Hindawi
Mathematical Problems in Engineering
Volume 2020, Article ID 6804290, 12 pages
https://doi.org/10.1155/2020/6804290

Research Article

A Novel Malware Classification Method Based on Crucial Behavior

Fei Xiao,[1,2] **Yi Sun**,[1,2] **Donggao Du**,[1,2] **Xuelei Li**,[3,4] **and Min Luo**[5]

[1]*Network and Information Center, Institute of Network Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[2]*National Engineering Laboratory for Mobile Network Security (No. [2013] 2685), Beijing University of Posts and Telecommunications, Beijing 100876, China*
[3]*Inspur Electronic Information Industry Co., Ltd, Jinan 250000, China*
[4]*State Key Laboratory of High-end Server and Storage Technology, Jinan 250000, China*
[5]*Ernst and Young, Tokyo, Japan*

Correspondence should be addressed to Yi Sun; sybupt@bupt.edu.cn

Received 24 December 2019; Revised 22 February 2020; Accepted 28 February 2020; Published 21 March 2020

Academic Editor: Ramon Sancibrian

Copyright © 2020 Fei Xiao et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Recently, some graph-based methods have been proposed for malware detection. However, current malware is generally characterized by sophisticated behaviors, which makes graph-based malware detection extremely challenging. To address this issue, we propose a graph repartition algorithm by transforming API call graphs into fragment behaviors based on programs' dynamic execution traces. The proposed algorithm relies on the *N*-order subgraph (*NSG*) for constructing the appropriate fragment behavior. Moreover, we improve the term frequency-inverse document frequency- (TF-IDF-) like measure and information gain (IG) to extract the crucial *N*-order subgraph (*CNSG*). This novel behavioral representation and improved extraction method can accurately represent crucial behaviors of malware. Experiments on 4,400 samples demonstrate that the proposed method achieves a high accuracy of 99.75% in malware detection and promising performance of 95.27% in malware classification.

1. Introduction

Malware refers to any software that aims at damaging or infiltrating computer systems [1]. The fast-growing malware variants pose a serious threat to malware detection. According to Symantec's 2018 Internet Security Threat Report (ISTR), the number of malware variants reached 669,947,865 in 2017, doubling that of 2016 [2]. Moreover, the advent of new technologies has contributed to the increasing complexity of malware. Facing numerous and sophisticated malware variants, malware detection is urgently needed (e.g., see [3–5]).

Among the existing methods, malware detection is mainly divided into static and dynamic analysis methods [6]. Static analysis methods are processes of analyzing instructions and structures to confirm program functions [7]. They do not need to run malware directly. Unfortunately, static analysis methods are sensitive to sophisticated obfuscation instructions and encryption techniques. Aiming at the shortcomings of the static analysis methods, dynamic analysis methods are proposed. The advantage of dynamic methods is that they observe behaviors by running programs in a virtual environment. Dynamic methods commonly address the threat of statically obfuscated malware [8] and encryption techniques. Obfuscated malware samples can change their code syntax while preserving their semantics [9]. Dynamic analysis is an effective way to recognize malware behavior. For example, when we want to analyze the keylogger, dynamic analysis can help us find the keylogger's log file and trace the information.

The program generally relies on Application Programming Interface (API) calls provided by the operating system to accomplish its functions. Hence, a program's execution trace can be obtained by monitoring the stream of API calls [10]. The API call graph is constructed by tracing API calls and their arguments, which is an effective method to indicate

program behavior [11]. Considerable effort has been expended to identify malware by using the API call graph [12, 13]. With the advent of sophisticated malware samples, the API call graph is becoming more and more complex [14, 15]. The major issue facing malware detection is computational complexity [16, 17] of graph matching. Moreover, it is a great challenge to construct a graph that is general enough to classify malware.

Our method is different from previous methods since it solves one major shortcoming. We propose a novel method that divides the API call graph into fragment behaviors. Moreover, we extract crucial behaviors by applying the term frequency-inverse document frequency- (TF-IDF-) like measure [18, 19] and information gain (IG) [20, 21]. Finally, we utilize Random Forest (RF) [22], Support Vector Machine (SVM) [23], Decision Tree (DT) [24, 25], and K-Nearest Neighbor (KNN) [26] for malware classification. We aim to enhance the malware classification performance by constructing an appropriate behavioral representation of the malware family. The main contributions of this method are as follows:

(1) We propose a graph repartition algorithm to extract fragment behaviors from original API call graphs. The extracted fragment behavior is a graph-based API sequence that preserves the dependency of the API call graph.

(2) We extract crucial behaviors by improving the feature extraction measure. The improved extraction measure which combines TF-IDF and IG shows great advantages in malware classification.

(3) The proposed method achieves promising performance in both malware detection and classification. The experimental results demonstrate that the extracted crucial behaviors can accurately describe malware activities.

The rest of this paper is organized as follows: Section 2 reviews the related work. Section 3 introduces some basic notations. Section 4 represents the proposed method which consists of system overview, graph repartition, and feature extraction module. Experiment and evaluation are described in Section 5. The limitations of the proposed method are discussed in Section 6, which is followed by the conclusion in Section 7.

2. Related Work

Programs generally perform various activities by taking advantage of different predefined API calls. API calls provide valuable information to identify potential exceptions and malicious activities. A considerable amount of researchers has been devoted to the research of the API call sequence. Eskandari et al. [27] proposed a dynamic malware detection system that explores system call via API call. In addition, they extracted API calls from the log file and used the n-gram to generate 4-gram API call sequences. Lee et al. [28] utilized the Cuckoo sandbox to execute programs dynamically. They extracted API behavior data and transformed API calls into sequences by using the n-gram method. Moreover, they calculated the frequency of sequences. After that, the cosine similarities of API sequences were calculated among different programs. Finally, the malware samples which are similar to each other were grouped.

Hansen et al. [29] presented a scalable dynamic analysis method by injecting programs into parallel virtual environments. The parallel virtual environment is implemented by developing the setup of the Cuckoo sandbox. They extracted labels and features from samples. The extracted features consist of API calls and their input arguments which include registry and DLLs. After that, they proposed two representation methods for malware detection and classification.

As mentioned above, the common parts of API call sequences can be utilized to identify the similarities of malware samples. The sequence-based approach is relatively simple to describe malware behavior. However, sequence-based methods only preserve temporal information between API calls, which are vulnerable to reorder or irrelevant API calls. Some methods have been proposed to address the drawbacks of sequence-based methods, such as deep learning-based models and more comprehensive feature representation.

Amin et al. [30, 31] explored bidirectional long short-term memory for building an antimalware system to detect static opcodes of malware. In addition, they designed a deep learning model of generative adversarial networks to detect Android malware.

D'Angelo et al. [32] transformed API call sequences which are invoked by apps during their execution to API-images. They autonomously extracted the most representative and discriminating features by applying autoencoders. The deep learning-based model shows great advantages in malware detection.

On the other hand, the API call graph is proposed to capture comprehensive relations (such as argument dependency) between API calls [33]. Park et al. [34] monitored the execution of programs and then constructed weighted directed behavioral graphs that represent kernel objects, object attributes, and dependencies between kernel objects. In addition, they proposed a method to generate a common behavioral graph by clustering individual behavioral graphs.

