff* are lower than τ, and their ability to distinguish close samples is weak and unstable. Thus, we believe that the firmware code genes of OpenSSL v1.0.0 and v1.0.1 compiled on ARM-32-bit with GCC v4.6.2 are composed of the remaining dimensions.

OpenSSL v1.0.0 and v1.0.1 are two very close versions, and their corresponding codes are very similar. Table 4 confirms our conjecture. It can be seen that the dimension of the gene vector extracted in this experiment is the same as that in Experiment I. This unification of horizontal and vertical comparison also proves the correctness of our method from another aspect.

In the firmware code gene verification stage, we carry out two sub-experiments as in the previous experiment to make the experiment more rigorous. The experimental results are shown in Fig. 8.

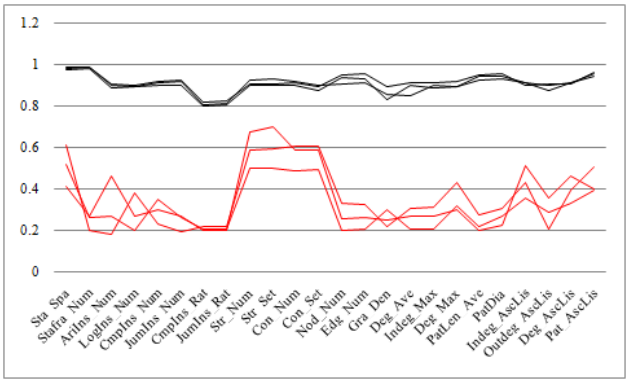

FIGURE 8. The similarity of gene vector in Experiment III.

The experimental curves are similar to those in Experiment I. It can be seen that when the same series of source codes are compiled to the same platform, even though their binary representation is different, some attribute information hidden behind the code remains stable and heritable.

2) EXPERIMENT IV

In this experiment, the data set used is BusyBox [44] v1.20.0 and v1.21.1, which are compiled to the x86 platform with GCC v4.6.2. In the firmware code gene extraction stage, we also select 30 common commands from BusyBox v1.20.0 and v1.21.1 for analysis and construct data set D, positive sample set D^+ and negative sample set D^-. The sampling times $p = 20$, and the weights of each dimension feature are calculated as shown in Table 5. We also set the threshold value $\tau = 0.0150$. It can be seen that the weights of *Ins_Num*, *MovIns_Num*, *MovIns_Rat*, *AriIns_Num*, *AriIns_Rat*, *LogIns_Num*, *LogIns_Rat*, *Ins_Ept*, *MovIns_Ept*, *AriIns_Ept*, *LogIns_Ept*, *CmpIns_Ept*, *JumIns_Ept*, *Var_Num*, *Indeg_Ave*, *Outdeg_Ave*, *Gra_Clu* and *PatEff* are lower than τ, and their ability to distinguish close samples is weak and unstable. Thus, we believe that the firmware code genes of BusyBoxv1.20.0 and v1.21.1 compiled on x86-32-bit with GCC v4.6.2 are composed of the remaining dimensions.

The version distance between BusyBox v1.20.0 and v1.21.1 is obviously larger than that of Experiment III. As seen from Table 5, the firmware code genes extracted are the same as those extracted from Experiment III, except for the two dimensions of *AriIns_Num* and *LogIns_Num*. The relevant reasons have been analyzed in Experiment II. In addition, synthesizing the results of four experiments, we also find that, although the data sets and instruction architectures are different, the extracted firmware code genes by our method overlap in many dimensions.

TABLE 5. The weights of features in Experiment IV.

