Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Malware homology identification based on a gene perspective*

Bing-lin ZHAO[†1], Zheng SHAN[†‡1], Fu-dong LIU[†1], Bo ZHAO[1,2], Yi-hang CHEN[1], Wen-jie SUN[1]

[1]*State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China*

[2]*Georg-August-University of Goettingen, Goettingen 37077, Germany*

[†]E-mail: gzu_zhaobl@126.com; zzzhengming@163.com; lwfydy@126.com

Received Aug. 31, 2018; Revision accepted Jan. 5, 2019; Crosschecked June 11, 2019

Abstract: Malware homology identification is important in attacking event tracing, emergency response scheme generation, and event trend prediction. Current malware homology identification methods still rely on manual analysis, which is inefficient and cannot respond quickly to the outbreak of attack events. In response to these problems, we propose a new malware homology identification method from a gene perspective. A malware gene is represented by the subgraph, which can describe the homology of malware families. We extract the key subgraph from the function dependency graph as the malware gene by selecting the key application programming interface (API) and using the community partition algorithm. Then, we encode the gene and design a frequent subgraph mining algorithm to find the common genes between malware families. Finally, we use the family genes to guide the identification of malware based on homology. We evaluate our method with a public dataset, and the experiment results show that the accuracy of malware classification reaches 97% with high efficiency.

Key words: Malware classification; Gene perspective; Dependency graph; Homology analysis
https://doi.org/10.1631/FITEE.1800523 **CLC number:** TP309.5

1 Introduction

In recent years, the dramatic increase of malware has become a serious threat to the Internet. In May 2017, the outbreak of the WannaCry virus and its variants hit more than 100 countries around the world and a large amount of institutions were paralyzed (Qihoo 360, 2017). An in-depth analysis of a large amount of malware shows that many new malware samples are variants of previously known malware. Malware writers obtain new variants by extending the function of the old code, or changing the binary feature with code obfuscation. With the expansion of open source technology and the development of modular malware generation tools, the problems of increased quantity and complexity of malware have become more serious (Yu et al., 2018). Therefore, there is an urgent need for a fast and efficient homology analysis method for malware. At present, the homology analysis of malware depends mainly on the experience of experts (Qiao et al., 2016). Expert analyses provide accurate results, but rely heavily on manual analysis, which cannot respond quickly to event outbreaks.

In genetic research, although the gene sequences of each organism are different, there are homologous gene sequences among different organisms of the same species. The species and homology can be determined by calculating the similarity of different gene sequences. Like gene sequences in organisms, there exist some stable substructures that can be copied between different malware samples of the same family. Consequently, we can study the genetic stability of high-level semantic structure in the samples of the same malware family. Mining the feature with genetic stability in a malware family can guide malware classification and homological analysis.

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 61472447 and 61802435)

ORCID: Bing-lin ZHAO, http://orcid.org/0000-0001-5948-9195

In this paper, we propose a new malware homology identification method based on a genetic perspective. In our method, the malware gene is defined as the subgraph of the function dependency graph (FDG). First, we select the key application programming interface (API) of different malware families by calculating the importance of each API from different malware families. Then the FDG is divided into different subgraphs, which contain only the path that connects to the key API. We then use the community partition algorithm to divide the subgraph into the key subgraphs. The key subgraph is defined as the malware gene. Finally, we encode the malware gene by graph compression. On this basis, we design an algorithm for frequent subgraph mining. The algorithm takes the compressed graph of the same family as the comparison object to mine the representative frequent subgraph, which will be considered as the gene of the malware family. The contributions of this paper are summarized as follows:

1. We first use the graph structure to represent the malware gene and use the frequent subgraph of the malware family to represent the family common gene. The malware family gene is genetically stable in the malware family, which can effectively guide the classification and homological analysis of malware.

2. We propose a frequent subgraph mining algorithm based on a compressed graph, which compresses the graph that ensures the functional integrity of the subgraph. The subgraph mining algorithm ignores the small differences in different malicious code in implementing the same behavior, and improves the efficiency of mining frequent subgraphs and the efficiency of comparison.

3. We propose a malware homological analysis system based on a genetic perspective and conduct an experiment with the Kaggle Microsoft dataset. The evolution results show that our approach is highly efficient in malware classification with 97% accuracy.

2 Related work

According to different research objects, there are two types of research on Internet security, research on information flow in a network and research on malicious code in terminals.

With the rapid development of the software defined network (SDN), research on SDN security that uses information flow as the object has been widely conducted. Wu et al. (2018a) proposed a security authentication scheme for cluster management based on big data analysis in the SDN. The scheme ensures the legality of the data sources and improves the performance of applications running in an SDN. In addition, Wu et al. (2018b) presented a systematic virtual networking architecture to perform global virtualization control and monitoring of a cyber-physical system, in which network function virtualization configuration and orchestration can be realized.

Research on the security of information flow in the network has achieved excellent results. However, most malicious code uses terminals as the target, which threatens the privacy and personal information of the terminal users. Therefore, security research on malicious code, such as malware detection and malware homological analysis, is also very important.

At present, the homological analysis of malicious code is done mainly by calculating the distance between malware features. Cesare et al. (2013) proposed Malwise to unpack and classify malware. Malwise uses information entropy to find packed malware and a fast program simulator to unpack it. Then, the control flow of malicious code is converted to the character, and the classification of malicious code is determined by calculating the distance between characters. Alam et al. (2014) proposed a method that uses Malware Analysis Intermediate Language (MAIL) (Alam et al., 2013) as the intermediate representation of malware to annotate the information in the control flow graph (CFG) and defined behavior schemas. On this basis, they built a real-time malware detection system, which can effectively detect the same subgraph by matching different patterns. Jang et al. (2014) established the system call graph for different malware, and applied social network algorithms to mine the attributes of the graph. The category of malware is determined by analyzing the properties and structure of the system call graph. Liu et al. (2017) proposed a malware classification method that uses grayscale images of malicious code, an n-gram of opcode sequences, and import tables as the features. The method uses a machine learning algorithm to determine the distance between samples by calculating the feature distance between different samples.

The above methods consider the similarity calculation of the features, but they do not guide the homological analysis using the similarity of the malware family from viewpoints such as population and genetics. At present, there is little research on malware analysis based on a gene perspective, and there is no unified definition of a malware gene.

Kirat and Vigna (2015) proposed MalGene, which uses the evasion signature represented by system call sequences as the malware gene. MalGene leverages bioinformatics algorithms to automatically locate evasive behavior. Drew et al. (2016) extracted the gene sequence from the malware binary file and established a gene classifier using Strand, a high-throughput multi-sequence comparison algorithm, to process the gene sequences. Han et al. (2018) used subsequences of critical operations in Android malware as gene sequences to analyze the homology of Android malware.

