Research Article

A Mobile Malware Detection Method Based on Malicious Subgraphs Mining

Yao Du,¹ Mengtian Cui^{ID},¹ and Xiaochun Cheng^{ID}²

¹Key Laboratory of Computer System, State Ethnic Affairs Commission, Southwest Minzu University, Chengdu 610041, China

²Department of Computer Science, Middlesex University, London NW44BE, UK

Correspondence should be addressed to Mengtian Cui; hangkongzy@163.com

Received 6 February 2021; Revised 10 March 2021; Accepted 7 April 2021; Published 19 April 2021

Academic Editor: Zhe-Li Liu

Copyright © 2021 Yao Du 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.

As mobile phone is widely used in social network communication, it attracts numerous malicious attacks, which seriously threaten users' personal privacy and data security. To improve the resilience to attack technologies, structural information analysis has been widely applied in mobile malware detection. However, the rapid improvement of mobile applications has brought an impressive growth of their internal structure in scale and attack technologies. It makes the timely analysis of structural information and malicious feature generation a heavy burden. In this paper, we propose a new Android malware identification approach based on malicious subgraph mining to improve the detection performance of large-scale graph structure analysis. Firstly, function call graphs (FCGs), sensitive permissions, and application programming interfaces (APIs) are generated from the decompiled files of malware. Secondly, two kinds of malicious subgraphs are generated from malware's decompiled files and put into the feature set. At last, test applications' safety can be automatically identified and classified into malware families by matching their FCGs with malicious structural features. To evaluate our approach, a dataset of 11,520 malware and benign applications is established. Experimental results indicate that our approach has better performance than three previous works and Androguard.

1. Introduction

Mobile device has become an essential social network communication tool which stores a huge amount of user privacy data. Therefore, it attracts persistent malicious attacks. Due to the open source policy, Android has become the most popular operating system for mobile devices and has the largest market share. With the widespread use of Android applications, Google company is committed to maintain the safety of its official application market—Google Play Store [1]. Aiming at security problems, Google has used various strategies to fight against malicious attacks such as regularly scanning billions of installed mobile applications, providing remote security services for the mobile device, and isolating malicious websites to protect users. However, Google's large investment only blocks some of the malware that threatens Google Play Store. Many third-party application markets are still facing an increasing numbers of malware. A report [2] showed that 97% of the total number of mobile malware was related to Android platform in 2013,

up from 79% during the previous year and 66% in 2011. In 2015, the total number of Android malware rose to 884,774 [3]. In 2019, Android malware variants grew 31% in a year and the total number closed to 20 million [4].

A commonly used detection strategy adopted by commercial antivirus tools (such as Norton and Lookout) is collecting as many as possible malware and extracting signature code as features [5, 6]. Then, these features are used to match with the signature code that is extracted from target applications to identify malware. Although this strategy can achieve high detection accuracy and low false positive rate (FPR), it still faces two challenges: (1) lagging behind malicious attacks and cannot detect unknown malware and (2) minor changes of applications may lead to failure of the detection method.

To solve the problems, researchers begin to use more effective expert features to recognize malicious code, such as permissions, component information, and APIs [7–9]. Many of them also apply machine learning technology to improve the detection performance further. Although these

approaches have been proved effective, the grammatical feature-based detection methods are easily affected by code obfuscation and injection technologies.

Fortunately, many research works have [10–12] shown that high-level properties of code, especially structural features, can promote the resilience ability of repackaging and code obfuscation technologies. The most commonly used method is to compare the application's structural features with existing malicious ones. Thus, it is a well-known strategy to transform a graph matching problems into an isomorphism. However, the isomorphism problem has been proved to be a nondeterministic polynomial (NP) problem and may be very inefficient when the graphs are large. How to find malicious subgraphs efficiently is still a problem to be solved.

In this paper, we propose a new malware detection method based on malicious subgraph searching. The subgraphs are generated from FCGs of Android applications. During the training phase, the FCGs are generated from the malware of each malware family to get malicious structural features. It can improve the detection efficiency and help to analyze the homology and evolution of variant viruses. In the test phase, structural features are used to automatically detect malware and classify them into different families. In the evaluation phase, several experiments are designed to evaluate the detection performance and execution efficiency of our method.

According to the above descriptions, the main contributions of our method can be summarized as follows:

- (1) A new efficient Android malware detection method is proposed. In this method, the malicious features extracted from applications are structural. It has a positive effect on the resilience to code obfuscation and repackaging technologies in static analysis. At the same time, the method can find a class of similar variant samples that could be useful for malware detection and new variant analysis.
- (2) A fast common subgraph searching and matching algorithm based on nodes similarity calculation is designed. There are several well-known algorithms that can match subgraphs by using graph isomorphism calculation, such as VF2 [13] and graph edit distance algorithms [14]. However, it is a big challenge for them to successfully match a large number of subgraphs with complex structure and large scale. In this situation, our method has much better performance compared with the VF2 algorithm and so on.
- (3) The evaluation processes of our method are executed based on several datasets of 7520 malware and 4000 benign Android applications. Evaluation results indicate that our method has better detection ability than three previous works and Androguard.

The following parts of this paper are organized as follows. The relevant research studies of our topic are collected in Section 2. Section 3 introduces the whole architecture of our method. The specific execution process of our method is

discussed in Section 4. Section 5 applies some experiments to test the detection efficiency of our method. At last, the conclusion and future works are discussed in Section 6.

2. Related Works

Structural analysis has been widely used in Android malware detection methods. It can be divided into the following two main categories.

2.1. Static Structural Feature-Based Detection Methods. In the analysis of structural features, many research studies have been done to solve the NP problem caused by isomorphism algorithms. At the same time, new methods were designed to find malicious structural features effectively.