Name	Weight6	Name	Weight6	Name	Weight6
Sta_Spa	0.0507	*MovIns_Ept*	**0.0033**	*Outdeg_Ave*	**0.0091**
Stafra_Num	0.0519	*AriIns_Ept*	**0.0032**	*Deg_Ave*	0.0196
Ins_Num	**0.0059**	*LogIns_Ept*	**0.0031**	*Indeg_Max*	0.0371
MovIns_Num	**0.0102**	*CmpIns_Ept*	**0.0133**	*Outdeg_Max*	**0.0098**
AriIns_Num	**0.0102**	*JumIns_Ept*	**0.0133**	*Deg_Max*	0.0350
LogIns_Num	**0.0101**	*Str_Num*	0.0245	*Gra_Clu*	**0.0075**
CmpIns_Num	0.0478	*Str_Set*	0.0370	*PatLen_Ave*	0.0357
JumIns_Num	0.0476	*Var_Num*	**0.0101**	*PatDia*	0.0451
MovIns_Rat	**0.0072**	*Con_Num*	0.0253	*PatEff*	**0.0086**
AriIns_Rat	**0.0071**	*Con_Set*	0.0372	*Indeg_AscLis*	0.0389
LogIns_Rat	**0.0072**	*Nod_Num*	0.0401	*Outdeg_AscLis*	0.0268
CmpIns_Rat	0.0182	*Edg_Num*	0.0413	*Deg_AscLis*	0.0312
JumIns_Rat	0.0181	*Gra_Den*	0.0249	*Pat_AscLis*	0.0467
Ins_Ept	**0.0058**	*Indeg_Ave*	**0.0101**		

In the firmware code gene verification stage, as in the previous experiments, we calculated the similarity of the gene vector in each dimension. The results are shown in Fig. 9. To avoid the situation in Experiment II, we try to choose commands that have different functions when constructing the data set in this experiment. From the experimental results, we can see that the similarity curves of positive samples and negative samples are consistent with our expectations. It can be seen that when the same series of source code is compiled to the same platform, even though the binary representations are different, some attribute information hidden behind the code remains stable and heritable.

E. ESSENTIALITY OF FIRMWARE CODE GENES

In fact, the four experiments have explained the essentiality of firmware code genes to some extent, because stability, antivariability and heritability are inherent manifestations. However, this essentiality is premised on the following:

(1) The firmware code genes in this paper are sublimated from the original feature vector, that is, whether the original

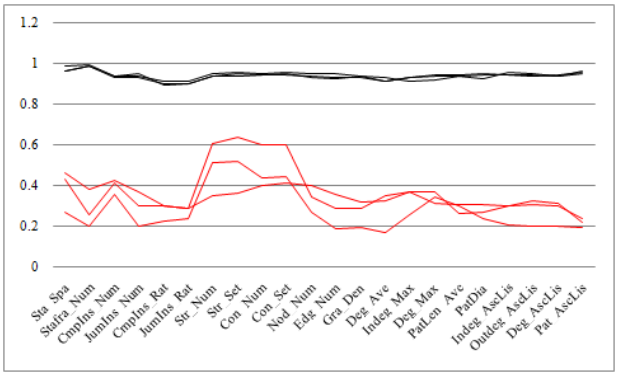

FIGURE 9. The similarity of gene vector in Experiment IV.

feature vector is complete or not will directly determine the quality of the genes. If the high-weight features are not taken into account in the feature extraction stage, the quality of the genesis certainly not high, or perhaps even the gene cannot be generated. This is why we should extract as many features as possible in the feature extraction section.

(2) The firmware code genes in this paper are also related to the threshold value τ. If τ is set too low, some low-weight features will be included. These features are weak in reflecting the nature of the code, have poor stability and low resistance to variability and inheritance, and may be misreported when identifying the code. If τ is set too high, some features that are able to reflect the stability, antivariability and heritability will be excluded. It is likely that omissions will occur. According to our research on the security detection of IoT terminals based on firmware code genes, this paper sets $\tau = 0.015$.

(3) The firmware code gene extraction algorithm cannot eliminate redundant features, so there may be redundant components in our gene vector. Redundant dimensions need to be avoided in the process of extracting original features.

Although the data sets and platforms of the four experiments are different, the gene vectors obtained through our system overlap in many dimensions. This not only further validates the essence of the extracted genes but also shows that some gene dimensions have a common ability to distinguish close samples at the binary code level in different source codes and different platforms. This makes it possible to analyze the similarity and homology of binary code across platforms based on firmware code genes.

VIII. SUMMARY

According to the heterogeneous, closed-source and high-code-reuse-rate characteristics of IoT terminal firmware, this paper proposes a firmware code gene extraction technology based on the idea of the hypothesis margin. The experimental results show that the firmware code genes extracted by FCGES are intrinsic, stable, antivariable and heritable. However, the related research work is still far from perfect, and some work needs to be completed in future research.