Using sequence as the gene representation, we can describe the inheritance and replication of behavior between malware samples of the same family. There are many ready-made sequence comparison algorithms and tools that are used in genetic research, but there are many differences between software and organisms. The sequence represents only a single path of behavior but loses the overall characteristics of behavior and the correlations among different functional regions. To describe the malicious code information more accurately, we use the graph structure to represent the malware gene.

3 Malware homological analysis based on the gene perspective

In this study, we propose a homological malware analysis method from the gene perspective. The overall architecture of the method is shown in Fig. 1, which is composed of five stages and two operational processes:

1. Preprocessing: The malicious code is disassembled to establish the FDG.

2. Gene selection: This stage analyzes the key API and finds the community relationship in the FDG to extract the key subgraph as the malware gene.

3. Gene encoding: This stage converts the gene into a compression graph, and then encodes the compression graph as gene encoding.

4. Family gene mining: This stage finds the frequent subgraph, which is treated as the common gene of the malware family.

5. Gene comparison: Genes are compared with those of different families in the gene database to determine the malware family.

3.1 Preprocessing

In this stage, we first disassemble the malicious code and build the function call graph (FCG) from the assembler code. Then, we transform the FCG to the FDG.

3.1.1 FCG and FDG

The FCG represents the function call relationship in the malware binary executable. A call graph is

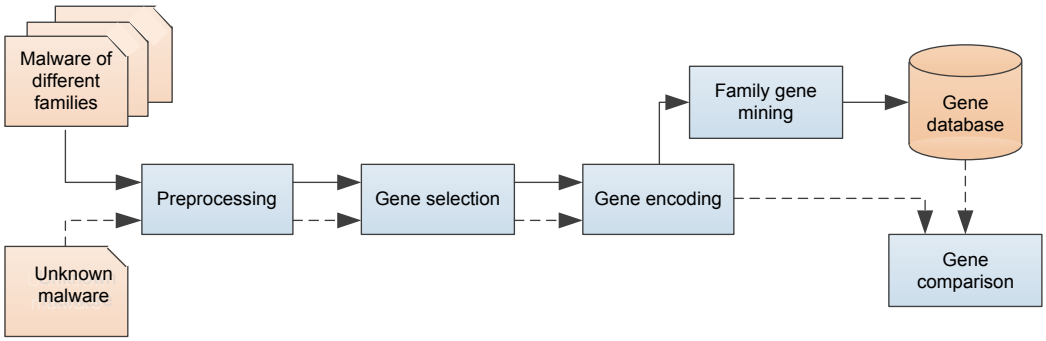

Fig. 1 Architecture of the method

The solid lines represent gene database construction and the dashed lines represent homological analysis based on gene comparison

a directed graph whose vertices represent the functions in the executable, and edges represent the call relationship between functions (Kinable and Kostakis, 2011). Fig. 2 shows an example of an FCG.

In Fig. 2, the rectangular nodes represent the local functions and the elliptical nodes are the external functions. Local functions are written by the malware writer, and are contained in the malware binary file. External functions, such as system calls and library calls, are stored in operating system libraries.

The FCG represents the hierarchical relationship between function calls, but the order of execution among different functions at the same level is not described. For example, in Fig. 2 the external functions WriteFile and CloseHandle are called by the local function Sub_407328; moreover, we do not know the execution order of the two external functions. Therefore, we transform the FCG to the FDG to describe the function execution order and the behavior more accurately. FDG is constructed by the function execution order, which is obtained by static analysis of the binary executable. Fig. 3 shows the intermediate representation of a transformation process, which preserves the function call edges and adds the function dependency edges at the same level.

Because the names of local functions, which are written by the malware writers, are not retained during software compilation, reverse software assigns the same symbol prefix "Sub_" and a random yet unique symbol to represent the local function name. Moreover, unlike the external function, we cannot associate the name of a local function with its feature, such as behavior and function. Therefore, we simplify it by reducing local functions to make the FDG of the malware contain only the description of behavior. Fig. 4 shows the FDG represented by the sequence of the execution order between functions, which contains only external functions. The definition of an FDG is given as follows:

Definition 1 (Function dependency graph) A function dependency graph is a directed graph G with vertex set $V=\{v_1, v_2, \ldots, v_n\}$, representing the functions of malware, and edge set $E=\{<v_i, v_j>|v_i, v_j \in V\}$, in correspondence to the function dependencies between the functions of malware execution.

3.1.2 FDG generation

FDG is generated from the binary executable using static disassembly. First, we use PEiD and ExeinfoPe to check the binary file to determine whether it is a packed program. For the packed program, we

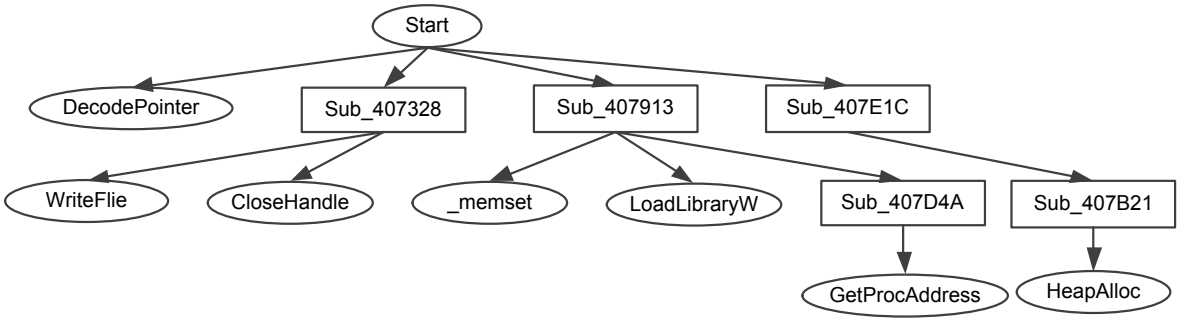

Fig. 2 Example of the call graph

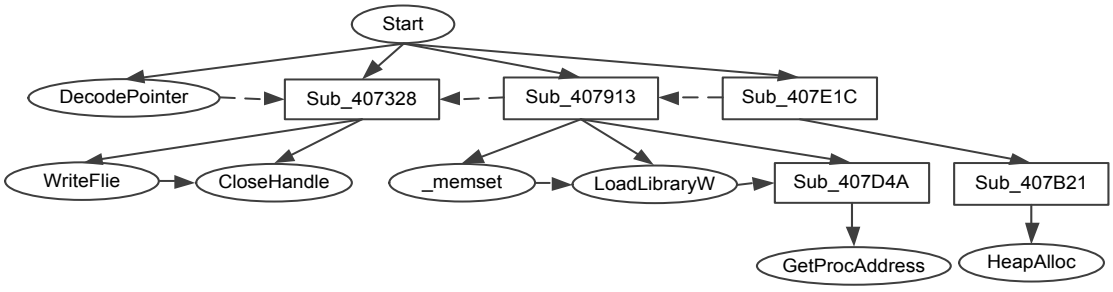

Fig. 3 Function call graph with function dependency
The dashed line indicates the execution order of the functions at the same level

use the general unpacking tool to decrypt it. After that, a disassembly tool such as interactive disassembler (IDA) is used to disassemble the binary code to get the assembler codes. The disassembly procedure identifies the function boundaries and calling relationships. For local functions, IDA assigns the prefixed "Sub_." On the other hand, for external functions, the dynamically imported library functions are named according to the import address table (IAT) in the PE header, while the static linked library functions can identify their original names using the IDA fast library identification and recognition technology (FLIRT). After a disassembly procedure, the binary codes are transformed to the assembler codes.