Crussell et al. [15] proposed a new method to identify the replication and clone behaviors of mobile applications called “DNADroid.” This method constructed function call graphs of applications at first. Then, the similarity values of function call graphs were calculated based on the VF2 algorithm. At last, these similarity values were used to evaluate the similarity of applications. In the experiment part, DNADroid was used to test the applications of their experimental dataset. It found that at least 114 applications had been cloned.

Xu et al. [16] proposed a malicious code detection method based on function call graphs. This method firstly extracted function call graphs from mobile applications. Secondly, graph edit distances were calculated based on applications' methods and methods' operation code. Finally, the similarity measurement of applications can be got according to the graph edit distance score. Experimental results showed that this method can identify variants of malware.

Zhang et al. [11] proposed a malicious code detection method based on semantics information called “DroidSIFT.” The method can extract function dependency graphs from mobile applications by implementing a graph generation tool on top of Soot [17]. Then, feature vector spaces were constructed according to the similarity of function dependency graphs. At last, these feature vector spaces were used to build two different classifiers to identify malicious code. In their experiment, DroidSIFT was used to test 13500 benign samples and 2200 malware. Experimental results showed that DroidSIFT could achieve 93% detection accuracy and 5.15% false positive rate (FPR).

Suarez-Tangil et al. [18] proposed a new malware detection method which can automatically extract code chunks (CCs) from the control flow graph of Android applications. Then, the CCs were used to be analyzed by the text mining model to classify malware into different families.

Hu et al. [19] proposed a static malware detection method called “MIGDroid.” The method extracted invocation graphs from Android applications and then divided them into subgraphs. By calculating threat scores of subgraphs, malicious code of applications can be found.

Niu et al. [20] built the opcode-level FCG of Android applications and used the long short-term memory model to

analyze malicious behaviors. The detection accuracy of the method is 97% based on their dataset. Gao et al. [21] used the invocator-invocator relationship of Android application's FCG to generate topological signatures. The malware detection method based on the topological signatures was designed and evaluated by 1249 malware and 49000 benign applications.

Sun et al. [22] proposed a new graph-based Android malware detection method called "DroidSim." DroidSim can construct component-based control flow graph (CB-CFG) by using APIs as nodes and control flow precedence order of Android components as edges. The evaluate similarity scores of CB-CFGs were used for malware detection. In their experiment, DroidSim achieved 96.6% detection accuracy on 121 benign Android applications and 706 malware.

Atici et al. [23] extracted static features from control flow graph-based and used machine learning classification to identify malware.

2.2. Dynamic Structural Feature-Based Detection Methods. John et al. [24] extracted the system call graph of the Android application and used graph convolutional nets to detect malware. The detection accuracy of their method is 92.3% on the experimental datasets with 2130 samples.

Zhang et al. [25] extracted the information of object reference graphs from the execution processes of Android applications to build an object reference information model. Then, a two-step malware detection method was designed based on the improved graph isomorphism algorithm.

Abdurrahman and Acarman [26] constructed API call graphs of Android applications and then transformed them into low-dimensional feature vectors. Finally, a deep neural network-based method was designed to detect malware.

Yerima et al. [27] researched the generation process of stateful events and designed a new method to improve the code coverage in dynamic analysis for malware detection.

Lin et al. [28] extracted features from behavior dependency graphs for machine learning classification process. Then, a prototype system was implemented to identify malware.

Xu et al. [29] proposed a new efficient representation of system call graph. Then, feature vector labels of the representations were used and optimized to improve the classification ability of the SVM algorithm.

Hou et al. [30] extracted dynamic behavior features from weighted directed graphs. Then, a deep learning model was applied to identify malware based on these features.

Although both static and dynamic methods can analyze structure features, our method is proposed based on static detection for the following two reasons: (1) static detection can get a complete function call graph without missing malicious subgraphs and (2) static detection does not need a virtual running environment, which means that it can detect a large number of viruses faster than dynamic detection. Differently from the above research studies, our method discards analyzing and matching large-scale graph structures directly. A new method which can iteratively construct structural malicious features from a single node is designed.

It can quickly locate the malicious code generated by code injection technology and identify variant viruses effectively.

3. Architecture Overview

The architecture of our malware detection method is depicted in Figure 1. The main steps are as follows:

- (1) *Decompile Process.* The "AndroidManifest.xml" and ".dex" files of the Android applications are used in our method. Thus, Android applications whose executable file named Android package (APK) files are decompiled by Androguard [31] to obtain these files.
- (2) *Static Information Collection.* Several Python script files are implemented on top of the Androguard tool. They are used to collect static information from the disassembled code files of step (1), including Android applications' methods, method call sequences, permissions, and APIs. Then, the FCG is constructed based on the methods' information. All these static information is output to text files in a uniform format, respectively.
- (3) *Structural Features Generation.* By analyzing the permissions of malware and benign applications, we choose the permissions that are used more frequently in malware as sensitive permissions. Then, two kinds of malicious structural features are constructed. The first one is a sensitive permission-based subgraph. It extracts sensitive permissions' relative APIs as initial nodes. The method call sequences which contain these initial nodes are used to construct structural features. The second one is the common subgraph of malware families. A nodes similarity-based subgraph searching method is designed. The method firstly searches the most similar node of the graphs, and then its similar adjacent nodes are searched based on the improved Kuhn-Munkres (KM) algorithm [32]. This process will be iteratively executed until the termination condition (introduced in Subsection 4.3) is met. To improve the execution efficiency, the maximum path length that contains the initial node is 3. Once all structural features are generated, they are put together as a malicious feature set.
- (4) *Malware Detection.* The application to be tested should be decompiled and generate its static information as step (1) and (2). The nodes similarity-based matching process is also implemented between the application's FCG and the feature set to identify malware.