Elhadi et al. [11] presented a static analysis system; the proposed system read samples and then extracted API calls and their parameters under a secure environment. They classified API call graphs based on sequence dependence, data dependence, declaration dependence, and API dependence. For each kind of dependence, they constructed an API call graph. Finally, they integrated four kinds of API call graphs into an integrating API call graph and calculated the similarity between graphs.

Nikolopoulos and Polenakis [35] proposed a graph-based model based on dynamic taint analysis. The proposed model is constructed by exploring main properties of system-call dependency graphs. They adopted the Euclidean distance-based Δ-similarity metric for malware detection and the SaMe-NP similarity metric for malware classification.

Programs generally accomplish tasks by executing similar behaviors or repeating behavior multiple times. More similar or repeated behaviors occur, and more duplicated nodes or subgraphs appear. The drawback arises with sophisticated behavior which results in high dimensional features and brings more calculations [36]. Furthermore, it is unsatisfied that a behavioral graph is too specific which may ignore the minor changes in malware variants [37]. Likewise, not specific enough of the behavior graph commonly leads to benign samples judged as malware. A large number of work have been concentrated on investigating accurate approximation methods for these problems.

Fredrikson et al. [37] mined significant behaviors from samples based on the data dependence graph. The mined significant behaviors are then utilized to synthesize an optimally discriminative specification based on concept analysis and simulated annealing [38] algorithm. The focus of this proposed method is to reduce the size of the graph [39].

Alam et al. [40] put forward the "Annotated Control Flow Graph" and "Sliding Window of Difference and Control Flow Weight" to reduce the effects of obfuscations. The proposed Annotated Control Flow Graph provides a quick graph matching method by dividing itself into many smaller Annotated Control Flow Graphs. The proposed Sliding Window of Difference and Control Flow Weight captures the semantics of the control flow and helps in malware detection.

Ding et al. [41] constructed an API dependency graph by tracing taint data. After that, they proposed a dependency graph pruning algorithm for pruning a dependency graph. Finally, they constructed a common behavioral graph based on the pruned dependency graph. The proposed common behavior graph prunes similar and repeated behaviors.

We provide a comprehensive summary of malware detection and classification work in Table 1. To simplify the representation of the graph, we propose a novel graph repartition algorithm. The proposed algorithm constructs fragment behaviors that describe crucial activities of the malware. The novel and simplified representation of fragment behavior preserves the dependency of the API call graph and effectively avoids problems in graph matching. This novel behavioral representation is designed to provide a better malware classification performance.

3. Basic Notation

We explain some notations in this section: subgraph, *N*-order subgraph, crucial *N*-order subgraph, and TF-IDF.

API call graph commonly represented by a directed acyclic graph which consists of nodes and edges. If an API call A is associated with API call B, an edge is established from node A to node B. That is, edges represent dependencies among different types of nodes (e.g., network, registry, and file system).

API call graph defines specific behaviors. We annotate root and leaf nodes with labels in each API call graph. After that, we extract the full execution paths from the root node to leaf nodes in an API call graph. These no-branching execution graphs extracted from API call graph are represented as subgraphs in our system.

N-order subgraph is extracted from the subgraph by sliding a window of size *N*.

Definition 1 (*N*-Order Subgraph (*NSG*)). *NSG* is a graph in which the maximum number of nodes does not exceed *N*.

NSGS stands for *NSG* set:

$$NSGS = \left(NSG_1, NSG_2, \ldots, NSG_m\right), \tag{1}$$

where m is the number of *NSG* in *NSGS*.

NSG with the crucial information is chosen as an indicator of malware. We call this crucial *NSG*.

Definition 2 (Crucial *N*-Order Subgraph (*CNSG*)). *CNSG* is a subset of *NSGS*, and it contains the crucial information of *NSGS*. It can be described as follows:

$$CNSG = \arg \max_{NSG_i \in NSGS} \left(Cru\left(NSG_i\right)\right), \tag{2}$$

where $Cru(NSG_i)$ is the crucial coefficient of *N*-order subgraph NSG_i.

TF-IDF is a numerical statistic in information retrieval. It reflects the importance of words in a document. TF refers to the number of times a given word appears in a document. IDF measures the general importance of words [42].

Definition 3. TF-IDF is the product of TF and IDF.

Given a document d_i and document set D which has n documents, $D = (d_1, d_2, \ldots, d_n)$. The word in the dataset is represented as w. TF-IDF is calculated as follows:

$$\begin{aligned} \mathrm{TF-IDF}\left(w, d_i, D\right) &= \mathrm{TF}\left(w, d_i\right) * \mathrm{IDF}\left(w, D\right), \\ \mathrm{TF}\left(w, d_i\right) &= \frac{f\left(w, d_i\right)}{\left|d_i\right|}, \\ \mathrm{IDF}\left(w, D\right) &= \log \frac{|D|}{\left|\left\{d_i \mid w \in d_i\right\}\right|}, \end{aligned} \tag{3}$$

where $\mathrm{TF}(w, d_i)$ represents the frequency of the word w in document d_i, $f(w, d_i)$ is the number of times the given word w appears in a document d_i, and $|d_i|$ represents the dimension of the document d_i. $\mathrm{IDF}(w, D)$ reflects the inverse document frequency of the word w in document set D, and $|\{d_i \mid w \in d_i\}|$ is the number of the documents which contain w.

4. The Proposed Method

4.1. Malware Classification System Overview. Our method consists of three parts: graph repartition, feature extraction, and malware classification. The whole process of the proposed system is outlined in Figure 1. Graph repartition consists of two modules: subgraph construction module and *NSG* construction module. Subgraph construction module extracts subgraphs from API call graphs which are constructed based on the registry, filesystem, process, services,

Table 1: Summary of malware detection and classification work.

Approach	Features	Note
Eskandari et al. [27]	API call sequence	
Lee et al. [28]	API call sequence	Simple, vulnerable to reorder or irrelevant API calls
Hansen et al. [29]	API call sequence; arguments; frequency	
Amin [30, 31]	Opcode	End-to-end learning
D'Angelo et al. [32]	API call sequence-based image	
Park et al. [34]	Behavioral graph	
Elhadi et al. [11]	API call graph	High dimensional features can bring more calculations
Nikolopoulos and Polenakis [35]	System call dependency graph	
Fredrikson et al. [37]	Optimally discriminative specification	
Alam et al. [40]	Control flow graph-based feature	Simplified representation of behavior graphs
Ding et al. [41]	API dependency graph	

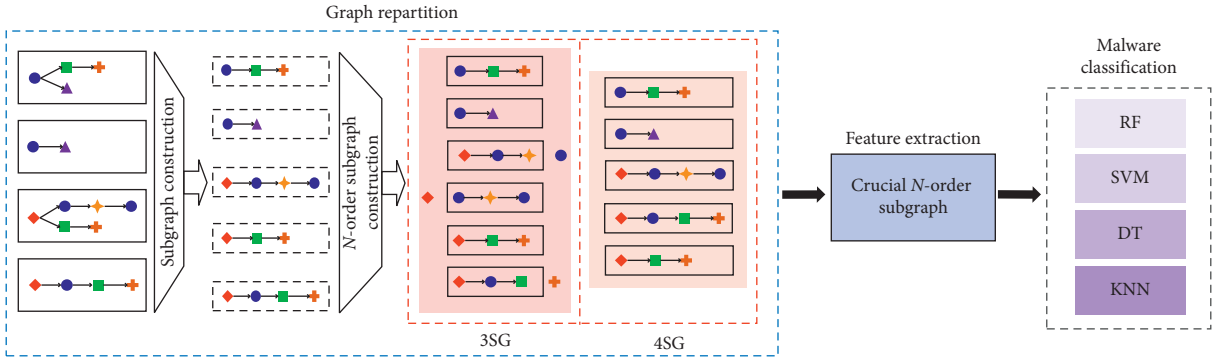

Figure 1: Overview of the proposed method.

network, and synchronization. *NSG* construction module extracts *NSG* from the subgraph construction module. Our goal is to build the appropriate behavioral representation and extract *CNSG* by using the improved TF-IDF-like measure in feature extraction. In the last step, RF, SVM, DT, and KNN are used for malware classification. The following are the steps of our proposed system.