We build the FCG by analyzing the calling relation and functions of the assembler codes. We obtain the FDG by reducing the local function of the FCG. First, we add the edge of the execution order between the adjacent function in the FCG by scanning the function execution order from the assembler codes to obtain the intermediate graph (Fig. 3). Then, we scan the intermediate graph to find the local function and add the edge between the previous node of sequential execution of the local function and the first calling node of the local function. After that, we reduce the local function from the intermediate graph. The FDG is obtained by repeating the above steps (Fig. 4).

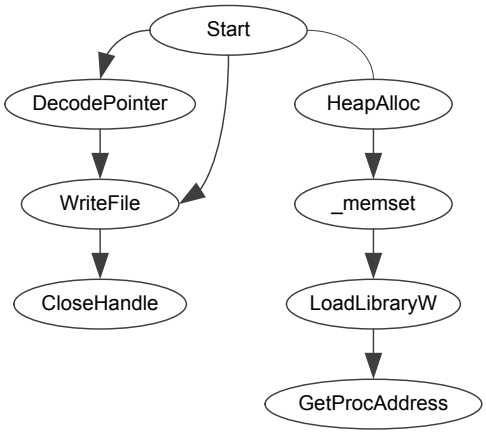

Fig. 4 Example of function dependency graph

3.2 Gene selection

After preprocessing, the FDG contains many nodes, but not all the behaviors represented by the nodes contribute to malicious behavior. Moreover, graph matching is an NP-complete problem. If the graph matching algorithm is used to compare the FDG, the computational efficiency will decrease sharply with the increase of the graph scale. According to Naval et al. (2017), there exist several paths in any graph that carry almost all the information of the graph. Consequently, the FDG has some key areas represented by the subgraph, which cover almost all the critical behaviors of the malicious code.

In an FDG, the related behaviors, which have strong connectivity and always exist in the same class or same function, are used together to achieve a common goal. Therefore, the key subgraph is taken as the comparison object, which not only reduces the scale of graph comparison but also maintains the integrity of the behavior. In addition, using the key subgraphs as the malware genes can represent the replication and inheritance relationship among the behaviors of the malware samples in the same family. The malware gene is defined as follows:

Definition 2 (Malware gene) Malware gene is a subgraph structure with genetic stability in a malware family. The malware gene can represent common behavior and code reuse of different malware samples, which shows the strong similarity between functions and behaviors of malicious code in the same malware family.

Definition 3 (Malware genome) Malware genome is the sum of all genetic information contained in a single malicious code, which is a collection of malware genes.

The malware gene selection process consists of two parts. First, the key API of the malicious code is selected. Then the subgraph, which contains the critical key API, is divided into communities. Compared with the FDG, the key subgraph greatly reduces the scale and preserves behavior integrity. The gene selection process is shown in Fig. 5.

In Fig. 5, there are three components, which represent the process of gene selection, and each node

represents an API of an FDG. The process from Fig. 5a to Fig. 5b represents key API selection and the process from Fig. 5b to Fig. 5c denotes community partition. In Fig. 5b, the gray nodes represent the key API of the FDG. The other nodes represent the non-critical API, which are not contained in the critical malware behaviors. After community partition, the FDG in Fig. 5b is divided into five communities,

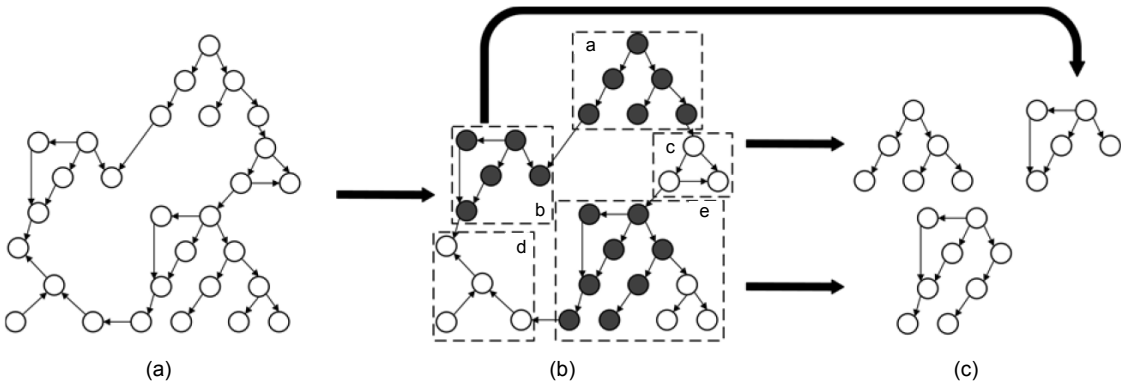

Fig. 5 Gene selection process: (a) FDG; (b) FDG with key API and communites; (c) malware gene

which are represented by the dashed line. From the communities, we select the three communities that have key APIs to build the malware genes (Fig. 5c).

3.2.1 Key API selection

By screening the APIs which are important for malicious behavior and unique to the family of samples, the subgraphs containing these APIs are used as the key areas of analysis. These key areas represented by the subgraph cover almost all the critical behaviors of malicious code. In this study, term frequency–inverse document frequency (TF-IDF) is used to select the key API. By selecting the contribution of different APIs to the sample behavior of different families, the key APIs of different families are selected. The TF-IDF is as follows:

$$\mathrm{tfidf}_{i,j}=\frac{n_{i,j}}{\sum_k n_{k,j}}\log\frac{|D|}{\left|\left\{j:t_i\in d_j\right\}\right|+1}, \tag{1}$$

where $n_{i,j}$ represents the total number of samples containing API i in sample family j, $\sum_k n_{k,j}$ represents the total sample number of sample family j, $|D|$ is the total number of all samples, $|\{j:t_i\in d_j\}|$ represents the number of samples containing API i.

3.2.2 Community detection of key area

After the key API is selected, we remove the edges that are not connected with key API nodes, and obtain the key area of the FDG, which contains only the key API. Based on the key area, the malware gene represented by the key subgraph is obtained by dividing the community from the key area.

