Hindawi
Security and Communication Networks
Volume 2022, Article ID 8398591, 11 pages
https://doi.org/10.1155/2022/8398591

Research Article

FGL_Droid: An Efficient Android Malware Detection Method Based on Hybrid Analysis

Weiping Wang,[1] **Congmin Ren**,[1] **Hong Song**,[1] **Shigeng Zhang**,[1,2] **and Pengfei Liu**[1]

[1]*School of Computer Science and Engineering, Central South University, Changsha, Hunan, China*
[2]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*

Correspondence should be addressed to Hong Song; songhong@csu.edu.cn

Received 22 December 2021; Revised 3 March 2022; Accepted 14 March 2022; Published 28 April 2022

Academic Editor: Weizhi Meng

With the popularity of Android intelligent terminals, malicious applications targeting Android platform are growing rapidly. Therefore, efficient and accurate detection of Android malicious software becomes particularly important. Dynamic API call sequences are widely used in Android malware detection because they can reflect the behaviours of applications accurately. However, the raw dynamic API call sequences are very usually too long to be directly used, and most existing works just use a truncated segment of the sequence or statistical features of the sequence to perform malware detection, which loses the execution order information of applications and consequently results in high false alarm rate. In this work, we propose a method that transforms the dynamic API call sequence into a function call graph, which retains most of the application execution order information with significantly reduced sequence size. To compensate for the missed behaviour information during the transformation, the advanced features of permission requests extracted from the application are utilized. We then propose FGL_Droid, which fusions the transformed function call graph feature and the extracted permission request feature to perform accurate malware detection. Experiments on benchmark dataset show that FGL_Droid achieves a high detection accuracy of 0.975 and a high F-score of 0.978, which are better than the existing methods.

1. Introduction

To prevent the threats caused by Android malware including financial loss to users, information leakage, and system damage, an efficient and accurate detection scheme is urgently needed. In recent years, Android operating system has been growing at an alarming rate. As of 2020, approximately 3 billion Android-based devices were shipped, accounting for 80% of all mobile operating systems [1]. Therefore, Android devices have become a popular target for malware developers. In 2020, Kaspersky mobile products and technologies detected 5,683,694 malicious installation packages, increasing 2.1 million from last year, 156,710 new mobile banking Trojans, and 20,708 new mobile ransomware Trojans [2]. In order to protect the property and information security of Android device users, it is urgent to provide an efficient and accurate malware detection method.

A large number of companies and researchers have conducted research on malware detection. The mainstream detection scheme mainly includes two categories: static analysis [3–6] and dynamic analysis [7–13]. Static analysis does not need to execute the application. It extracts features from the APK file through reverse engineering, such as Permissions [3], API calls [14], bytecodes, and other static features [4], and then performs malicious detection based on these features. However, with the advancement of technology, many malwares begin to use code obfuscation or dynamic loading techniques to hide static features, which leads static analysis schemes to be completely ineffective. On the contrary, dynamic analysis needs to actually run the program and collect information about its behaviour at running time [15]. By analysing the application's behaviour, dynamic analysis can achieve higher accuracy and robustness than static analysis. In this paper, we focus on dynamic analysis.

Dynamic API call sequence can describe the behaviour information of the application, which contains all the operations during application execution (e.g., accessing network, sending SMS, etc.). It is an important data for application behaviour analysis and has been applied by many researchers in Android malicious judgment. However, in order to trigger malicious behaviour completely, a huge number of detective operations are required, which will produce distinctly long API call sequences, reaching millions of levels. It is a great challenge to process the huge amount of dynamic API call sequence. According to our statistics on dynamic API call sequences, the average length of collected dynamic API sequences is 1.698 million, and the number of different API types is 25834. There are two problems in using dynamic API call sequence as the feature for malicious determination: (1) The large amount of data makes it difficult to find the behaviour information of the application. (2) There are many kinds of features, which makes the model easy to overfit.

Several related works have used dynamic API call sequence to detect Android malware; the main challenge is how to process dynamic API call sequence to reduce the amount of data size. The current solutions are broadly categorized into two types: using statistical features of dynamic API call sequence and truncating a fixed-length subsequence of dynamic API call sequence. The methods of using statistical features of dynamic API call sequence have an advantage that the computational overheads are relatively low, but these methods cannot save the order information of the APIs. The methods of truncating a fixed-length subsequence of dynamic API call sequence can save the order information; however, these methods will lose most of the sequence information, while the computational overheads are extremely high.