Step 1. Extract subgraphs

We extract subgraphs from API call graphs of malware and benign samples. Icons with different shapes and colors represent various API calls in Figure 1. We can see that four API call graphs are listed in different rectangles. The subgraph construction module extracts five different subgraphs from four API call graphs.

Step 2. Build fragment behavior of *NSG*

NSG is obtained through an API call repartition algorithm based on the sliding window. We illustrate *3SG* and *4SG* in Figure 1. Icons that are not in the shadow refer to the parts that need to be discarded. *NSG* preserves more complex semantic information than API sequences, which contains the dependencies of API call graphs.

Step 3. Extract crucial behavior of *CNSG*

We adopt the TF-IDF-like measure and IG to calculate the crucial coefficient of *NSG*. The *NSG* with the higher crucial coefficient is selected as the significant behavior (e.g., *CNSG*) in our method.

Step 4. Malware classification

For each program analyzed in Cuckoo sandbox, we use some classifiers (e.g., RF, SVM, DT, and KNN) to identify whether the program is benign or malware. We obtain the appropriate *CNSG* in this process by comparing the performance of the experiments.

4.2. Malware Classification System Overview. In this section, we propose a graph repartition algorithm that reconstructs the API call graph to *NSG*. The purpose of the proposed algorithm is to build the appropriate fragment behavior of malware families by pruning similar behaviors.

Figure 2 shows the trace extracted from the log file generated through the Cuckoo sandbox. This is part of the input that the API call graph is built from. In line 1, one can see that the malware creates and opens a registry. After that, it repeatedly retrieves and sets the data. On lines 7 and 8, the malware creates a file and changes its information. In line 9, the program retrieves the information of the file. It closes the file on line 10. On lines 11 to 13, the program creates, retrieves, and closes another file.

In Figure 3, G_1, G_2, and G_3 are three API call graphs from line 1 to line 6, line 7 to line 10, and line 11 to line 13 in Figure 2, respectively. As illustrated in Figure 3, API call is applied to construct the node of a graph and the arguments are utilized to connect two API calls based on dependencies. For example, API call of line 1 in Figure 2 is labeled as RegCreateKey (Handle => 0x0000044c, Registry => 0x80000001, SubKey => ...proxyTool). The value

(i) RegCreateKey (Out Handle => 0x0000044c, Registry => 0x80000001, SubKey => ... proxyTool)
(ii) RegQueryValue (Out Handle => 0x0000044c, ValueName => team tick)
(iii) RegSetValue (Out Handle => 0x0000044c, ValueName => team tick)
(iv) RegQueryValue (Out Handle => 0x0000044c, ValueName => team tick)
(v) RegSetValue (Out Handle => 0x0000044c, ValueName => team tick)
(vi) RegQueryValue (Out Handle => 0x0000044c, ValueName => team tick)
.....
(vii) NtCreateFile (Out FileHandle => 0x000000a0, ..., Filename => ...index.dat)
(viii) NtSetInformationFile (In FileHandle => 0x000000a0)
(ix) NtQueryInformationFile (In FileHandle => 0x000000a0)
(x) NtClose (In FileHandle => 0x000000a0)
(xi) NtCreateFile (Out FileHandle => 0x0000010c, ..., Filename => ...phcl2vj0e73a.bmp)
(xii) NtQueryInformationFile (In FileHandle => 0x0000010c)
(xiii) NtClose (In FileHandle => 0x0000010c)

Figure 2: Execution trace of a malware sample.

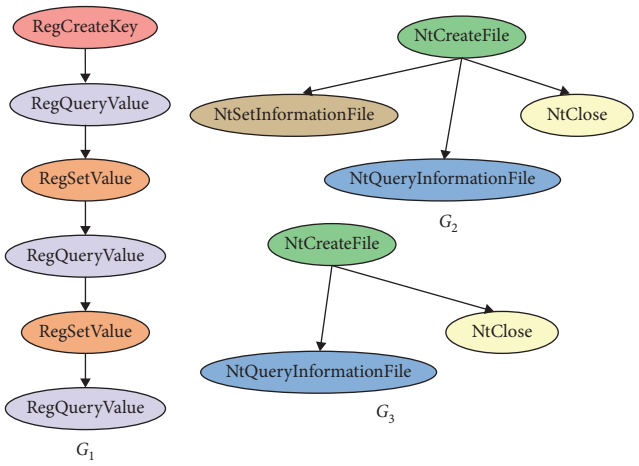

Figure 3: Some API call graphs.

0x0000044c of Handle is used to connect the RegQueryValue on line 2. The details of API call graph construction are described in our previous work [43]. It is necessary to extract crucial behaviors from the API call graph for malware classification.

For each API call graph, we first identify the root and leaf nodes. The root node is a node with no input information, and the leaf node is the node whose output is null in our system. Also, we need to extract subgraphs from the established API call graph. The extracted subgraphs are simple no-branching graphs that start at the root node and end with the leaf node. We obtained all subgraphs as follows.

When there is only one branch in the API call graph, the extracted subgraph is the same as the API call graph. The subgraph of G_1 is

{*RegCreateKey*, *RegQueryValue*, *RegSetValue*, *RegQueryValue*, *RegSetValue*, *RegQueryValue*}

There are multiple branches in G_2 and G_3. In this case, we need to extract different no-branching subgraphs based on root and leaf nodes.

Subgraphs of G_2 are explained as follows:

{*NtCreateFile*, *NtSetInformationFile*}
{*NtCreateFile*, *NtQueryInformationFile*}
{*NtCreateFile*, *NtClose*}

Subgraphs of G_3 are explained as follows:

{*NtCreateFile*, *NtQueryInformationFile*}
{*NtCreateFile*, *NtClose*}

We divide the subgraph into fragment behavior through a sliding window. The behavior in a sliding window is a fragment behavior. The fragment behaviors are a set of behaviors that can accomplish a part or a certain function. The extracted fragment behavior is represented by *NSG* in our method. The size of the sliding window determines the maximum number (N) of *NSG*.