Because of the low computational complexity and remarkable result of graph node classification, we use the theory of graph convolution network (GCN) (Kipf and Welling, 2016) to divide the key area of the FDG into different communities. The GCN uses a similar generator as effective embedding mechanisms for graph signals, which is applied to semi-supervised community detection. The GCN experiments on a well-known graph dataset achieve outstanding results.

The foundation of the GCN is to use spectral theory to realize convolution operation on a graph. The essential operator in spectral analysis of the graph is symmetric normalized Laplacian:

$$\boldsymbol{L}=\boldsymbol{I}_R-\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}. \tag{2}$$

In Eq. (2), $\boldsymbol{A}\in\mathbb{R}^{R\times R}$ is the adjacency matrix associated with graph G, D is the diagonal degree matrix, and $\boldsymbol{I}_\mathrm{R}$ is the identity matrix. $\boldsymbol{L}$ is transformed by $\boldsymbol{L}=\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\mathrm{T}$, where $\boldsymbol{U}$ is the matrix of eigenvectors and $\boldsymbol{\Lambda}$ is the diagonal matrix of eigenvalues. Therefore, the graph is defined as signal $\boldsymbol{x}$, which represents the eigenvector of a graph. The graph Fourier transform of signal $\boldsymbol{x}$ can be expressed as $\boldsymbol{x}=\boldsymbol{U}^\mathrm{T}\hat{x}$. It defines convolution on a graph as a multiplication between signal x and the filter $\boldsymbol{g}_\theta$=diag($\boldsymbol{\theta}$) in the spectral domain:

$$\boldsymbol{g}_\theta\,\boldsymbol{x}=\boldsymbol{U}\,\boldsymbol{g}_\theta\,\boldsymbol{U}^\mathrm{T}\,\boldsymbol{x}, \tag{3}$$

where, $\boldsymbol{\theta}\in\mathbb{R}^R$ is a Flourier coefficient vector, and $\boldsymbol{g}_\theta=\boldsymbol{g}_\theta(\boldsymbol{\Lambda})$ can be regarded as the function of the eigenvalue of $\boldsymbol{L}$. However, in Eq. (3) each convolution

calculation needs to calculate the product between $\boldsymbol{U}$, $\boldsymbol{g}_\theta$, and $\boldsymbol{U}^{\mathrm{T}}$, and its computational complexity is $O(n^2)$. It will take much time, when the scale of the graph is large. To reduce the computational complexity, Defferrard et al. (2016) used the truncation expansion of k-order Chebyshev polynomials to approximate the formula $\boldsymbol{g}_{\theta'}$. The approximation of the formula $\boldsymbol{g}_{\theta'}$ is as follows:

$$\boldsymbol{g}_{\theta'}\boldsymbol{x}=\sum_{k=0}^{K}\boldsymbol{\theta}'_k T_k(\tilde{\boldsymbol{L}})\boldsymbol{x}. \tag{4}$$

In Eq. (4), $\tilde{\boldsymbol{L}}=2\boldsymbol{L}/\lambda_{\max}-\boldsymbol{I}_R$, $\lambda_{\max}$ represents the largest eigenvalue of $\boldsymbol{L}$, and $\boldsymbol{\theta}'\in\mathbb{R}^R$ represents a Chebyshev coefficient vector. The Chebyshev polynomials are defined as

$$T_k(\boldsymbol{x})=2\boldsymbol{x}T_{k-1}(\boldsymbol{x})-T_{k-2}(\boldsymbol{x}). \tag{5}$$

In Eq. (5), $T_0(\boldsymbol{x})=1$ and $T_1(\boldsymbol{x})=\boldsymbol{x}$. In addition, the filter operation of signal $\boldsymbol{x}$ in the K domain filter can be deduced according to $\boldsymbol{L}^2=\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{\mathrm{T}}\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{\mathrm{T}}=\boldsymbol{U}\boldsymbol{\Lambda}^2\boldsymbol{U}^{\mathrm{T}}$ and $\boldsymbol{U}\boldsymbol{U}^{\mathrm{T}}=E$:

$$\boldsymbol{g}_{\theta'}(\boldsymbol{L})\boldsymbol{x}\approx\sum_{k=0}^{K}\boldsymbol{\theta}'_k T_k(\tilde{\boldsymbol{L}})\boldsymbol{x}. \tag{6}$$

The graph convolution layer in the GCN is defined as

$$y_{\mathrm{output}}=\sigma\left(\sum_{k=0}^{K}\boldsymbol{\theta}'_k T_k(\tilde{\boldsymbol{L}})\boldsymbol{x}\right). \tag{7}$$

In Eq. (7), the ReLU function is used for the activation function $\sigma(\cdot)$. After optimization, the expression of convolution calculation becomes a Laplacian k-order polynomial, which is k-localized. So, the convolution computation depends only on the node of order k, which greatly improves the computational efficiency of community mining and node classification.

3.3 Gene encoding

The malware gene defined in this study is represented by a graph structure. To compare genes more quickly and store them more efficiently, we design a gene encoding method to encode the malware gene. Data encoding is done mainly to transform the information from one format to another. The encoding process eliminates redundancy and reduces the storage space effectively, and is based on preserving the integrity of information. In this study, the malware gene is encoded by extracting the information of the compressed graph of the gene. The compressed graph maintains the behavior stability of the malware gene and neglects the subtle differences in the implementation of malware behavior. In addition, the integrity of the behavior contour can be preserved with the edge weight. The gene compression is shown in Fig. 6.

In Fig. 6, we can see the malware gene compression process represented by the subgraph. From Fig. 6a to Fig. 6b, we first traverse the binary dependency edge of the input gene to build the binary dependency edge list. Then, we count the binary dependency edge from the list to obtain the count number of each edge. After that, we obtain the compression graph, and the two statuses are shown as Figs. 6c and 6d. In the compression graph, the weight of each edge represents the count number of each binary dependency edge. The graph compression algorithm is shown in Algorithm 1.

In Algorithm 1, by traversing the binary dependency edge of malware gene G, which is

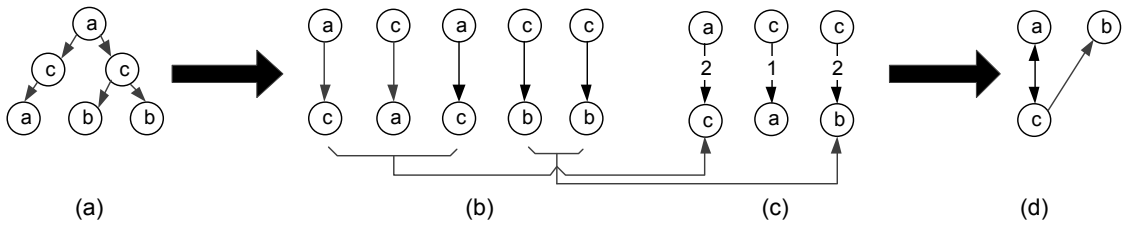

Fig. 6 Gene compression: (a) binary dependency edge; (b) binary dependency edge list; (c−d) compression graph with two statuses

represented by a graph, the binary dependency edge is extracted from the graph, and the dependency set Dep_set is obtained. After that, the binary dependency edge is inserted into the compressed graph of gene G'. In a compressed graph, each binary dependency edge appears only once, and the weight value of each dependency edge represents the number of times that the binary dependency appears in graph G.