In this paper, we propose a scheme to transform the API call sequence into a function call graph. It can convert a million levels of dynamic API call sequence into a directed and edge-weighted graph structure with only a few nodes, which can both preserve information about the order of the API calls and greatly reduce the scale of data. We first replace each API with its corresponding function class to get the function call sequence and then convert the function call sequence to function call graph. Coarse-grained substitution loses part of semantic information. Therefore, we use advance feature Permission to make up for this part of semantic information. We use GCN (graph convolutional network) model to extract application's behaviour features and concatenate these features with permission feature to construct the final feature vector. Then we send the concatenated feature into LR (logistic regression) model to obtain classification result.

Our method can effectively solve the problem of excessive data volume of dynamic behaviour information. At the same time, the detection accuracy is better compared to the existing methods. The main contributions of this paper include the three following points:

(a) We developed a dynamic behaviour capture tool that integrates APE [16], which can capture API calls during application running
(b) We propose a method to convert a dynamic API call sequence into a function call graph, which can save the behaviour information of the application while reducing the amount of data
(c) We design a fusion model FGL_Droid, which can achieve higher detection accuracy

The rest of the paper is organized as fallows: Section 2 discusses related work for Android malware detection using dynamic detection scheme. Section 3 introduces the overall structure of our framework. The method of processing dynamic API call sequence and the fusion model is explained in Section 4. Section 5 shows the experimental results to illustrate the performance of our framework. Section 6 concludes our research.

2. Related Work

Faced with such a huge amount of dynamic API call sequence, the existing research solutions can be classified into the two following types: using statistical features of dynamic API call sequence and truncating a fixed-length subsequence of a dynamic API call sequence:

(a) Using statistical features of dynamic API call sequence: use statistical algorithms to select the APIs that have a greater impact on malware detection from the dynamic API call sequence, such as the information gain algorithm [17] and the TF-IDF (term frequency-inverse document frequency) algorithm [18], and then use the selected API as feature for malware detection
(b) Truncating a fixed-length subsequence of dynamic API call sequence: intercept the first part of the entire dynamic API call sequence, and then use deep learning algorithms for malware detection [19]

2.1. Malware Detection Based on Statistical Features of API Call Sequence. Uppal. et al. [20] extracted 3-gram vectors from dynamic API call sequence and then used the odds ratio to select the most important vectors. Finally, they applied SVM model to complete malicious judgment. Mohammed K. Alzaylaee et al. [17] used the tool DynaLog to extract 178 behaviour features of applications execution process and then used the information gain algorithm to select the most important 120 from them. In order to increase the accuracy of the model, the authors extracted permission requested by the application and formed a 420-dimensional feature vector. Finally, the authors used DNN (deep neural network) to determine maliciousness. Fang et al. [21] constructed a TFIT (trace frequency information table) to save the number of API calls. After that, they calculated the weight of each API call according to TFIT and selected the most important API calls. Finally, they fed the selected API calls into XGBoost. Yong Qiao et al. [8] used a frequent itemset mining algorithm to find frequent API calls from the API call sequence and then used frequent item sets as features for malware clustering. Singh et al. [22] utilized Cuckoo Sandbox to capture the application's dynamic API call sequence and then checked whether a certain API appeared as feature. Kim J. et al. [23] counted the frequency of each API in dynamic API call sequence as feature. Afterward, the maliciousness of applications was judged by comparing the difference in API call counts.

Merely using statistical features will lose the order information of API call sequence. However, the order of API calls is very important; different orders can indicate different behaviours. For example, *getContentResolver().query()* $\longrightarrow$ *socket.getOutputStrean().write()* can complete the behaviour of stealing user information, but, if conversely, there is no such behaviour. Such statistical methods damage the accuracy of malicious detection. At the same time, due to the excessive number of APIs, there are too many feature dimensions, which can easily cause overfitting problem and reduce the generalization ability of malicious detection.