(1) To highlight the experimental purpose, the experimental process was simplified. For example, without considering the impact of compiler tools and optimization options. Thus, the similarity curves of the experiments seem perfect. However, in the practice of IoT terminal security detection based on the firmware code gene, we find that the experimental results are not as perfect as we imagine. This is mainly due to different compilers, different optimization options, tailorable operating environments, inline functions, dynamic link libraries and other reasons, but firmware code genes still show superiority over traditional similarity studies. The purpose of this paper is to reveal the existence of the firmware code genes rather than security detection based on firmware code genes; therefore security detection has exceeded the scope of this article, related content will be discussed in the following articles.

(2) Although this paper proposes a technical framework for extracting firmware code genes of IoT terminal firmware, this framework can extract firmware genes from binary code with different granularities. In the initial steps of original feature extraction, only 41-dimensional features, such as the basic attribute information of firmware code and structural attribute information of CFG, are extracted. In fact, for different firmware and different application scenarios, these 41 features may not be perfect. For example, the call graph information reflecting the function call relationship is not extracted in this paper. It need to be continuously improved in future research.

(3) The object of this study is firmware binary code, but to highlight the discussion of firmware code genes, the ''binary code'' is the binary code that has been unpacked, not the firmware binary image before unpacking.

(4) We have also performed some research on the security detection of IoT terminal firmware based on firmware code genes. However, due to space limitations, this paper has focused on the introduction of firmware code gene extraction technology, mainly for complementing the basic link for the firmware security research based on similarity theory. Research on the security detection of IoT terminal firmware based on firmware code genes will be introduced in subsequent papers.

REFERENCES

[1] *White Paper on Internet of Things Security*, China Inst. Inf. Commun., Beijing, China, 2018.
[2] (2018). *In-Depth Analysis Report on Internet of Things Security and Applications*. [Online]. Available: https://wenku.baidu.com/view/40b64e8329ea81c758f5f61fb7360b4c2e3f2ad8.html
[3] 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.
[4] D. D. Chen, M. Woo, and D. Brumley, ''Towards automated dynamic analysis for linux-based embedded firmware,'' in *Proc. NDSS*. 2016, pp. 1–16.
[5] *Thoughts on a Vulnerability in Industrial Control System*. Accessed: May 7, 2019. [Online]. Available: https://mp.weixin.qq.com/s/LZnvDQ9lSqgfd8LvKKgteA
[6] *Talking About the Safety of Camera*. Accessed: May 7, 2019. [Online]. Available: https://mp.weixin.qq.com/s/xY6W-zq2dzgeH4N6t6-ouQ
[7] J. Zaddach, L. Bruno, and A. Francillon, ''AVATAR: A framework to support dynamic security analysis of embedded systems' Firmwares,'' in *Proc. NDSS*, 2014, pp. 1–16.
[8] M. Muench, D. Nisi, and A. Francillon, ''Avatar 2: A multi-target orchestration platform,'' in *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.)*, Feb. 2018, Art. no. 18.