4. Malicious Subgraphs Generation

This section introduces the details of the FCG and malicious subgraph feature generation processes as well as the nodes similarity-based matching method and improved KM algorithm.

4.1. FCG Generation. An Android application implements its operations based on methods and method call sequences. Thus, the function call graph contains the affluent behavior

FIGURE 1: The architecture of our method.

information of an application. As mentioned in the previous section, our method constructs the function call graph by implementing a Python script file on top of the Androguard. The Python script file needs to exhaustively search all methods which are likely to be ignored by indirect calls.

The first step is to get an Android application's packages, classes, and methods' information from different objects of decompiled files. In this process, Androguard will automatically assign a numeric label to each method. These numeric tags are fixed, which means they are not changed with multiple executions of decompiling operations. The second step is to search all methods and store them as a graph's nodes in a node set. The third step is to apply a depth-first searching process to find nodes' all related nodes and call relations. To finish this work, two more node sets are established to store nodes' parent nodes and child nodes and then continuously search the parent nodes of every node in the parent node set, as well as the child nodes of every node of the child node set. When all parent nodes and child nodes are searched, all call sequences of a node are collected. At last, all call sequences are joined based on the numeric labels of nodes to construct the complete function call graph.

4.2. Sensitive Permission-Based Subgraph Extraction.

Android has set up a permission mechanism to control the access behaviors of applications. It can limit the excessive abuse of user privacy information and system resources by application developers. For example, if a program wants to obtain the information of Wireless Fidelity (WiFi) network status, its request should be written in the "AndroidManifest.xml" file, as follows:

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE">
</uses-permission>
```

Permissions can be used both by malware and benign applications. Therefore, a malicious score is assigned to each permission by calculating the frequency of occurrence of a

permission between malware and benign applications. The top 20 permissions with the highest malicious score are treated as sensitive permissions in our method. At the same time, the APIs which are related to the sensitive permissions can be found, as shown in Figures 2 and 3.

In the framework provided by Android, a lot of drivers and functions are encapsulated in the bottom layer. Users can invoke these drivers by calling the APIs in their user-defined methods. Thus, the method call sequences which contain these user-defined methods can be found and used to construct sensitive permission-based subgraphs.

4.3. Generation of Common Subgraph of Malware Families.

In our method, the common subgraph is generated based on the nodes similarity calculation process. Considering the different definitions of similarity, there are different computational processes to get the value of similarity, for example, the use of graph isomorphism algorithms to evaluate the similarity based on the graph's structural features or calculate the similar distances of string features which are extracted from structural information and so on.

In Android malware detection, a large number of malicious variants are generated by repackaging technologies. Although many of them also have been processed by obfuscation technologies, they still have partial similarities in internal structures. In addition, code injection technologies can inject malicious code into various kinds of benign applications. It makes completely unrelated applications implement similar attacks. This situation is particularly evident in the variants of the same malware family.

To find malicious code of applications, the nodes similarity calculation process is proposed.

Suppose that a graph is defined as $G = (V, E)$, where V represents the node set and E represents the edge set. If there are two graphs G_A and G_B , node $v_a \in G_A$ and node $v_b \in G_B$. v_{ai} is the i th adjacency node of v_a , and v_{bi} is the i th adjacency node of v_b . The similarity Sim_{ab} of two nodes can be defined as follows:


```
['android.permission.WAKE_LOCK','android.permission.WRITE_APN_SETTINGS',
'android.permission.RECEIVE_BOOT_COMPLETED',
'android.permission.ACCESS_NETWORK_STATE','android.permission.READ_PHONE_STATE',
'android.permission.WRITE_EXTERNAL_STORAGE','android.permission.INTERNET',
'android.permission.MODIFY_PHONE_STATE']
```

FIGURE 2: The permissions of “6a0bfabcc1cce2a5424313b34ca967fbc8f98bea.apk.”