2.2. Malware Detection Based on a Fixed-Length Subsequence of Dynamic API Call Sequence. Xi Xiao et al. [9] extracted the system call sequence during application execution and then applied the LSTM model to extract the information in the sequence. In order to adapt the length of the sequence to the LSTM (Long Short-Term Memory) model, the authors truncateed the sequence, limiting the length of the sequence to 500. Wenqi Xie et al. [10] proposed an algorithm to segment the system call sequence. They first cut the system call sequence into fixed-length subsequences and then labeled each subsequence. Finally, the LSTM model is used to determine the maliciousness of each subsequence. Zhaoqi Zhang et al. [11] proposed a method to map an API and its parameters to a fixed-length vector. The LSTM model is used to determine its maliciousness based on the vector. Kolosnjaji et al. [12] proposed a scheme to integrate CNN and LSTM models. The method first uses CNN (convolutional neural network) to learn a set of features and then uses LSTM to determine the maliciousness. Pascanu et al. [13] proposed a phased detection model, including feature extraction stage and malicious detection stage. In the feature extraction phase, they used the RNN (recurrent neural network) to predict the next API call based on the previous sequence of API calls. In the classification stage, RNN is frozen, and all API outputs are converted into feature vectors by maximum pooling for classification.

The LSTM model can learn the sequential relationship of API calls. However, the best effect length of the LSTM model is 200, so, in order to make API sequences suitable for the LSTM model, many researchers intercept a part of API call sequences. The intercepted API call sequence cannot fully represent the behaviour of the application at runtime, and the malicious behaviour may be cut off, which greatly reduces the detection accuracy.

Therefore, we propose a method to convert API call sequence into a function call graph, which not only retains most of the application execution order information but also significantly reduces sequence size. We will introduce our schema in the next part.

Table 1 compares the existing state-of-the-art schemas with the proposed work. The schemas which use statistical features of API call sequence will lose the order information of API call sequence. The schemas which truncate a fixed-length subsequence of dynamic API call sequence will cause huge computational overheads. Therefore, we propose a method to convert API call sequence into a function call graph, which not only preserves the complete sequence but also preserves the order information of the sequence. We will introduce our schema in the next part.

3. System Framework

The overview of the FGL_Droid system is shown in Figure 1, which is divided into four parts: AndroidGuard [24], DyAPICapture, Graph Construct, and FGL_Droid model.

3.1. Static Feature Extraction. In the static feature extraction stage, we use AndroGuard [24] to extract the static features of each application and select the permission request list as our static features. The reasons mainly include the two following aspects:

(a) The distribution of Permissions in normal applications and malicious applications is significantly different, which can reflect the malicious behaviour of applications to a certain extent.

(b) For benchmark dataset, we collect a total of 135 kinds of Permissions. Using Permissions as static features will not suffer from high-level dimensions.

3.2. Dynamic Feature Extraction. We have designed a dynamic behaviour capture system DyAPICapture to capture the API calls during the running of the application. As shown in Figure 2, DyAPICapture mainly includes two modules: dynamic test module and API capture module. In the dynamic test module, we adapt the model-based stateful dynamic triggering scheme APE [16], which can achieve higher function coverage with less detection actions. In the API capture module, we use the functions provided by JDWP (Java Debug Wire Protocol) [25] to obtain the API calls during the execution of applications. At the same time, in order to achieve higher functional coverage, the entire test process will last for 5 minutes, and the captured dynamic API sequence length has reached millions level, with an average length of 1.698 million. Moreover, some malicious applications can detect the operating environment; if they are found to be running in a virtual machine, they will hide their malicious behaviour and avoid detection. Therefore, our dynamic capture system can be deployed on real devices.

3.3. Function Call Graph Construction. The collected API call sequence is too long. We propose algorithm 1, which converts API call sequence into a directed and edge-weighted function call graph. Function call graph contains

Table 1: Comparison of the proposed work with state-of-the-art android malware detection schemas.

Schema	Features	Classification algorithm	Computational overheads	Capacity of saving information
[1]	Statistical features of API call sequence	SVM	Medium	Low
[2]		DNN		
[3]		XGBoost		
[7]	A fixed-length subsequence of API call sequence	LSTM	High	Medium
[10]		CNN + LSTM		
[11]		RNN		
Ours	**Function call graph + permission**	**GCN + LR**	**Low**	**High**