We show the process of some *NSG*s extracted from the subgraph of G_1 in Figure 4. When $N = 3$ (*3SG*), SG1 in the first window is the first *3SG* extracted from the subgraph. The sliding window of size 3 slides from top to bottom at intervals of 1. Different colors of the window reflect the moving trail of the sliding window. We can see that the fragment behavior in the fourth window is the same as the fragment behavior in the second window. Hence, three unique *3SG*s of SG1, SG2, and SG3 are extracted from the subgraph:

SG1: {*RegCreateKey*, *RegQueryValue*, *RegSetValue*}
SG2: {*RegQueryValue*, *RegSetValue*, *RegQueryValue*}
SG3: {*RegSetValue*, *RegQueryValue*, *RegSetValue*}

For G_2 and G_3, we notice that the number of nodes in each subgraph is smaller than 3 when we want to build *3SG*. In this condition, the subgraph is regarded as a *3SG*, which does not need to be divided. When all *NSG*s are extracted from all subgraphs of a program, we obtain *NSGS*. This set is used to represent the program behavior. The combination of *NSG*s contains the complete semantic information of a program's API call graph. Our goal is to describe malware with an appropriate fragment behavioral representation by searching for N.

Algorithm 1 describes our proposed API Call Repartition Algorithm. The input of this algorithm is the API call graph $(G_1, G_2, \ldots, G_r)$. For an API call graph G_i, we first search for the root and leaf nodes. Lines 3 and 4 describe the root node and leaf nodes in an API call graph. We extract subgraphs from an API call graph in line 5. The extracted

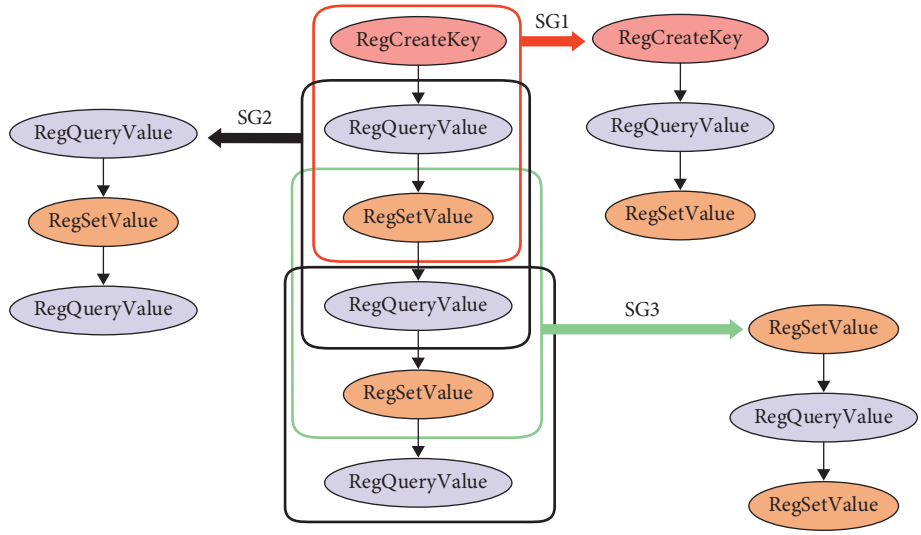

Figure 4: Construction of *3SG*.

Input: API call graph $(G_1, G_2, \ldots, G_r)$
Output: $NSG_1, NSG_2, \ldots, NSG_m$ // $m \geq r$
(1) Begin
(2) **For** $i = 1$ to r **do** // i represents $i-th$ API call graph in a sample
(3) Find the root node V_0 in G_i
(4) Find leaf nodes $(V_k, V_{k+1}, \ldots, V_{k+r})$ in G_i
(5) Extract subgraphs from V_0 to leaf nodes
(6) **End**
(7) Obtained all extracted subgraphs $(G_1', G_2', \ldots, G_n')$
(8) **For** $i = 1$ to n **do** // i represents $i-th$ subgraph and $m \geq n$
(9) **If** the order h in G_i' is smaller than N: $h \leq N$
(10) $NSG_i = G_i'$
(11) **Else**
(12) Divide G_i' into NSG
(13) **End**
(14) **End**
(15) Output $NSG_1, NSG_2, \ldots, NSG_m$
(16) End

Algorithm 1: API call repartition algorithm.

subgraphs start from the root node and end with leaf nodes. It is worth mentioning that the simple paths from the root node to a certain leaf node in an API call graph may occur more than once. For each subgraph in line 8, if the order of a subgraph is smaller than N, *NSG* is the same as the subgraph. Otherwise, we should extract the appropriate *NSG* based on the sliding window. The output of this algorithm is *NSG*. In this algorithm, we transform the original API call graph into fragment behaviors *NSG*.

The subgraph is a no-branching fragment behavior extracted from API call graphs. Hence, the number of subgraph n is no less than the number of API call graph r ($n \geq r$). In the same way, $m \geq n$. An important problem of this issue is the value of N. Our goal is to better describe malware with an appropriate N as small as possible.

API call repartition algorithm removes two types of similar behaviors: internal similarity and external similarity. Internal similarity refers to the similarity of *NSG* in a subgraph. External similarity represents the similarity of *NSG* among different subgraphs. Internal similarity is generally caused by repeatedly executing API calls. For example, if a program repeatedly invokes RegQueryValue and RegSetValue in Figure 4, then two repeated *3SG*s of {*RegQueryValue*, *RegSetValue*, *RegQueryValue*} are generated. In this condition, we select {*RegQueryValue*, *RegSetValue*, *RegQueryValue*} only once. External similarity is commonly caused by API calls that perform the same type of behaviors among different subgraphs. For example, NtCreateFile outputs two different values FileHandle 0x000000a0 and FileHandle 0x0000010c on lines 7 and 11, respectively. As a result, *3SG*s of {*NtCreateFile*, *NtQueryInformationFile*} and {*NtCreateFile*, *NtClose*} are the same type of behaviors in different subgraphs. Therefore, we select {*NtCreateFile*, *NtQueryInformationFile*} and {*NtCreateFile*, *NtClose*} only once in the program. Removing similar behaviors can help to simplify the representation of the graph.

4.3. Feature Extraction. The characteristic of the program is represented as fragment behavior *NSGS* by applying the API call repartition algorithm which eliminates similarity behaviors. To remove unimportant ones, we need to calculate the crucial coefficient of *NSG* in *NSGS*. We propose a method that exploits the idea of TF-IDF and IG to evaluate the importance of an *NSG*.

We have four malware families and different types of benign samples in our proposed system. Different types of benign samples are defined as one family of benign. Hence, we have five categories; the category set is represented as C, where $C = (C_0, C_1, C_2, C_3, C_4)$. Each family has k samples (in our proposed system, $k = 880$).

TF-IDF's main idea is that a fragment behavior *NSG* is appropriate for selecting as a crucial behavior when it appears with a high frequency (TF) in a category and appears with a low frequency in other categories. For IDF, *NSG* is appropriate for selecting as a *CNSG* when a fragment behavior NSG_i appears in a small number of categories.

We consider that a fragment behavior *NSG* appears p times in a category C_j ($C_j \in C$). In addition, it appears q times in other categories except for C_j. Hence, fragment behavior appears $(p + q)$ times altogether. *NSG* is a crucial behavior of C_j when p is large enough, which means that the value of $Cru(NSG_i)$ is very high. However, the value of $\mathrm{IDF}(NSG_i)$ is relatively small because of the large $(p + q)$.