```
READ_PHONE_STATE :
1 Lcom/xxx/yyy/MyService; -> onCreate()V (0x16) ---->
Landroid/telephony/TelephonyManager; -> getDevice()Ljava/lang/String;
1 Lcom/xxx/yyy/MyService; -> onCreate()V (0x22) ---->
Landroid/telephony/TelephonyManager; -> getSubscriberId()Ljava/lang/String;
```

FIGURE 3: The APIs related to READ_PHONE_STATE permission of “6a0bfabcc1cce2a5424313b34ca967fbc8f98bea.apk.”

$$\Delta w(v_i) = \alpha \cdot \frac{|D_a^{\text{in}} - D_b^{\text{in}}|}{D_a^{\text{in}} + D_b^{\text{in}}} + \beta \cdot \frac{|D_a^{\text{out}} - D_b^{\text{out}}|}{D_a^{\text{out}} + D_b^{\text{out}}}, \quad (1)$$

$$E_{ab} = \max(S(V_A, V_B)),$$

$$\text{Sim}_{ab} = (\Delta w(v_i), E_{ab}),$$

where D_a^{in} and D_b^{in} are the in-degree of nodes v_a and v_b , D_a^{out} and D_b^{out} are the out-degree of nodes v_a and v_b , $\Delta w(v_i)$ is the node similarity value between v_a and v_b , $S(V_A, V_B)$ is the similarity value between the adjacency nodes of V_A and V_B , and E_{ab} is the optimal value of $S(V_A, V_B)$. The calculation process of E_{ab} will be introduced in Subsection 4.4 in detail. α and β whose values between (0, 1) are constant coefficients are used to optimize the matching efficiency which can be determined in the training phase.

According to the above definitions, our common subgraph searching method is designed as follows:

- (1) Search different function call graphs of malware in the same family, and find the most similar nodes as initial nodes of the structural malicious feature.
- (2) Continue to search initial nodes' the most similar adjacency nodes, and then add new-found nodes and edges to the existing subgraphs. This process will be iteratively executed once the values of $1/\Delta w(v_i)$ or E_{ab} are less than the thresholds whose range is (0.9, 1). When the iterative process has been stopped, the structural feature of a malware family is found.

4.4. Optimal Matching Strategy of Nodes. As mentioned in the previous subsection, a node probably has many adjacency nodes to be matched. It makes the calculation of E_{ab} a multimatching problem. To find the optimal matching result, a new matching method is designed.

The first step is to take a pretreatment. During the subgraph searching process, once $1/\Delta w(v_i)$ is less than the threshold, our method will continue to calculate the value of E_{ab} . It is necessary to note that there are many adjacency nodes which are leaf nodes. To reduce computational complexity, the similarity value ΔL of these leaf nodes is calculated as follows:

$$\Delta L = \frac{|x - y|}{x + y}, \quad (2)$$

where x is the number of leaf nodes of node v_a and y is the number of leaf nodes of node v_b . If ΔL is less than the threshold whose range is (0.9, 1), the leaf adjacency nodes will be deleted and will go to the next step. Otherwise, it means V_A and V_B cannot be matched, and the subgraph searching process will be terminated.

The second step is to construct a bipartite graph $G' = ((V'_A, V'_B), E')$, where V'_A is the adjacency nodes set of v_a , V'_B is adjacency nodes set of v_b ($V'_A \cap V'_B = \emptyset$), and E' is edges set between V'_A and V'_B .

As shown in Figure 4, v_{ai} is the i th adjacency node of v_a , v_{bj} is the j th adjacency node of v_b , and $\exists e_{ij} \in E'$ represents a possible connection between v_{ai} and v_{bj} . The weight of e_{ij} is $w(e_{ij})$. Equation (3) represents the total weight of a matching W_T .

$$W_T = \sum_{e_{ij} \in E'} w(e_{ij}). \quad (3)$$

Thus, if there are more than one matche, the max W_T ($W_{\max}$) can be treated as the optimal matching, as shown in the following:

$$W_{\max} = \max \sum_{e_{ij} \in E'} w(e_{ij}). \quad (4)$$

To get $W_{\max}$, the third step is to assign a weight $w(e_{ij})$ to the edges of E' as shown in Table 1.

Table 1 shows the edge-weighted matrix of E' , where m is the row number and n is the column number. If ($m! = n$), our method will always make ($m < n$) and add ($m - n$) virtual nodes whose value of $w(e_{ij})$ is 0. The edge-weighted matrix generation process can be described in Algorithm 1.

According to the edge-weighted matrix, the maximum weight $W_{\max}$ of G' can be calculated based on the improved KM algorithm. The KM algorithm assigns each node a label value and transforms the maximum weight finding process into the complete match searching. Suppose that the label value of v_{ai} is $l(v_{ai})$ and the label value of v_{bj} is $l(v_{bj})$, the condition $l(v_{ai}) + l(v_{bj}) \geq w(e_{ij})$ should be met during the whole execution of the KM algorithm.

Although the KM algorithm can find $W_{\max}$ successfully, the time complexity of KM is (O^4). It can be found that the efficiency of the algorithm is obviously reduced when the number of nodes exceeds 1000 as shown in Figure 5. To solve this problem, the KM algorithm is improved as follows:

- (1) Calculate the label value of nodes based on edge's weight. Let $w(e_{ij}) = l(v_{ai}) + l(v_{bj})$. In this step, let $l(v_{ai}) = \max(w(e_{ij}))$ and $l(v_{bj}) = 0$.

FIGURE 4: The bipartite graph of adjacency nodes.

TABLE 1: Edge-weighted matrix of E' .

	v_{b1}	v_{b2}	$\dots$	v_{bn}
v_{a1}	$w(e_{ij})$	$\dots$	$\dots$	$w(e_{ij})$
v_{a2}	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
v_{am}	$w(e_{ij})$	$\dots$	$\dots$	$w(e_{ij})$

Input: Bipartite graph of adjacency nodes $G' = ((V_A', V_B'), E')$ and the weight of all adjacency nodes.

Repeat:

Choose $\exists v_{ai} \in V_A'$ and $\exists v_{bj} \in V_B'$,

if (v_{ai} or v_{bj} is not a virtual node)

if ($w(v_{ai}) - w(v_{bj}) \neq 0$)

Calculate the weight of edge:

$$w(e_{ij}) = (w(v_{ai}) + w(v_{bj})) / |w(v_{ai}) - w(v_{bj})|$$

Insert $w(e_{ij})$ into the edge-weighted matrix

else

$$w(e_{ij}) = 0$$

Insert $w(e_{ij})$ into the edge-weighted matrix

Until: Every edge is assigned a weight.

Output: The edge-weighted matrix.