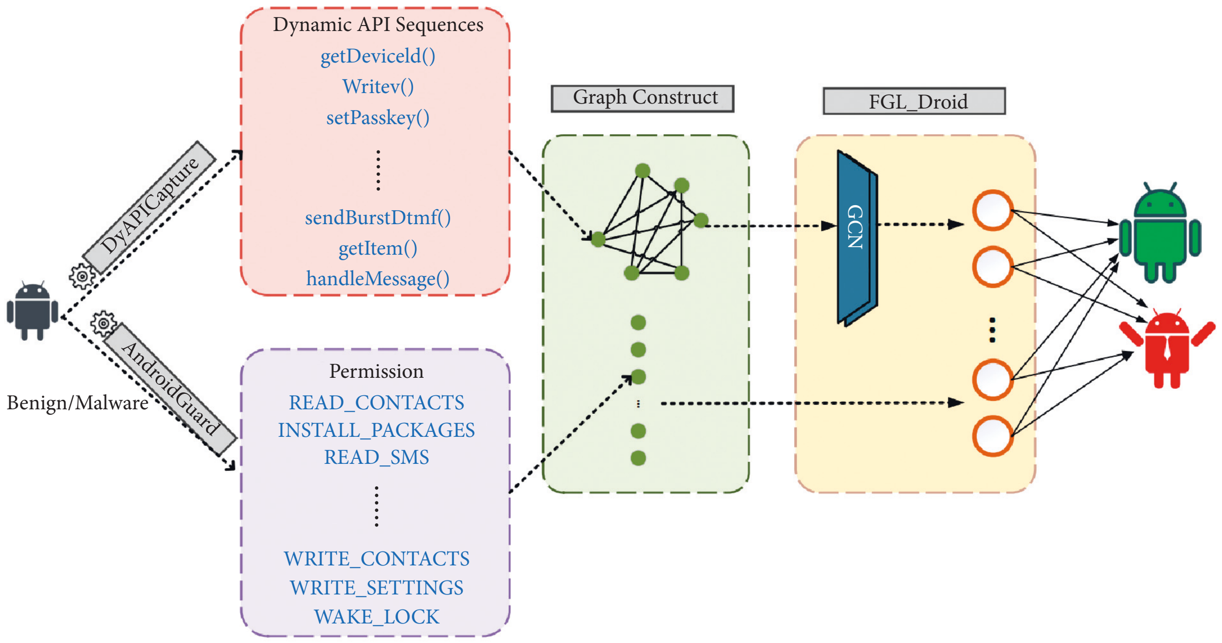

Figure 1: Framework of FGL_Droid. It includes four parts: DyAPICapture: extract the dynamic API call sequence; AndroidGuard: extract the Permission; Graph Construct: function call graph construction; and FGL_Droid: detection model.

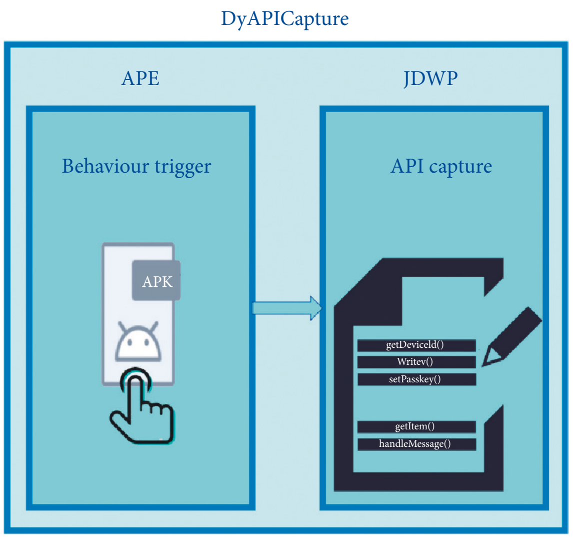

Figure 2: DyAPICapture consisting of two parts: behaviour trigger and API capturer.

26 vertices, each representing a function class in SUSI [26]. Each directed edge in function call graph represents a call relationship, and the corresponding weight means the number of call relationship. Therefore, the function call graph can keep order information of the API call sequence.

3.4. Model Training. We design a fusion model FGL_Droid, which can effectively mine applications behaviour pattern from the function call graph and merge it with the Permission feature. Once the function call graph is constructed, we train our FGL_Droid model on this graph for malware detection. The details of the entire model are explained in the Methodology chapter.

4. Methodology

4.1. Construction of the Function Graph. We propose a method to transform the sequence of API calls into a function call graph, which can save the order information of API calls while reducing data size of dynamic API call sequence. The method applies Algorithm 1 to construct a graph $G(V, E, W)$ by representing a unique function class at vertex V and a call relationship by a weighted edge E. As shown in Figure 3, Algorithm 1 mainly contains four steps: API substitution, deredundancy, graph construct, and normalization.