Algorithm 1 Gene compression

Input: Malware gene G
Output: Compressed graph of gene G'

```
Dep_set={}
  for e in G
    if e in Dep_set
      Dep_set[e]=Dep_set[e]+1
    else
      Add e to G'
      Dep_set[e]=1
    end if
  end for
```

Because each binary dependency relation in the compressed graph appears only once, the compressed graph can be expressed as a set of binary dependency relations. Therefore, the comparison between compressed graphs can be transformed into set comparison, which greatly reduces the computational complexity and improves the efficiency of comparison. For each binary dependency in the set of the compressed graphs, we give a unique number to encode it. Fig. 7 shows a malware gene example and Table 1 lists the encoding of binary dependency, which appears in each gene of Fig. 7. The malware gene encoding results (Fig. 7) are shown in Table 2.

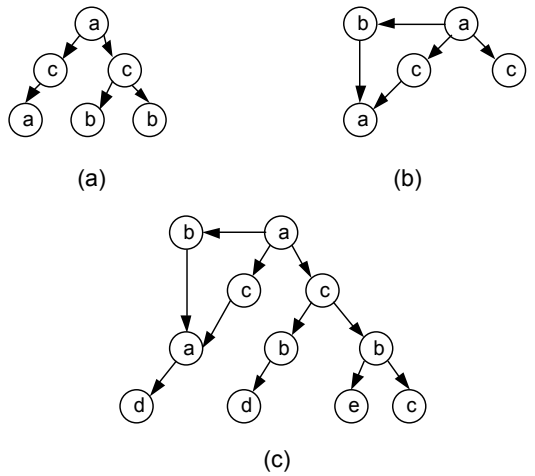

Fig. 7 Gene representation: (a) gene A; (b) gene B; (c) gene C

Table 1 Encoding of binary dependency

Binary dependency	Encoding result
a→c	0
c→a	1
c→b	2
b→d	3
a→b	4
b→e	5
b→a	6
a→d	7
d→b	8

From Table 2, we can see that the method of gene encoding in this study uses a smaller substructure to describe the malware gene, which not only ensures the basic integrity of gene information, but also reduces the storage space effectively. After encoding, gene comparison can be transformed into set comparison, which effectively improves the efficiency of comparison.

Table 2 Gene encoding

Gene	Encoding results
A	0, 1, 2
B	0, 1, 2, 4, 6
C	0, 1, 2, 3, 4, 5, 7, 8

3.4 Family gene mining

Malware genes represent features that are widespread and genetically stable in the malware family. To determine the homology of unknown malicious code, it is necessary to extract family genes from different malware families and establish the malware family gene database. To establish the gene database, we first extract the genes from the malware samples in the dataset, and then select the family genes of different malware families by mining their common features.

In this study, the subgraph of FDG is used to represent the malware gene. We can use the frequent subgraph mining algorithm to mine the frequent subgraph of the malware family as the common family gene. Because of gene encoding, the gene represented by the graph is transformed into set

representation. The process of frequent subgraph mining with a compressed graph can avoid a lot of graph iteration and comparison, which improves the efficiency of subgraph mining. Frequent subgraph mining based on the compressed graph is shown in Algorithm 2.

Algorithm 2 Frequent subgraph mining based on the compressed graph

Input: Compressed graph set of the same malware family H, threshold of frequency β

Output: Frequent compressed graph set of the same malware family H'

```
1  SubGraphSet={}
2  GraphSet={}
3  Frequent_graph=[]
4  cage_num=|H|
5    for sample in H
6      for g in sample
7        add g to GraphSet
8      end for
9    end for
10   for g′ in GraphSet
11     for g″ in GraphSet
12       mixg=g′∩g″
13       if mixg in Frequent_graph
14             Frequent_graph[mixg]=
               Frequent_graph[mixg]+1
15       else
16             add mixg to SubGraphSet
17             Frequent_graph[mixg]=1
18       end if
19     end for
20   end for
21   for mixg′ in SubGraphSet
22     Graph_fre=Frequent_graph[mixg′]/cage_num
23     if graph_fre>β
24       add mixg′ to H′
25     end if
26   end for
```

The algorithm takes the set H of the compressed graph of each sample as the input and the frequent subgraph set H' as the output. Frequent_graph is an array that represents the number of times a potential subgraph appears in the malware family. cage_num represents the compressed graph number in the malware family. First we traverse each sample of the same family and merge all subgraphs of each sample into a total graph set GraphSet (line 7). Then we calculate the occurrence number of each common subgraph between the different subgraphs in the total set Frequent_graph (lines 8 to 15). After that, we calculate the frequency of each subgraph and select the subgraph whose frequency exceeds the threshold β to insert the subgraph into the frequent subgraph set H'. The threshold β is obtained by synthesizing the classification results and the scale of the gene database. We will discuss this in the experiment. Finally, we obtain the frequent subgraph set as the family gene of each malware family and save it to the gene database.

3.5 Gene comparison

When unknown malware is analyzed in our method, we first select its key API. Because the family of malware is unknown, we select the key API of each malware family from the training dataset, which is computed by TF-IDF, to form the key API set for the unknown malware key API selection. If the API of the unknown malware is included in the key API set, it will be selected as the key API of the unknown malware.

After that, we divide the FDG and key API of unknown malicious code into communities, and the subgraph obtained from community partition is used as the candidate gene. Then, the candidate genes are compressed and encoded to obtain the genomes. Finally, by comparing the unknown malware genome and the common genes of different malware families, the family of the unknown malware is determined by selecting the most similar results. In this study, the Jaccard coefficient is used to describe the similarity of set comparison:

$$\text{Gene\_sim}=\frac{\text{MG}(i)\cap\text{FG}(j)}{\text{MG}(i)\cup\text{FG}(j)}. \tag{8}$$

In Eq. (8), the ratio of the intersection to union of the candidate genome of the target code MG(i) and the common gene set of the malicious code family FG(j) is calculated as the result of similarity.

4 Experiments

In this section, we apply our method to a large collection of real-world datasets and evaluate its performance in terms of effectiveness and efficiency.