ALGORITHM 1: Edge-weighted matrix generation.

FIGURE 5: The execution time of our method on different sizes of applications.

- (2) Prepare for augmenting path searching of equal subgraphs. Suppose that M represents a perfect match of the bipartite graph, the augmenting paths are searched to find the optimal matching which makes the sum of edges' weights of M maximum. However, the lack of required edges usually leads to the inefficiency of augmenting path search. To solve this problem, several improvements are taken as follows:

Firstly, adjust the label values of nodes. Let

$$\begin{aligned} l(v_{ai}) - \eta, \\ l(v_{bj}) + \eta, \\ \eta = \min(l'(v_{ai}) + kl'(v_{aj}) - w(e_{ij})), \end{aligned} \quad (5)$$

where $v_{ai} \in V'_A$, $v_{bj} \in V'_B$, and $v_{aj} \in V'_B$. $l'(v_{ai})$ is the label value of v_{ai} where v_{ai} belongs to the searched augmenting path. $l'(v_{aj})$ is the label value of v_{aj} where v_{aj} has not been in the searched augmenting path yet. This change can make more nodes and edges meet the condition $l(v_{ai}) + l(v_{bj}) = w(e_{ij})$. Secondly, define an array named "slack"; let $slack[i] = \infty$ before the augmenting path searching each time. When $w(e_{ij})$ is searched, let $slack[i] = \min(l'(v_{ai}) + l'(v_{bj}) - w(e_{ij}))$, where v_{ai} belongs to the searched augmenting path. v_{bj} has not been in the searched augmenting path yet. This change makes the algorithm never search the edge repetitively until an augmenting path is successfully found.

- (3) Search the augmenting path by the Hungarian algorithm to find M .
 (4) If M is not found, change the label values of accessed nodes as follows:

$$\begin{aligned} l(v_{ai}) - \eta', \\ l(v_{bj}) + \eta', \\ \eta' = \min(slack[j]), \\ slack[j] - \eta', \end{aligned} \quad (6)$$

where $j \in V'_B$ and j has not been in the searched augmenting path yet.

- (5) Execute steps (3)-(4) repetitively until the perfect matching of the equal subgraph is found.

According to the above steps, on the basis of the paths that have been searched, only the newly added edges are searched to ensure that each edge is searched once.

4.5. Malware Detection. When both sensitive permission-based subgraphs and common subgraph of each family are collected, our malicious feature set can be generated. The detection result of a test application can be obtained by matching its FCG with the feature set. The matching steps are as follows. Firstly, find the similar nodes of structural features' initial nodes in test application's FCG. Secondly, continue to

match the initial nodes' adjacent nodes according to the nodes similarity calculation process which is introduced in Subsections 4.3 and 4.4. The FCG can be judged to contain a malicious structure if all nodes of a malicious structure feature are successfully matched. At the same time, the malware family to which the application belongs can be detected. Otherwise, the application is judged to be safe.

In order to promote execution speed, the family feature straining process is taken off-line.

5. Experiments and Evaluation

5.1. Dataset and Evaluation Metrics. To test the effectiveness of our method more comprehensively, three malware datasets and one benign dataset are collected. The first dataset (dataset 1) is Genome Project which was collected by Jiang and Zhou [33] in 2012. It contains 1247 malware samples of 49 malware families. This dataset involves many kinds of attack techniques such as repackaging, remote control, personal information stealing, and update attack techniques. The authors' experimental results showed that four famous commercial malware detection tools can only achieve unsatisfied detection rate on this dataset (20–79%).

The second dataset (dataset 2) is Drebin which contains 5560 malware samples of 179 families. Its malware samples were collected from mobile application markets in Russia, China, and so on [34].

These two datasets are chosen because of their high frequency of use in many former works [11, 21, 24, 35–37].

Malware dataset 3 contains malicious apps collected from the virus share [29] and Android malware dataset (AndMal) [38] which contains 713 malware of 42 families.

The benign apps are collected from Google Play Store, 360 application market [1, 33, 38], and so on. All of them are checked by frequently used antivirus softwares McAfee and Kaspersky to ensure their safety, as shown in Table 2.

Based on these datasets, the performance of detection methods is evaluated by True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN), accuracy, FPR, TPR, recall, and precision. They can be defined as follows.

The row of Table 3 means the actual type of applications. The column means the prediction type of the detection method.

$$\begin{aligned} \text{Accuracy} &= \frac{TP + TN}{TP + FN + TN + FP}, \\ \text{Precision} &= \frac{TP}{TP + FP}, \\ \text{Recall} &= \frac{TP}{TP + FN}, \\ \text{FPR} &= \frac{FP}{TN + FP}, \\ \text{TPR} &= \frac{TP}{TP + FN}. \end{aligned} \quad (7)$$

The following experiment is organized into four parts: firstly, the malware family identification ability of our

TABLE 2: Information of dataset.

Name	Sample number	Family/category number	Alias in this paper
Genome project	1247	49	Dataset 1
Drebin	5560	179	Dataset 2
AndMal	713	42	Dataset 3
Benign	4000	20	Benign dataset

TABLE 3: Matrix of metrics.

	Malware	Benign
Malware	TP	FN
Benign	FP	TN

method is compared with several former works in Subsection 5.2; secondly, the FPR of our method in malware family detection is evaluated in Subsection 5.3; thirdly, a 3-fold detection process is implemented to evaluate the detection ability of our method on unknown malware in Subsection 5.4; and finally, the runtime performance of our method is discussed in Subsection 5.5.