We present the improved TF-IDF-like measure by applying IG which is described in our previous work [20]. IG is defined as how much information the feature brings to the system. The more the information this feature brings to the system, the more important the feature is. The fragment behavior *NSG* appears p times in the category C_j and appears q times in other categories except for C_j. When p is large enough, the value of $\mathrm{IG}(NSG_i)$ is sufficient to select *NSG* as a crucial behavior.

Based on the TF-IDF and IG, we derive a symbolic expression for calculating the coefficient of *NSG* as follows:

$$Cru(NSG_i) = \alpha \mathrm{TF-IDF}(NSG_i) + (1-\alpha)\mathrm{IG}(NSG_i),$$

$$\mathrm{TF-IDF}(NSG_i) = \frac{f(NSG_i, C_j)}{|C_j|} * \log \frac{5 * k}{|\{C \mid NSG_i \in C\}|}, \tag{4}$$

where $\mathrm{TF-IDF}(NSG_i)$ represents the value of the TF-IDF-like measure and $\mathrm{IG}(NSG_i)$ stands for the value calculated by IG. The improved TF-IDF-like method determines the effects of different factors of TF-IDF-like measure and IG by finding appropriate α ($0 < \alpha < 1$). $f(NSG_i, C_j)$ is the number of times NSG_i appears in family C_j, $|C_j|$ is the dimension of C_j, and $\left|\{C_j \mid NSG_i \in C_j\}\right|$ is the number of samples which contain NSG_i.

5. Results

This section describes the dataset and the evaluation method in Section 5.1. Section 5.2 shows the experiment and evaluation results.

5.1. Dataset and Evaluation Method. To prove that our method is effective in detecting malware, a set of malware classification experiments are presented in this section. To ensure the fairness and effectiveness of the experiment, we selected the same amount of families which consist of Delf, Small, Zlob, and OBfuscated. To prevent confusion with obfuscated malware, we use the OBfuscated that begins with two uppercase letters to represent the Trojan.Win32.Obfuscated family. In addition, we download 880 benign samples from different websites. More precisely, benign samples consist of Desk Widget, Facebook Messenger, Google Earth, Matlab, Minclock, and Quicktime player.

Ubuntu is selected as the operating system to run a standard Cuckoo sandbox. First, we process malware samples in bulk by developing the Cuckoo sandbox. Each sample was executed several times. After a comprehensive analysis, the samples that performed malicious behaviors were selected for experimental analysis. As we all know, file-less malware can delete all the files it saves on the infected system disk, injects code into running processes, and uses PowerShell, Windows Management Instrumentation, and other technologies to make detection and analysis difficult. This antianalysis method can bypass hooks deployed in automated analysis sandboxes (such as Cuckoo sandbox). This article does not focus on file-less malware and other escape circumstance. Second, to ensure the fairness and effectiveness of the experimental results, we select 880 samples for each malware family and benign for experiments. Finally, we perform 10-fold cross-validation. In 10-fold cross-validation, we divide all dataset samples into ten parts. To guarantee the proportion of each family, we choose nine parts for training and the last part for testing each time. The experiments are repeated ten times and the accuracy is the average of the experimental results.

In our proposed malware classification method, TP, FN, FP, TN, TPR, and FPR in the formulas are defined in Table 2.

This definition uses Delf as an example. Delf is a malware family in our work.

TP represents the number of samples in which the sample belongs to Delf and is correctly classified as Delf.

FN is the number of samples in which the sample belongs to Delf but not classified as Delf.

FP indicates the number of samples in which the sample not belongs to Delf but classified as Delf.

TN indicates the number of samples in which the sample not belongs to Delf and is not classified as Delf.

The common performance of accuracy is defined as follows:

$$\mathrm{accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FN} + \mathrm{FP} + \mathrm{TN}}. \tag{5}$$

5.2. Experiment and Evaluation Results. RF, SVM, DT, and KNN are employed to evaluate the detection effectiveness of our method and to explore the impact of α in malware classification. We studied the effect of the value of α on different classifiers. We set the size of α from 0.1 to 0.9 to observe the effect of α on different classifiers. The effect of α on the accuracy of different *CNSG*s and classifiers is shown in Figure 5.

Table 2: Descriptions of the metrics.

Abbr	Symbol	Definition
TP	True positive	#The number of malware samples in family f is correctly classified as family f
FN	False negative	#The number of malware samples in family f is incorrectly not classified as family f
FP	False positive	#The number of malware samples not in family f is incorrectly classified as family f
TN	True negative	#The number of malware samples not in family f is correctly not classified as family f
TPR	True positive rate	#TP/(TP + FN)
FPR	False positive rate	#FP/(FP + TN)