[9] A. Costin, A. Zarras, A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded Web interfaces," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 437–448.
[10] A. Danese, G. Pravadelli, and V. Bertacco, "Work-in-Progress: DOVE: Pinpointing firmware security vulnerabilities via symbolic control flow assertion mining," in *Proc. Int. Conf. Hardw./Softw. Code Sign Syst. Synth.*, 2017, pp. 1–2.
[11] G. Hernandez, F. Fowze, and D. J. Tian, "Firmusb: Vetting USB device firmware using domain informed symbolic execution," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2245–2262.
[12] L. Sha, F. Xiao, and W. Chen, "IIoT-SIDefender: Detecting and defense against the sensitive information leakage in industry IoT," *World Wide Web*, vol. 21, no. 1, pp. 59–88, 2018.
[13] O. Or-Meir, N. Nissim, L. Rokach, and Y. Elovici, "Dynamic malware analysis in the modern era—A state of the art survey," *ACM Comput. Surv.*, vol. 52, no. 5, 2019, Art. no. 88.
[14] F. Gauthier, T. Lavoie, and E. Merlo, "Uncovering access control weaknesses and flaws with security-discordant software clones," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2013, pp. 209–218.
[15] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.
[16] A. Costin, J. Zaddach, and A. Francillon, "A large-scale analysis of the security of embedded firmwares," in *Proc. Usenix Secur. Symp.*, 2014, pp. 95–110.
[17] J. Pewny, B. Garmany, and R. Gawlik, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.
[18] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2017, pp. 79–94.
[19] Q. Feng, R. Zhou, and C. Xu, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 480–491.
[20] D. Zhao, H. Lin, and L. Ran, "CVSkSA: Cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph," *Softw. Qual. J.*, vol. 27, pp. 1–24, Feb. 2019.
[21] X. Xu, C. Liu, and Q. Feng, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.
[22] 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.
[23] 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.
[24] H. Huang, A. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proc. ACM Asia Conf. Comput. Commun. Secur. (ASIACCS)*. ACM Press, 2017, pp. 155–166.
[25] S. Mukherjee, *The Gene: An Intimate History*. Beijing, China: CITIC Press Corporation, 2018.
[26] H. Jin *et al.*, "Detection and classification of Android malware based on malware gene," *Appl. Res. Comput.*, vol. 36, no. 6, pp. 1813–1818, 2019.
[27] *Written After the Sub-Forum of 'Software Genetics Technology' (I)*. Accessed: Jun. 7, 2019. [Online]. Available: https://www.sohu.com/a/228476725_468696
[28] *Written After the Sub-Forum of 'Software Genetics Technology' (II)*. Accessed: Jun. 7, 2019. [Online]. Available: http://www.sohu.com/a/228946546_468696
[29] *CCF Held Seminar on Software Gene in ZhengZhou*. Accessed: Jun. 7, 2019. [Online]. Available: https://www.ccf.org.cn/c/2018-12-06/657463.shtml
[30] K. R. Irvine and L. B. Das, *Assembly Language for X86 Processors*. Upper Saddle River, NJ, USA: Prentice-Hall, 2011.
[31] D. Sweetman, *See MIPS Run Linux*. Amsterdam, The Netherlands: Elsevier, 2010.
[32] G. Lei, *Development of Embedded Linux System Based on ARM*. Beijing, China: Tsinghua Univ. Press, 2014.
[33] K. Kira and L. A. Rendell, "The feature selection problem: Traditional methods and a new algorithm," in *Proc. AAAI*, San Jose, CA, USA, vol. 2, Jul. 1992, pp. 129–134.
[34] Z. Zhihua, *Machine Learning*. Beijing, China: Tsinghua Univ. Press, 2016.
[35] M. Robnik-Sikonja and I. Kononenko, "Theoretical and empirical analysis of ReliefF and RReliefF," *Mach. Learn.*, vol. 53, nos. 1–2, pp. 23–69, 2003.
[36] R. Gilad-Bachrach, A. Navot, and N. Tishby, "Margin based feature selection-theory and algorithms," in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, Art. no. 43.
[37] M. Yang, F. Wang, and P. Yang, "A novel feature selection algorithm based on hypothesis-margin," *JCP*, vol. 3, no. 12, pp. 27–34, 2008.
[38] M. Yang and P. Yang, "Hypothesis-margin model incorporating structure information for feature selection," in *Proc. 2nd IEEE Int. Symp. Electron. Commerce Secur.*, May 2009, pp. 634–639.
[39] 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.
[40] C. Eagle, *The IDA Pro Book*. Beijing, China: Posts and Telecom Press, 2012.
[41] W. Xin, *MATLAB R2014a From Entry to Mastery*. Beijing, China: Publishing House of Electronics Industry, 2015.
[42] W. Zhihai, T. Xinhai, and S. Hanhui, *OpenSSL and Network Information Security: Foundation, Structure and Instructions*. Beijing, China: Tsinghua Univ. Press, 2007.
[43] *Coreutils-GNU Core Utilities*. Accessed: Aug. 25, 2019. [Online]. Available: http://www.gnu.org/software/coreutils
[44] C. Hallinan, *Using BusyBox (Digital Short Cut)*. London, U.K.: Pearson, 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.

PING ZHANG is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. Her research interests include information security and advanced 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.