5.2. Compare with Graph-Based Detection Methods. As mentioned in Section 2, many research studies have applied their malware detection experiments on Genome dataset. Thus, the detection ability of our method is compared with three efficient former graph-based detection models and Androguard tool in this subsection. The first method is Dendroid [18]. Dendroid put a detailed analysis on CFGs of Android applications and designed a string feature extraction strategy. It successfully transformed the subgraph comparison into text mining. By comparing the similarity of string features, Dendroid can classify similar applications into the same malware family. The second one is “MIGDroid” [19], which is also a subgraph analysis-based detection method. Differently from Dendroid, MIGDroid calculated subgraphs’ threat scores according to the sensitive features belonging to them. By evaluating the threat scores, MIGDroid can identify malware of each family. The third method is [23] (it is named “CFG-based method” in this experiment). This method improved the text mining model of Dendroid by adding machine learning classification into it.

Table 4 shows the detection rates among these former works and ours. The experimental results indicate that Androguard gets the worst detection rate because it is over-reliant on signatures in malware detection. Without relative signatures, it cannot identify malware even if the variants are very similar to the existing malware. MIGDroid is more easily influenced by the APIs. For example, the variants of “zHash” family are added a lot of garbage code. These garbage codes have close connections with the rest normal codes. Therefore, identifying which part of the garbage codes is benign or malicious is difficult for MIGDroid. Dendroid classifies malware into families based on CCs. However, the CCs’ distribution is irregular. It means some families may have a large number of CCs, but some families may have few. At the same time, when families are similar to each other, the

classification performance of Dendroid may decline, such as DroidKungFuX, BaseBridge, and AnserverBot. The CFG-based method has good performance on the detection of DroidKungFu family, but the average detection rate is lower than Dendroid. Our method has the highest average detection rate, which means it can locate the malicious structural feature with high efficiency. However, the accuracy of our method is still reduced when it is used to detect high similar families, such as DroidKungFu families.

5.3. The TPR and FPR Evaluation of Malware Family Classification. In addition to the detection rate, TPR and FPR are also important metrics. The TPR and FPR of our method among malware families are evaluated in two parts. Firstly, the identification result is evaluated by the 8 most closely related malware families. They are BaseBridge (BB), AnserverBot (AB), DroidKungFu (DK), GoldDream (GD), DroidDreamLight (DDL), Pjapps (PJ), DroidDream (DD), Plankton (PK), and Zsone (ZS). These families are chosen because they may cause higher FPR than other unrelated families [25].

Table 5 shows the confusion matrix of the family classification. The column represents actual families, and the row represents the identification results. Experimental result shows that the highest FPR is 3.2%. It only appears between BaseBridge and AnserverBot since they contain the virus variants of the same ancestor.

Secondly, 30 representative families of dataset 1 and dataset 3 are selected to evaluate their FPR and TPR in malware detection. Table 6 shows the FPR of our method on each family. The experimental result shows that the highest FPR is 3.8%, and the average FPR of these malware families is 1.1%.

5.4. Evaluation with Unknown Applications. In this subsection, the detection ability of our method on the unknown malware is evaluated. Thus, the experimental dataset is divided into two different parts: 80 percent random malware of each family (6016 samples with 221 families of dataset 1–dataset 3) are chosen as the training set. The rest applications (1504 malware and 4000 benign samples) are chosen as the unknown test set. In order to evaluate the stability of our detection method, this dataset division process is executed three times to get three different training sets and unknown test sets. The detection result of each time (named result 1, result 2, and result 3) is shown in Tables 7–9.

Tables 7–9 are the confusion matrixes of each detection result. The row is the actual type of applications, and the column is the prediction type of the detection method. Table 10 provides the metrics comparison among result 1 to result 3. It shows that the maximum difference of accuracy, recall, and precision is 1%, 0.1%, and 2.1%, respectively. This result indicates that the detection rate is stable, but it is still affected by different training samples.

5.5. Runtime Performance. In this subsection, the runtime performance of our method is evaluated. The execution efficiency of our method is mainly affected by the size of the

TABLE 4: Comparison of detection rates among malware families.

Family	Dendroid (%)	MIGDroid (%)	Androguard (%)	CFG-based method (%)	Ours (%)
ADRD	100	100	59.09	95.50	100
AnserverBot	96.70	100	0	98.93	96.80
BeanBot	100	100	0	62.5	100
Bgserv	100	100	0	88.9	100
DroidDream	100	100	93.75	93.75	100
DroidDreamLight	100	100	28.26	100	100
DroidKungFu1	88.24	97.06	0	97.05	100
DroidKungFu2	80	60	0	96.66	96.66
DroidKungFu3	92.56	99.35	0	100	100
DroidKungFu4	83.33	72.92	0	100	100
Geinimi	100	100	97.10	100	100
GoldDream	100	100	0	93.75	100
jSMShider	100	100	0	100	100
Pjapps	100	72.40	41.38	97.78	98.27
Plankton	100	90	18.18	90.91	100
YZHC	100	100	95.45	95.45	100
Zsone	100	100	100	100	100
Asroot	100	50	0	37.50	100
BaseBridge	92.80	44.26	0	99.18	98.36
KMin	100	21.15	78.86	100	100
RogueSPPush	100	55.56	100	98.88	100
SndApps	100	0	100	100	100
zHash	100	0	0	90.91	100
Average	97.11	76.63	35.30	92.94	99.56

TABLE 5: The confusion matrix for classification of 8 families (%).

	BB	AB	DK	GD	DDL	PJ	DD	PK	ZS