The experimental environment used in this experiment is Windows Server 2012, the CPU is Intel Core® i7-7500 2.70 GHz, and the memory is 16.0 GB. We use Python 3.5 as the programming language.

4.1 Datasets

4.1.1 VxHeaven dataset

In this study, the malware dataset is obtained from VxHeaven and the Kaggle website. For each sample in the VxHeaven dataset, we first scan and unpack it. However, there are many samples that cannot be unpacked with the general unpacking tools. Because unpacking is not the main focus in this study, we remove the unpacked samples from the dataset. After that, we obtain 45 658 PE format samples to construct the new VxHeaven dataset.

We disassemble all the samples and extract the high-level semantic structure from the assembler code, which is represented by the FDG. Table 3 shows the sample distribution of different families of the VxHeaven dataset. The sample families are analyzed according to the name from the Kaspersky security software. As shown in Table 3, the dataset has 4310 malware families, among which 88 malware families each contain more than 100 samples.

Table 3 Distribution of samples from different families

Sample number of malware families	Number of families
1	2725
2	560
2−5	473
5−10	217
10−20	112
20−50	93
50−100	42
>100	88

4.1.2 Kaggle malware classification dataset

In February 2015, Microsoft announced the Malware Classification Challenge to the Kaggle data science community. Microsoft has provided almost 500 GB of data for the challenge, which contain 10 868 types of labeled malware in training data and 10 783 unlabeled malware in testing data. In the training data and testing data, there are nine malware families and each malware file has a 20-character hash value, which uniquely identifies the file. Each file in the dataset contains a binary file with the extension ".byte" and a corresponding disassembled file with the extension ".asm" in the assembly language.

However, the label of the testing data has not been published on the Internet; therefore, we can use only the training data in our experiment. The count number of each family in the training data is shown in Fig. 8 and the corresponding information in Table 4.

It can be seen from Fig. 8 that the number of samples in the fifth family is small. To make the results more objective, we remove the fifth family from the dataset.

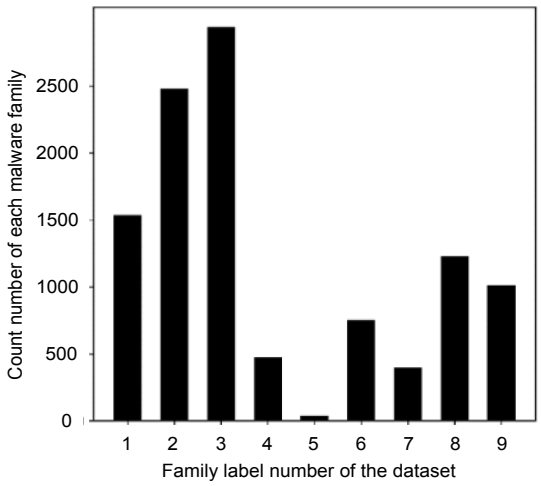

Fig. 8 Sample num of the Kaggle dataset

Table 4 Information of the Kaggle dataset

Family number	Family name	File count
1	Ramnit	1541
2	Lollipop	2478
3	Kelihos_ver3	2942
4	Vundo	475
5	Simda	42
6	Tracur	751
7	Kelihos_ver1	398
8	Obfuscator.ACY	1128
9	Gatak	1013

4.2 Malware gene extraction

In this subsection, we use the Kaggle dataset as object and describe the extraction of the malware gene in detail. Because the Kaggle dataset contains the ".asm" file, which is represented by assembler code, we can omit the disassembly process. We first convert the ".asm" file to an FDG, which is described

Table 5 Key API distribution

Family number	Key API
1	LeaveCriticalSection, memcpy, errno, dllonexit, strlen, getenv, fputc, VirtualProtect, MultiByteToWindeChar, IsDBCSLeadByteEx, …
2	GetModuleHandleA, strncmp, strtol, memset, VirtualFree, HeapFree, GetModuleFileNameA, Encode_pointer, Decode_pointer, encoded_null strlen, …
3	LoadLibraryExA, IstrlenW, SetFocus, LocalUnlock, LocalFree, IsWindow, IsDebuggerPresent, GlobalLock GetStdHandle, GetModuleHandleA, …
4	GetPixel, GetRgnBox, GetObjectA, GdiGetBatchLimit, GetProcessHeap, CreateMutexW, GetLocalTime, GetModuleHandLea, GetAcp, …
6	HeapFree, CloseHandle, GetProcAddress, LstrlenA, GetWindowsDirectoryA, GetVolumeInformation, HeapAlloc, Process32First, srand, OpenProcess, …
7	Sleep, GetUserDefaultLangID, CreateMutexA, ReleaseMutex, UnmapViewOfFile GetVersion, OpenMutexA, GetVersion, GetOEMCP, GetLastError, …
8	MultiByteToWindeChar, GetLastError, WideCharToMultiByte, CloseHandle, VirtualAlloc, SetFilePointer, LCMapStringW, HeapReAlloc, HeapFree, HeapAlloc, …
9	glGetError, glHint, glDrawArrays, glDeleteTextures, glClearStencil, VirtualAlloc, GetCurrentProcessId, stoptraceA, ProcessTrace, QueryTraceA, …

in Section 3.5. Then, we evaluate the importance of the API in different families by calculating the TF-IDF of different APIs. We rank the importance of the top 75% of the APIs in different families as the key APIs that are obtained through multiple experiments, to ensure low correlation in genes in different families and the classification accuracy. Table 5 lists some key APIs in different malware families.

From Table 5, we can see that the key APIs of different families have some differences, which shows that the classification method based on API is helpful in classifying malicious code. Then we remove the FDG edges that are not connected to the key API, and obtain the malware gene by dividing it into communities. After that, we give a unique number to each binary gene dependency from different families and use that number to encode the malware gene. Finally, we use Algorithm 2 to mine the encoded frequent genes in different malware families as the malware family gene and establish the gene database. Table 6 shows the number of genes in different malware families and the corresponding number of binary dependencies.

At the same time, we compare the similarity of binary dependencies in different malware families to obtain the similarity rate of genes between different malware families.

In Table 7, we can see that there are some common genes in different malware families, which indicates that these malware families have similar behavior. On the other hand, most of the family genes are unique to the family, playing an important role in homological detection and analysis.

Table 6 Gene number distribution

Family number	Family gene number	Binary dependency number
1	9	344
2	6	181
3	14	538
4	10	10
6	13	384
7	7	68
8	37	989
9	6	47

Table 7 Similarity of gene between malware families

Family number	Similarity of gene							
	1	2	3	4	6	7	8	9
1	1.00	0.00	0.00	0.00	0.01	0.02	0.01	0.02
2	0.02	1.00	0.01	0.00	0.03	0.00	0.04	0.00
3	0.00	0.01	1.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
6	0.01	0.03	0.00	0.00	1.00	0.00	0.13	0.00
7	0.02	0.00	0.00	0.00	0.00	1.00	0.00	0.00
8	0.01	0.04	0.00	0.00	0.13	0.00	1.00	0.10
9	0.02	0.00	0.00	0.00	0.00	0.00	0.10	1.00

Next, we analyze threshold β. By adjusting the threshold value, the gene similarity rate between different malware families and the classification results are recorded. The gene similarity rate is calculated by computing the average value Table 7, except the value of the diagonal, and the classification procedures are

described in Section 4.3. The gene similarity rate and malware classification with threshold β are shown in Table 8.