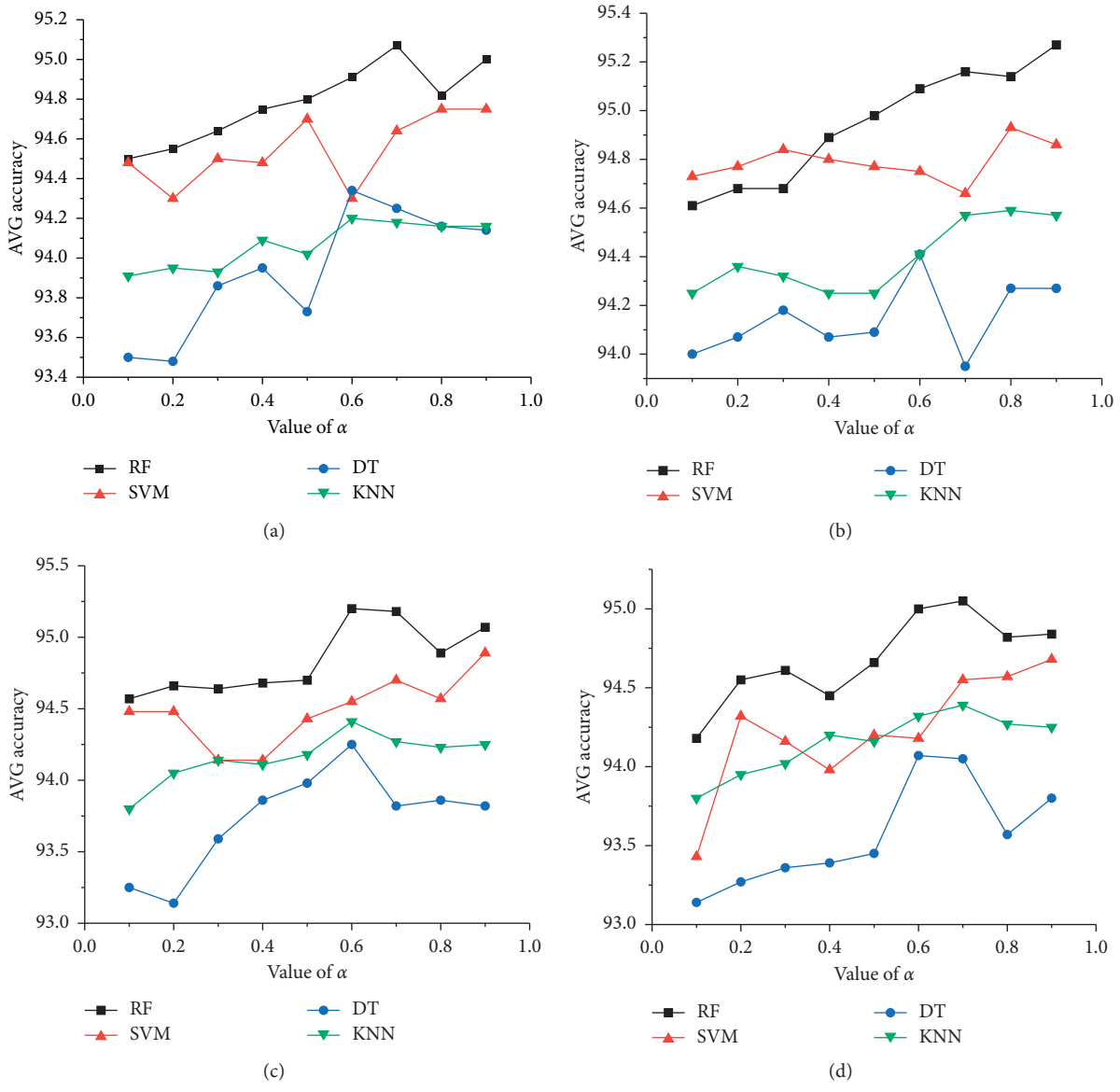

Figure 5: *CNSG* classification performance with α. (a) The relationship between average accuracy of different classifiers (e.g., RF, SVM, DT, and KNN) and the value of α in C3SG. (b) The relationship between average accuracy of different classifiers and the value of α in C4SG. (c) The relationship between average accuracy of different classifiers and the value of α in C4SG. (d) The relationship between average accuracy of different classifiers and the value of α in C6SG.

The horizontal axis of Figure 5 indicates the value of α. The vertical axis of Figure 5 is the average accuracy we obtained from the 10-fold cross-validation.

We can see from Figure 5 that RF has good performance in malware classification based on behavioral fragment *CNSG*. In Figure 5(a), the average accuracy of RF is higher than of other classifiers for different values of α. The average accuracy of RF increases first and then decreases with the increase of α. The average accuracy of RF reaches the optimal value when α is equal to 0.7. In Figure 5(b), when the value of

α is between 0.1 and 0.3, the average accuracy of SVM is optimal. When α is greater than 0.3, the average accuracy of RF is the best and is slowly increasing. The average accuracy of RF reaches the optimal value when α is equal to 0.9. In Figures 5(c) and 5(d), the average accuracy of RF is better than the other three classifiers, and the highest average accuracy is achieved when α is equal to 0.6 and 0.7, respectively.

It can be seen from Figure 5 that with the change of α, the average accuracy of *CNSG* classified by different classifiers has obvious changes. In other words, exploring changes in α has a positive impact on malware classification. α is an indispensable factor in malware classification. We can conclude that the IG can well compensate for the shortcomings of the TF-IDF-like measure when the optimal value of α is obtained.

To prove the validity of our improved TF-IDF-like measure, we compare the TF-IDF-like measure with our proposed method. Table 3 describes the average accuracy of the TF-IDF-like measure and the improved TF-IDF-like measure. We can see from Table 3 that with different classifiers and *CNSG*s, the improved TF-IDF-like measure is better than the TF-IDF-like measure, in most cases.

The experimental results also demonstrate that IG can make up for the deficiency of the TF-IDF in malware classification. When α is 0.9, C4SG has the highest classification accuracy (when the classifier is RF), which is as high as 95.27%. Based on the experimental results, we select C4SG as the final fragment behavior.

For malware detection, we select the optimal value of α obtained in malware classification. We draw a Receiver Operating Characteristic (ROC) curve in Figure 6. The horizontal axis of Figure 6 represents FPR, and the vertical axis of Figure 6 is TPR. The ROC curve reflects the correlation between FPR and TPR. It can be calculated in Figure 6 that the accuracy is as high as 99.7% with the FPR of 1.2%. The experimental results show that C4SG is promising in malware detection.

For malware classification, an example of the ROC curve is depicted in Figure 7. It illustrates the classification performance of C4SG detected by RF. Four pictures with the detection performance of Delf, OBfuscated, Small, and Zlob are presented. Figure 7(a) describes the detection performance of Delf. In Figure 7(a), we compare the ROC curve of API sequence (4 gram), C4SG, and subgraph. We can see from Figure 7(a) that the performance of C4SG is better than the subgraph and API sequence and the performance of the subgraph is better than the API sequence. Figure 7(b) describes the detection performance of OBfuscated. We can see from Figure 7(b) that both C4SG and subgraph obtained better detection performance than API sequence and C4SG is better than the subgraph. Figures 7(c) and 7(d) represent the detection performance of Small and Zlob, separately. In Figures 7(c) and 7(d), C4SG has better detection performance than the subgraph and API sequence, and the detection performance of the subgraph is better than the API sequence.

Table 3: Classification accuracy.

Type	Method	RF	SVM	DT	KNN
C3SG	TF-IDF	0.9450	0.9350	0.9448	0.9391
	Improved TF-IDF	0.9507	0.9475	0.9434	0.9420
C4SG	TF-IDF	0.9498	0.9448	0.9350	0.9391
	Improved TF-IDF	0.9527	0.9493	0.9441	0.9459
C5SG	TF-IDF	0.9482	0.9489	0.9382	0.9425
	Improved TF-IDF	0.9520	0.9489	0.9425	0.9441
C6SG	TF-IDF	0.9470	0.9470	0.9375	0.9425
	Improved TF-IDF	0.9505	0.9468	0.9407	0.9439

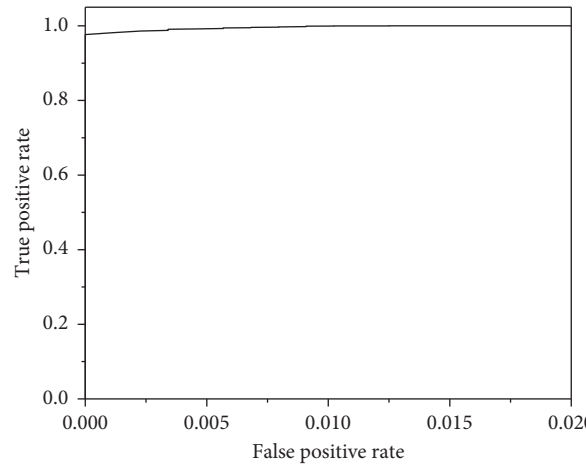

Figure 6: Detection performance with RF.

Subgraph and C4SG contain many API calls and their dependencies. Hence, the semantic in subgraph and C4SG is more abundant than in the API sequence. C4SG achieves a better detection performance than the subgraph. This effectively proves that the C4SG we built is suitable for malware classification.

For malware detection and classification, we make a comparison with some related models, i.e., Fredrikson et al. [37], Alam et al. [40], and Ding et al. [41] in Table 4. Our malware detection result shows good advantages in related studies. For malware classification, authors of [41] have surpassed our results; we take note that Delf, Small, and Zlob in our experiment have some of the same malicious behavior, which may be an important cause of the reduction in classification accuracy.

6. Discussion

We summarize the limitations of the system in this section. In addition, possible solutions are counseled on these limitations.