BB	98.4	1.6	0	0	0	0	0	0	0
AB	3.2	96.8	0	0	0	0	0	0	0
DK	0	0	100	0	0	0	0	0	0
GD	0	0	0	100	0	0	0	0	0
DDL	0	0	0	0	100	0	0	0	0
PJ	0	0	0	0	0	100	0	0	0
DD	0	0	0	0	0	0	100	0	0
PK	0	0	0	0	0	0	0	100	0
ZS	0	0	0	0	0	0	0	0	100

TABLE 6: The FPR of 30 typical malware families of the dataset.

Family name	TPR/FPR (%)	Family name	TPR/FPR (%)	Family name	TPR/FPR (%)
DroidDream	100/1.5	RogueSPPush	100/2	Koodous	100/0
DroidDreamLight	100/1	SndApps	100/2	Mobidash	100/2.2
Geinimi	100/1.4	zHash	100/0	RansomBO	100/0
jSMShider	100/0	Dowgin	100/3.8	Svpeng	100/0
Plankton	100/1.3	Edwin	100/0	Koler	100/2.4
YZHC	100/0	Feiwo	100/0	Pletor	100/0
Zsone	100/0	Gooligan	100/2.5	PornDroid	100/2.3
Asroot	100/0	Kemoge	100/0	Charger	100/3.7
Bgserv	100/0	Fakemart	100/3	Jisut	100/0
KMin	100/0	Jifake	100/3.1	MazarBot	100/0
Average	1.1				

TABLE 7: The confusion matrix of result 1.

	Malware	Benign
Malware	1475	29
Benign	40	3960

TABLE 8: The confusion matrix of result 2.

	Malware	Benign
Malware	1455	49
Benign	73	3927

TABLE 9: The confusion matrix of result 3.

	Malware	Benign
Malware	1459	45
Benign	68	3932

TABLE 10: Comparison of result 1–result 3.

Detection result	Accuracy	Recall	Precision
Result 1	0.987	0.968	0.973
Result 2	0.977	0.967	0.952
Result 3	0.979	0.97	0.955

TABLE 11: Information of the different sizes of applications.

Label	Average number of nodes	Average number of edges
A	180	340
B	550	1020
C	690	1800
D	1110	2800
E	1600	3730
F	2500	6800
G	2900	7000
H	4400	10300
I	5620	16030
J	8680	20700

FCGs. Thus, the test dataset of this experiment is established based on different sizes of applications' FCGs. Each application's FCG size is represented by the number of nodes and edges, which are divided into 10 levels, as shown in Table 11.

Figure 5 shows the execution time of our method on these applications. Experimental result shows that the execution time of our method rises from 0.003 seconds to 0.524 seconds, whereas the execution time of the KM algorithm rises from 0.006 seconds to 44.5 seconds. It indicates that the efficiency of our method has been improved obviously.

6. Conclusion

In this paper, a new structural feature-based Android malware detection method is introduced. The method can automatically extract static features from applications and generate FCGs and sensitive permissions. Then, sensitive permission-based subgraphs and the common subgraph of each malware family are constructed as malicious features. At last, unknown applications' safety can be identified by these malicious features. This method is proved useful from the following three aspects. First, the family detection rate of our method is evaluated by comparing with three former works and Androguard. Experimental results indicate that our method can get higher detection accuracy among many representative malware families of our dataset. Second, the detection ability of unknown malware is evaluated. Third,

the runtime performance of our method is evaluated by different sizes of applications. All evaluation results indicate that our method can achieve good performance on different kinds of malware with various attack technologies.

Although our method is efficient, the experimental results also indicate that our method can be improved in many ways. Specifically, the next work can be taken in the following two directions: (1) research robust defense methods for malicious obfuscation technologies which can modify the code structures of malicious subgraphs and (2) design more efficient models to store the expanding structural features. Thus, more efficient graph analysis models or algorithms should be designed in future works. Moreover, more heuristic features are needed to cope with the rapid improvement of malicious attack technologies.

Data Availability

The data used to support this study are from previously reported studies [33], [34], and [38].

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the Sichuan Science and Technology Program (grant no. 2021JDRC0075), Fundamental Research Funds for the Central Universities, Southwest Minzu University (grant no. 2020NZD02), and Chengdu Science and Technology Program (grant no. 2021-GH03-00001-HZ).

References

- [1] "Google play store," 2019, <https://play.google.com/store/>.
- [2] G. Kelly, "Report: 97% of mobile malware is on android. this is the easy way you stay safe," 2014, <http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/>.
- [3] "Kaspersky report," 2015, <https://www.kaspersky.com/about/press-releases?rel=1&sel=date>.
- [4] "Nokia threat intelligence report," 2019, <https://pages.nokia.com/T003B6-Threat-Intelligence-Report-2019.html>.
- [5] V. Deepak and H. Guoning, "Efficient signature based malware detection on mobile devices," *Mobile Information Systems*, vol. 4, no. 1, pp. 33–49, 2014.
- [6] R. H. Niazi, J. A. Shamsi, T. Waseem et al., "Signature-based detection of privilege-escalation attacks on Android," in *Proceedings of the Conference on Information Assurance and Cyber Security*, pp. 44–49, Rawalpindi, Pakistan, February 2016.
- [7] İ. A. Doğru and M. önder, "AppPerm analyzer: malware detection system based on android permissions and permission groups," *International Journal of Software Engineering and Knowledge Engineering*, vol. 30, no. 4, pp. 427–450, 2020.
- [8] M. Scalas, D. Maiorca, F. Mercaldo, C. A. Visaggio, F. Martinelli, and G. Giacinto, "On the effectiveness of system

- API-related information for Android ransomware detection,” *Computers & Security*, vol. 86, pp. 168–182, 2019.
- [9] O. Yildiz and I. A. Dogru, “Permission-based android malware detection system using feature selection with genetic algorithm,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 2, pp. 245–262, 2019.
 - [10] A. Bhattacharya, R. T. Goswami, and K. Mukherjee, “A feature selection technique based on rough set and improvised PSO algorithm (PSORS-FS) for permission based detection of Android malwares,” *International Journal of Machine Learning and Cybernetics*, vol. 10, no. 7, pp. 1893–1907, 2019.