Table 8 Factors of threshold β

Threshold β	Classification result	Gene similarity rate between families
0.3	0.65	0.100
0.4	0.78	0.080
0.5	0.91	0.040
0.6	0.98	0.020
0.7	0.93	0.010
0.8	0.85	0.008
0.9	0.87	0.005

In Table 8, we can see that if the threshold is too large, the recognition rate of the classification method is low. On the other hand, if the threshold is too small, it will increase the size of the gene database and the similarity rate between genes of different families, which will cause false positives. After attempting various thresholds repeatedly, we obtain a comparison of the classification results and determine the threshold to be 0.6.

4.3 Malware classification

The accuracy of classification is an important index to evaluate the effect of the homology identification method. In this study, the Kaggle and VxHeaven data sets are used to verify the effectiveness of system classification.

4.3.1 Classification of the Kaggle dataset

In our method, the gene is extracted from FDG and converted from the assembler code. Therefore, we use the ".asm" file in the dataset as the experimental object. For each family, we first randomly select 80% of the samples as the training data and other 20% as the testing data. We extract and mine the family gene from each malware family in the training data to establish the gene database, as mentioned in Section 4.2.

Then, we use the testing data to verify the effect of the classifier. For each sample of the testing data, we use the whole key API set of every malware family of the training data to select the key API, and use the GCN to detect the malware gene. We compare the gene of each testing sample with the family gene of each malware family in the gene database. After that, we calculate the gene similarity using the Jaccard coefficient. We repeat the experiment 10 times, and the classification results are shown in Table 9.

Table 9 Classification results of the Kaggle dataset

Predicted	Actual							
	1	2	3	4	6	7	8	9
1	0.97	0.00	0.02	0.00	0.00	0.01	0.00	0.00
2	0.00	0.98	0.01	0.00	0.01	0.00	0.00	0.00
3	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
6	0.00	0.00	0.00	0.02	0.98	0.00	0.00	0.00
7	0.00	0.00	0.02	0.00	0.01	0.97	0.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Predicted: classification result of prediction; Actual: actual family of the data

In Table 9, the results show that the accuracy of our classifier can reach 97%. There are still some misjudgment cases due to the similar behavior of two malicious code populations, which leads to the similarity of some areas in the FDG and thus leads to misjudgment.

4.3.2 Classification of the VxHeaven dataset

We select eight families from the VxHeaven dataset. There are more than 300 samples in each family. The family information is shown in Table 10.

Table 10 Malware family information

Family representation	Malware family	Sample number
A	Trojan-downloader.Win32.small	451
B	Backdoor.Win32.VB	343
C	Worm.Win32.AutoRun	480
D	Backdoor.Win32.Agent	546
E	Backdoor.Win32.Small	789
F	Trojan.Win32.Buzus	697
G	Trojan-GameThief.Win32.Nilage	332
H	Trojan-Clicker.Win32.Vapsup	962

We split each family into 80% training and 20% test subsets. Because each file type of the VxHeaven dataset is ".exe," we must preprocess the dataset file to obtain the assembler code. Then we establish the gene database from the training data and extract the malware gene from the testing data, which is similar to that in Section 4.2. Finally, we compare the gene set of each sample with the gene database to find the malware family. The experiment is repeated 10 times

and the results are shown in Table 11.

In Table 11, the horizontal coordinate represents the actual families of the samples, and the vertical coordinate represents the families given by our method. The results show that the accuracy of our classifier can reach 90%. There are still some mistakes because malware of the same type can have similar behaviors. Therefore, the classifiers tend to confuse their behavior.

Table 11 Classification results of the VxHeaven dataset

Predicted	Actual							
	A	B	C	D	E	F	G	H
A	0.92	0.00	0.02	0.00	0.00	0.04	0.02	0.00
B	0.00	0.90	0.01	0.04	0.03	0.02	0.00	0.00
C	0.00	0.00	0.94	0.04	0.00	0.02	0.00	0.00
D	0.00	0.05	0.00	0.89	0.06	0.00	0.00	0.00
E	0.00	0.02	0.00	0.05	0.90	0.03	0.00	0.00
F	0.02	0.00	0.02	0.00	0.02	0.91	0.03	0.00
G	0.00	0.00	0.00	0.00	0.00	0.03	0.93	0.04
H	0.02	0.00	0.00	0.00	0.00	0.02	0.02	0.94

Predicted: classification result of prediction; Actual: actual family of the data

4.4 Efficiency

In this subsection, we investigate the efficiency of the analysis and training of our method. Malware homological analysis requires high real-time performance, so a good homological analysis method should be very efficient. To assess the efficiencies of the system, we record the classification time required for 1000 Kaggle dataset malware samples. At the beginning of the experiment, the classification method is trained by the dataset of 8000 samples. The results are shown in Table 12, including statistical information concerning classification time for malware samples.

Table 12 Distribution of detection time

Detection time (s)	Sample number
1	348
2	435
3	54
4	104
5	46
6	13

In Table 12, the classification time for most samples is less than 2 s. The longest time is 6.9 s, but only in isolated cases. The average detection time of most samples is less than 3 s, which is acceptable for practical applications. At the same time, we compare the efficiencies of the family gene selected by different frequent graph mining algorithms. We compare our compression-based method with the gSpan algorithm. We use 8000 subgraphs as the dataset. The subgraph scale distribution for the dataset is shown in Table 13 and the compressed graph scale distribution in Table 14.

Table 13 Distribution of the subgraph scale

Graph node number	Graph number
1−10	1035
10−20	1367
20−30	2467
30−40	2201
40−50	891
50−60	39

Table 14 Distribution of the compressed graph scale

Compressed graph node number	Graph number
1−10	2045
10−20	3351
20−30	1567
30−40	843
40−50	107
50−60	87

In Tables 13 and 14, we can see that the compression processes effectively reduce the graph scale. After that, we compare the processing time of different algorithms.

In Fig. 9, the compression-based method saves more time than the gSpan algorithm. With increase in the number of subgraphs, the efficiency improvement of the compression-based method is obviously higher than that of the gSpan algorithm.