The main premise of our proposed malware classification method is that we observe malicious activities by executing Cuckoo sandbox. Sandbox is widely used for detecting malware in dynamic analysis. Nevertheless, certain malware samples can evade detection by analyzing the virtual environment to avoid executing malicious operations. In addition, malware writers can also use some

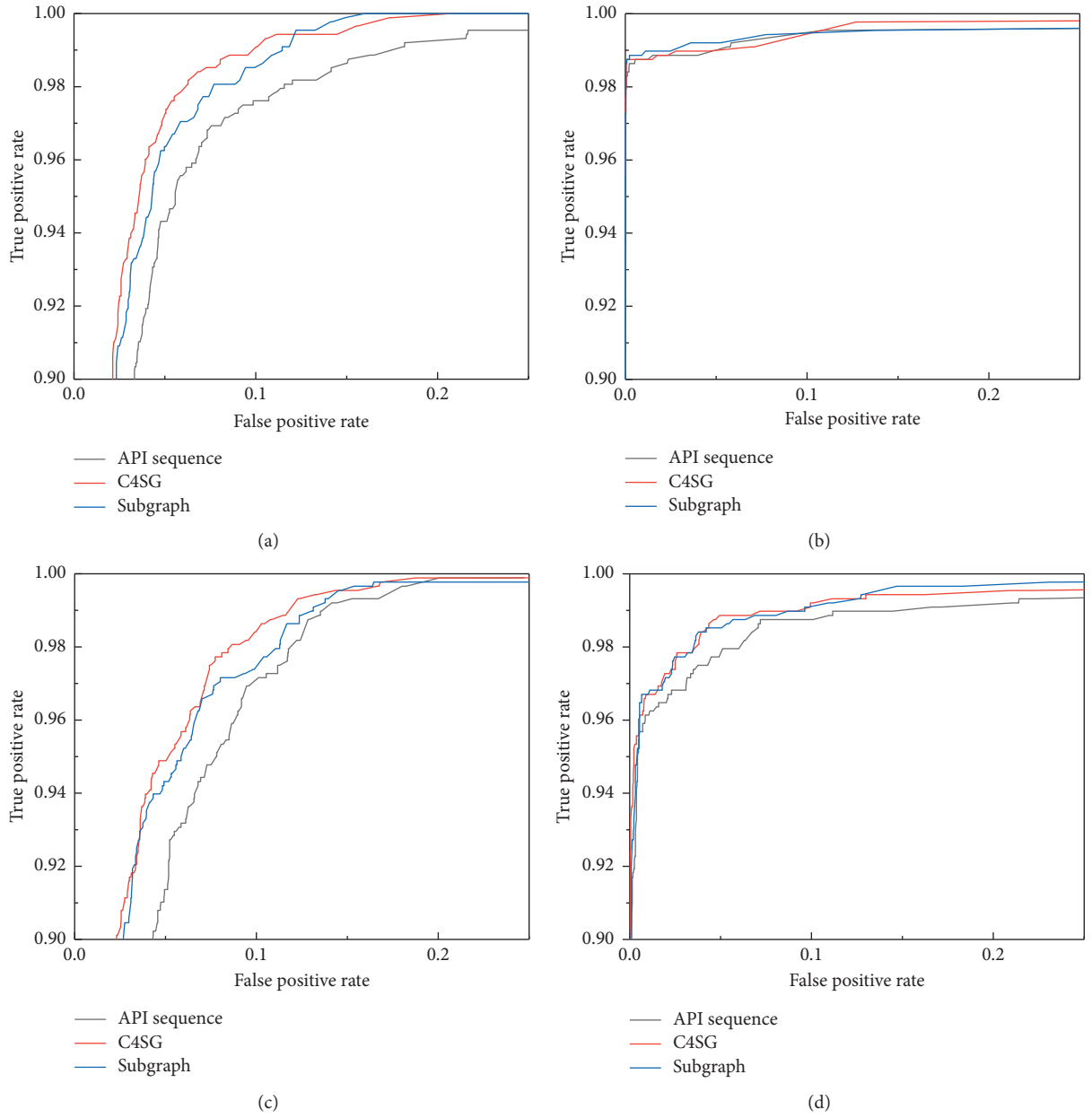

Figure 7: Classification performance with RF. (a) ROC curve of Delf with (i) API sequence, (ii) C4SG, and (iii) subgraph detected by RF. (b) ROC curve of OBfuscated with (i) API sequence, (ii) C4SG, and (iii) subgraph detected by RF. (c) ROC curve of Small with (i) API sequence, (ii) C4SG, and (iii) subgraph detected by RF. (d) ROC curve of Zlob with (i) API sequence, (ii) C4SG, and (iii) subgraph detected by RF. The overall average accuracy of C4SG is better than in the subgraph and API sequence.

Table 4: Summary of related work.

Approach	Analysis type	Accuracy (detection/classification)	Precision	F1 score
Fredrikson et al. [37]	Dynamic	86.56% (D)	NA	NA
Alam et al. [40]	Static	94%–99.6% (D)	NA	NA
Ding et al. (SDG-A) [41]	Dynamic	96.2% (C)	NA	NA
Ding et al. (CDG) [41]	Dynamic	96.4% (C)	NA	NA
Our method	Dynamic	95.27 (C)	95.3%	95.3%
Our method	Dynamic	99.7% (D)	99.7%	99.7%

methods (e.g., delays) to restraint malicious operations during analysis. Executing malware in multiple analysis environments is an effective way to detect evasive samples.

The dataset for malware analysis is relatively small. Larger numbers of malware samples may have better results. Therefore, more samples are needed to implement a large

multiclassification. This is also the work we want to do in the future.

The proposed method is very promising for family classification, but there are miss predictions. In our experiments, some Delf samples are detected as Small and Zlob. The main reason for this misclassification is that they have some of the same malicious behavior. Delf generally downloads and runs files on designated IP and port, causing the malware to run automatically on remote hosts. Small usually infects a computer and connects to remote servers to download malware. Zlob is a Trojan that remotes access to infected computers unauthorized. That is to say, Delf, Small, and Zlob perform remote connection operations and have similar behaviors with each other. In addition, the graph-based sequence may have certain limitations. Therefore, in future work, we intend to explore the similarity of *CNSG* in the form of the graph.

To overcome the shortcomings of traditional detection models, we also need to explore some state-of-the-art modes, i.e., Amin et al. [30], Amin et al. [31], and D'Angelo et al. [32]. We will explore deep learning-based methods to improve the detection rate of malware.

The dataset for malware analysis is relatively small. Larger numbers of malware samples may have better results. Therefore, more samples are needed to implement a large multiclassification. This is also the work we want to do in the future.

7. Conclusions

In this paper, we propose a dynamic malware analysis method that relies on novel feature representation and extraction for malware classification. The proposed feature representation measure transforms malware behavior into fragment behavior. Moreover, the improved feature extraction measure is utilized to extract crucial behaviors of malware families. The experimental results show that the proposed C4SG achieves a promising performance of 95.27% detected by RF in malware classification.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was funded by the National Natural Science Foundation of China under Grant no. 61601041 and the Fundamental Research Funds for the Central Universities under Grant no. 2019PTB-003.

References

[1] J. Aycock, *Computer Viruses and Malware*, vol. 22, Springer Science & Business Media, Boston, MA, USA, 2006.