 - [11] M. Zhang, Y. Duan, H. Yin et al., “Semantics-aware Android malware classification using weighted contextual API dependency graphs,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1105–1116, Scottsdale, AZ, USA, November 2014.
 - [12] L. Taheri, A. F. A. Kadir, and A. H. Lashkari, “Extensible android malware detection and family classification using network-flows and API-calls,” in *Proceedings of the 2019 International Carnahan Conference on Security Technology (ICCST)*, Chennai, India, October 2019.
 - [13] W. Huanran, H. Hui, and Z. Weizhe, “Demadroid: object reference graph-based malware detection in android,” *Security and Communication Networks*, vol. 2018, pp. 1–16, Article ID 7064131, 2018.
 - [14] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke, “Approximation of graph edit distance based on Hausdorff matching,” *Pattern Recognition*, vol. 48, no. 2, pp. 331–343, 2015.
 - [15] J. Crussell, C. Gibler, and H. Chen, “Attack of the clones: detecting cloned applications on android markets,” *European Symposium on Research in Computer Security*, vol. 81, no. 13, pp. 2454–2456, 2012.
 - [16] M. Xu, L. Wu, S. Qi et al., “A similarity metric method of obfuscated malware using function-call graph,” *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 1, pp. 35–47, 2013.
 - [17] “Soot: a java optimization framework,” 2019, <https://gitee.com/alexbill/soot>.
 - [18] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, “Dendroid: a text mining approach to analyzing and classifying code structures in Android malware families,” *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.
 - [19] W. Hu, J. Tao, X. Ma et al., “MIGDroid: detecting APP-Repackaging Android malware via method invocation graph,” in *Proceedings of the Conference on Computer Communication and Networks*, pp. 1–7, IEEE, Washington, DC, USA, August 2014.
 - [20] W. Niu, R. Cao, X. Zhang, K. Ding, K. Zhang, and T. Li, “OpCode-level function call graph based android malware classification using deep learning,” *Sensors*, vol. 20, no. 13, p. 3645, 2020.
 - [21] T. Gao, W. Peng, D. Sisodia, T. K. Saha, F. Li, and M. Al Hasan, “Android malware detection via graphlet sampling,” *IEEE Transactions on Mobile Computing*, vol. 18, no. 12, pp. 2754–2767, 2019.
 - [22] X. Sun, Y. Zhongyang, Z. Xin et al., “Detecting code reuse in android applications using component-based control flow graph,” *IFIP Advances in Information & Communication Technology*, vol. 428, pp. 142–155, 2016.
 - [23] M. A. Atici, S. Sagiroglu, and I. A. Dogru, “Android malware analysis approach based on control flow graphs and machine learning algorithms,” in *Proceedings of the 2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, IEEE, Little Rock, AR, USA, May 2016.
 - [24] T. S. John, T. Thomas, and S. Emmanuel, “Graph convolutional networks for android malware detection with system call graphs,” in *Proceedings of the 2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, IEEE, Guwahati, India, April 2020.
 - [25] W. Zhang, H. Wang, H. He, and P. Liu, “DAMBA: detecting android malware by ORGB analysis,” *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 55–69, 2020.
 - [26] P. Abdurrahman and T. Acarman, “Deep learning for effective Android malware detection using API call graph embeddings,” *Soft Computing*, vol. 24, no. 2, pp. 1027–1043, 2020.
 - [27] S. Y. Yerima, M. K. Alzaylaee, and S. Sezer, “Machine learning-based dynamic analysis of Android apps with improved code coverage,” *EURASIP Journal on Information Security*, vol. 4, 2019.
 - [28] Z. Lin, R. Wang, X. Jia et al., “Classifying android malware with dynamic behavior dependency graphs,” in *Proceedings of the 2016 IEEE Trustcom/BigDataSE/I SPA*, IEEE, Tianjin, China, August 2016.
 - [29] L. Xu, D. Zhang, M. A. Alvarez et al., “Dynamic android malware classification using graph-based representations,” in *Proceedings of the 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, IEEE, Tianjin, China, August 2016.
 - [30] S. Hou, A. Saas, L. Chen et al., “Deep4MalDroid: a deep learning framework for android malware detection based on linux kernel system call graphs,” in *Proceedings of the 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WTW)*, ACM, Omaha, NE, USA, October 2016.
 - [31] “Androguard project,” 2016, <http://code.google.com/p/androguard/>.
 - [32] Y. Zhou and J. Kuang, “A sort method to enhance significant spectral components of test set,” in *Proceedings of the 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pp. 2147–2151, Changsha, China, August 2016.
 - [33] X. Jiang and Y. Zhou, “Dissecting android malware: characterization and evolution,” in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, CA, USA, April 2012.
 - [34] D. Arp, M. Spreitzenbarth, M. Hübner et al., “DREBIN: effective and explainable detection of android malware in your pocket,” in *Proceedings of the Network & Distributed System Security Symposium*, San Diego, CA, USA, February 2014.
 - [35] S. Arzt, S. Rasthofer, C. Fritz et al., “FlowDroid,” *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
 - [36] M. Grace, Y. Zhou, Z. Qiang et al., “RiskRanker: scalable and accurate zero-day android malware detection,” in *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services*, pp. 281–294, Istanbul, Turkey, June 2012.
 - [37] V. Rastogi, Y. Chen, and X. Jiang, “Catch me if you can: evaluating android anti-malware against transformation attacks,” *IEEE Transactions on Information Forensics & Security*, vol. 9, no. 1, pp. 99–108, 2013.
 - [38] A. H. Lashkari, A. F. A. Kadir, L. Taheri et al., “Toward developing a systematic approach to generate Benchmark android malware datasets and classification,” in *Proceedings of the 2018 International Carnahan Conference on Security Technology (ICCST)*, Montreal, Canada, October 2018.