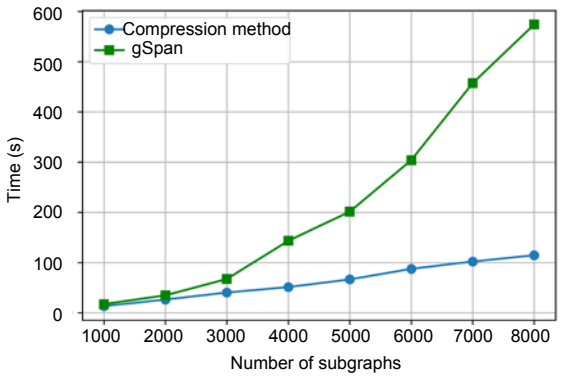

Fig. 9 Frequent graph mining time

4.5 Comparison with other methods

There were 377 international teams competing in the Kaggle contest. The winning team reported that their method produced an accuracy greater than 99% (Wang et al., 2015). The wining team proposed a very complex model that requires more than 500 GB for the training data and other 200 GB for engineered features. Furthermore, it took 72 h to produce the model with best model ensemble. This is unacceptable in homological analysis.

In addition to accuracy, time efficiency of homological analysis is very important. The method proposed by Drew et al. (2016) has good accuracy and time efficiency. The classification time of 1000 samples is about 30 min, and the accuracy is 97%.

Similar to Drew's method, our method is based on the gene perspective and the classification accuracy reaches 97%. Because of gene encoding, the encoded gene adopts the method of set comparison, which has better time efficiency. The classification time of 1000 samples is not more than 30 min. In addition, the spatial efficiency of our method is better than that of Drew's, which requires more than 5 GB memory. The family gene database based on the Kaggle dataset requires 2.5 GB memory.

5 Conclusions

In this study, we present a malware homology identification method based on a gene perspective. By establishing the FDG for different malicious code and taking the common graph structure of the dependency graph as the common gene of a malware family, we judge the homology of malicious code from a gene perspective. With compression and encoding, the efficiency of gene comparison is improved. Finally, we perform extensive experiments on different datasets. The results show that the method is highly efficient in malware classification, with an accuracy of 97%.

In the future, we will include more features to improve gene encoding in malware classification, and use gene technology to establish a system to detect the malicious code of unknown families in practical applications. We will also study the similarity of the malware which is packed by unique pack tools to find the pack tool gene. In this way, the gene-based approach will play an important role in practical applications.

Compliance with ethics guidelines

Bing-lin ZHAO, Zheng SHAN, Fu-dong LIU, Bo ZHAO, Yi-hang CHEN, and Wen-jie SUN declare that they have no conflict of interest.

References

Alam S, Horspool RN, Traore I, 2013. MAIL: Malware Analysis Intermediate Language: a step towards automating and optimizing malware detection. Proc 6th Int Conf on Security of Information and Networks, p.233-240. https://doi.org/10.1145/2523514.2527006

Alam S, Horspool RN, Traore I, 2014. MARD: a framework for metamorphic malware analysis and real-time detection. 28th Int Conf on Advanced Information Networking and Applications, p.212-233. https://doi.org/10.1109/AINA.2014.59

Cesare S, Xiang Y, Zhou WL, 2013. Malwise—an effective and efficient classification system for packed and polymorphic malware. *IEEE Trans Comput*, 62(6):1193-1206. https://doi.org/10.1109/TC.2012.65

Defferrard M, Bresson X, Vandergheynst P, 2016. Convolutional neural networks on graphs with fast localized spectral filtering. Conf and Workshop on Neural Information Processing Systems, p.3837-3845.

Drew J, Moore T, Hahsler M, 2016. Polymorphic malware detection using sequence classification methods. Security and Privacy Workshops, p.81-87. https://doi.org/10.1109/SPW.2016.30

Han J, Zhao RC, Shan Z, et al., 2018. Analyzing and recognizing Android malware via semantic-based malware gene. Int Conf on Cyber-Enabled Distributed Computing and Knowledge Discovery, p.17-20. https://doi.org/10.1109/CyberC.2017.36

Jang JW, Woo J, Yun J, et al., 2014. Mal-netminer: malware classification based on social network analysis of call graph. Proc 23rd Int Conf on World Wide Web, p.731-734. https://doi.org/10.1145/2567948.2579364

Kaggle, 2015. Microsoft Malware Classification Challenge (Big 2015). https://www.kaggle.com/c/malware-classification [Accessed on Nov. 4, 2015].

Kinable J, Kostakis O, 2011. Malware classification based on call graph clustering. *J Comput Virol*, 7(4):233-245. https://doi.org/10.1007/s11416-011-0151-y

Kipf TN, Welling M, 2016. Semi-supervised classification with graph convolutional networks. https://arxiv.org/abs/1609.02907?context=cs

Kirat D, Vigna G, 2015. MalGene: automatic extraction of malware analysis evasion signature. Proc 22nd ACM SIGSAC Conf on Computer and Communications Security, p.769-780. https://doi.org/10.1145/2810103.2813642

Liu L, Wang BS, Yu B, et al., 2017. Automatic malware classification and new malware detection using machine learning. *Front Inform Technol Electron Eng*, 18(9): 1336-1347. https://doi.org/10.1631/FITEE.1601325

Naval S, Laxmi V, Rajarajan M, et al., 2017. Employing program semantics for malware detection. *IEEE Trans Inform Forens Secur*, 10(12):2591-2604. https://doi.org/10.1109/TIFS.2015.2469253

Qiao YC, Yun XC, Zhang YZ, et al., 2016. An automatic malware homology identification method based on calling habits. *Acta Electron Sin*, 44(10):2410-2414. https://doi.org/10.3969/j.issn.0372-2112.2016.10.019

Qihoo 360, 2017. Ransomware Threat Situation Analysis Report. http://zt.360.cn/1101061855.php?dtid=1101062360&did=490927082

Wang XZ, Liu JW, Chen XE, 2015. Microsoft Malware Classification Challenge (Big 2015) first place team: say no to overfitting. https://github.com/xiaozhouwang/kaggle_Microsoft_Malware/blob/master/Saynotooverfitting.pdf [Accessed on Nov. 2, 2015].

Wu J, Dong MX, Ota K, et al., 2018a. Big data analysis-based secure cluster management for optimized control plane in software-defined networks. *IEEE Trans Network Ser Manag*, 15(1):27-38. https://doi.org/10.1109/TNSM.2018.2799000

Wu J, Luo SB, Wang S, et al., 2018b. NLES: a novel lifetime extension scheme for safety-critical cyber-physical systems using SDN and NFV. *IEEE Int Things J*, 6(2):2463-2475. https://doi.org/10.1109/JIOT.2018.2870294

Yu B, Fang Y, Yang Q, et al., 2018. A survey of malware behavior description and analysis. *Front Inform Technol Electron Eng*, 19(5):583-603. https://doi.org/10.1631/FITEE.1601745