[2] http://images.mktgassets.symantec.com/Web/Symantec/%7B3a70beb8-c55d-4516-98ed-1d0818a42661%7D_ISTR23_Main-FINAL-APR10.pdf?aid=elq_.

[3] M. Piskozub, R. Spolaor, and I. Martinovic, "MalAlert," *ACM SIGMETRICS Performance Evaluation Review*, vol. 46, no. 3, pp. 151–154, 2019.

[4] J. R. C. Piqueira and C. M. Batistela, "Considering quarantine in the SIRA malware propagation model," *Mathematical Problems in Engineering*, vol. 2019, Article ID 6467104, 8 pages, 2019.

[5] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.

[6] M. Yousefi-Azar, L. G. C. Hamey, V. Varadharajan, and S. Chen, "Malytics: a malware detection scheme," *IEEE Access*, vol. 6, pp. 49418–49431, 2018.

[7] M. Chikapa and A. P. Namanya, "Towards a fast off-line static malware analysis framework," in *Proceedings of the 6th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pp. 182–187, IEEE, Barcelona, Spain, August 2018.

[8] T. Wüchner, A. Cisłak, M. Ochoa, and M. Ochoa, "Leveraging compression-based graph mining for behavior-based malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 1, pp. 99–112, 2017.

[9] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 1970–1977, ACM, Sierre, Switzerland, March 2010.

[10] R. Agrawal, J. W. Stokes, M. Marinescu, and K. Selvaraj, "Neural sequential malware detection with parameters," in *Proceedings of the International Conference On Acoustics, Speech And Signal Processing (ICASSP)*, pp. 2656–2660, IEEE, Calgary, AB, Canada, April 2018.

[11] A. A. E. Elhadi, M. A. Maarof, B. I. A. Barry, and H. Hamza, "Enhancing the detection of metamorphic malware using call graphs," *Computers & Security*, vol. 46, pp. 62–78, 2014.

[12] M. Fan, J. Liu, W. Wang et al., "DAPASA: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.

[13] M. Fan, J. Liu, X. Luo et al., "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[14] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete problems," in *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pp. 47–63, Seattle, WA, USA, April 1974.

[15] L. Livi and A. Rizzi, "The graph matching problem," *Pattern Analysis and Applications*, vol. 16, no. 3, pp. 253–283, 2013.

[16] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 25–36, 2009.

[17] X. Hu, T. C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications*

Security, pp. 611–620, ACM, Chicago, IL, USA, November 2009.

[18] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting TF-IDF term weights as making relevance decisions," *ACM Transactions on Information Systems (TOIS)*, vol. 26, no. 3, pp. 1–37, 2008.

[19] J. Ramos, "Using TF-IDF to determine word relevance in document queries," in *Proceedings of the First Instructional Conference on Machine Learning*, vol. 242, pp. 133–142, Piscataway, NJ, USA, December 2003.

[20] Z. Lin, F. Xiao, Y. Sun et al., "A secure encryption-based malware detection system," *KSII Transactions on Internet & Information Systems*, vol. 12, no. 4, pp. 1799–1818, 2018.

[21] W. Lee and D. Xiang, "Information-theoretic measures for anomaly detection," in *Proceedings of the Symposium on Security and Privacy*, pp. 130–143, IEEE, Oakland, CA, USA, May 2001.

[22] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: a classification and regression tool for compound classification and QSAR modeling," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 6, pp. 1947–1958, 2003.

[23] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and Their Applications*, vol. 13, no. 4, pp. 18–28, 1998.

[24] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[25] C. Kingsford and S. L. Salzberg, "What are decision trees?" *Nature Biotechnology*, vol. 26, no. 9, pp. 1011–1013, 2008.

[26] M. L. Zhang and Z. H. Zhou, "A k-nearest neighbor based algorithm for multi-label classification," in *Proceedings of the International Conference on Granular Computing GrC*, vol. 5, pp. 718–721, Beijing, China, July 2005.

[27] M. Eskandari, Z. Khorshidpur, and S. Hashemi, "To incorporate sequential dynamic features in malware detection engines," in *Proceedings of the European Intelligence And Security Informatics Conference*, pp. 46–52, IEEE, Odense, Denmark, August 2012.

[28] T. Lee, B. Choi, Y. Shin, and J. Kwak, "Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient," *The Journal of Supercomputing*, vol. 74, no. 8, pp. 3489–3503, 2018.

[29] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen, "An Approach for Detection and Family Classification of Malware Based on Behavioral Analysis," in *Proceedings of the International Conference On Computing, Networking And Communications (ICNC)*, pp. 1–5, Kauai, HI, USA, Febraury 2016.

[30] M. Amin, T. A. Tanveer, M. Tehseen, M. Khan, F. A. Khan, and S. Anwar, "Static malware detection and attribution in android byte-code through an end-to-end deep system," *Future Generation Computer Systems*, vol. 102, pp. 112–126, 2020.

[31] M. Amin, B. Shah, A. Sharif et al., "Android malware detection through generative adversarial networks," *Transactions on Emerging Telecommunications Technologies*, no. e3675, 2019.

[32] G. D'Angelo, M. Ficco, and F. Palmieri, "Malware detection in mobile environments based on Autoencoders and API-images," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 26–33, 2020.

[33] C. Kolbitsch, P. M. Comparetti, C. Kruegel et al., "Effective and efficient malware detection at the end host," in *Proceedings of the USENIX security symposium*, vol. 4, no. 1, pp. 351–366, Montreal, Canada, August 2009.

[34] Y. Park, D. S. Reeves, and M. Stamp, "Deriving common malware behavior through graph clustering," *Computers & Security*, vol. 39, pp. 419–430, 2013.

[35] S. D. Nikolopoulos and I. Polenakis, "A graph-based model for malware detection and classification using system-call groups," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 29–46, 2017.

[36] A. Amin, B. Shah, A. Abbas et al., "Features Weight estimation using a genetic algorithm for customer churn prediction in the telecom sector," in *Proceedings of the World conference on information systems and technologies*, vol. 931, Springer, Galicia, Spain, April 2019, Advances in Intelligent Systems and Computing.

[37] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 45–60, IEEE, Oakland, CA, USA, May 2010.

[38] P. Brémaud, *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*, vol. 31, Springer Science & Business Media, Berlin, Germany, 2013.

[39] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han, "Mining graph patterns efficiently via randomized summaries," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 742–753, 2009.

[40] S. Alam, R. N. Horspool, I. Traore, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," *Computers & Security*, vol. 48, pp. 212–233, 2015.

[41] Y. Ding, X. Xia, S. Chen, and Y. Li, "A malware detection method based on family behavior graph," *Computers & Security*, vol. 73, pp. 73–86, 2018.

[42] U. Erra, S. Senatore, F. Minnella, and G. Caggianese, "Approximate TF-IDF based on topic extraction from massive message stream using the GPU," *Information Sciences*, vol. 292, pp. 143–161, 2015.

[43] F. Xiao, Z. Lin, Y. Sun, and Y. Ma, "Malware detection based on deep learning of behavior graphs," *Mathematical Problems in Engineering*, vol. 2019, Article ID 8195395, 10 pages, 2019.