(1) API substitution: the algorithm first transfers the original dynamic API call sequence into function class sequence. Each API is replaced with the corresponding function class in SUSI [21]. As shown in Figure 3, the API call sequence [*'getDeviceId()', 'getPhoneNumber()', 'getCellLocation()', 'getSerialNumber()', 'query()', 'sendMessage()', 'sendSms()', . . ., 'getRawContactId()', 'getContactUri()', 'getAllContractName()', 'getViewAt()', 'getUserPassword()', 'getDriverName'*] will be converted into function call sequence [*'UNIQUE_IDENTIFIER', 'UNIQUE_IDENTIFIER', 'LOCATION_INFORMATION', 'NETWORK_INFORMATION', 'FILE_INFORMATION', 'ACCOUNT_INFORMATION', 'e-mail', 'SMS_MMS', . . ., 'CONTACT_INFORMATION', 'CONTACT_INFORMATION', 'CONTACT_INFORMATION', 'BROWSER_INFORMATION', 'SYSTEM_SETTINGS', 'NFC', 'NFC'*].

(2) Deredundancy: in order to reduce the length of function call sequence, we only retain one of the same function classes that appears continuously. The function classes call sequence will become [*'UNIQUE_IDENTIFIER', 'LOCATION_INFORMATION', 'NETWORK_INFORM-ATION', 'FILE_INFORMATION', 'ACCOUNT_INFORMATION', 'e-mail', 'SMS_MMS', . . ., 'CONTACT_INFORMATION', 'BROWSER_INFORMATION', 'SYSTEM_SETTINGS', 'NFC'*].

(3) Graph construction: Algorithm 1 converts the execution sequence of function classes to function call graph. For the two consecutive function classes [*'UNIQUE_IDENTIFIER', 'LOCATION_INFORMATION'*] in the sequence, we create a directed edge pointing from *UNIQUE_IDEN-TIFIER* to *LOCATION_INFORMATION.* If this edge does not exist in the graph, add this edge to the function call graph; if this edge exists in the graph, increase the weight of the edge by 1. We get the function call graph as shown in Figure 2 through this step.

(4) Edge normalization: in order to reduce the impact caused by different orders of magnitude of each edge, the min_max *normalization method* was adopted to map all the weights into a range between (0, 1). As shown in Equation 1, $W_{(i,j)}$ represents normalized weight of the edge, $w_{(i,j)}$ represents the number of calls, MAX_{weight} represents the maximum weight of the edge in the figure, and $\text{MIN}_{\text{weight}}$ represents the minimum weight of the edge in the figure. Finally, we obtain the directed weighted function call graph F_GRAPH (V, E, W) originating from API sequence. Each node of F_GRAPH represents a function class. Each directed edge in F_GRAPH represents a call relationship, and the corresponding weight means the frequency of call relationship. Therefore, the function call graph can keep sequence information of the API call sequence.

$$W_{(i,j)} = \frac{w_{(i,j)} - \text{MIN}_{\text{weight}}}{\text{MAX}_{\text{weight}} - \text{MIN}_{\text{weight}}}. \quad (1)$$

4.2. Fusion Model of GCN and LR. Our model consists of three steps: extracting dynamic behaviour feature, merging dynamic behaviour feature and Permission, and malicious judgment.

4.2.1. Extracting Dynamic Behaviour Feature. For function call graph F_GRAPH (V, E, W) obtained by Algorithm 1, we utilize graph neural network to extract application's behaviour feature from function call graph. For each vertex in the graph, the graph neural network can fuse the information in adjacent nodes and edges and keep the call information between nodes. The information fusion mode is

$$H^{(l+1)} = \sigma\left(\widetilde{D}^{-1/2}\widetilde{A}\widetilde{D}^{-1/2}H^{l}W^{l}\right), \quad (2)$$

where A represents the sum of the adjacency matrix of the graph and the identity matrix, so that the information of its own nodes can be kept in the process of information fusion. D is the degree matrix of A, H is the characteristic of each layer, and σ is the activation function.

Through our observation, a malicious behaviour can be accomplished only by calling 2-3 function classes. For example, information leakage malware usually only needs to

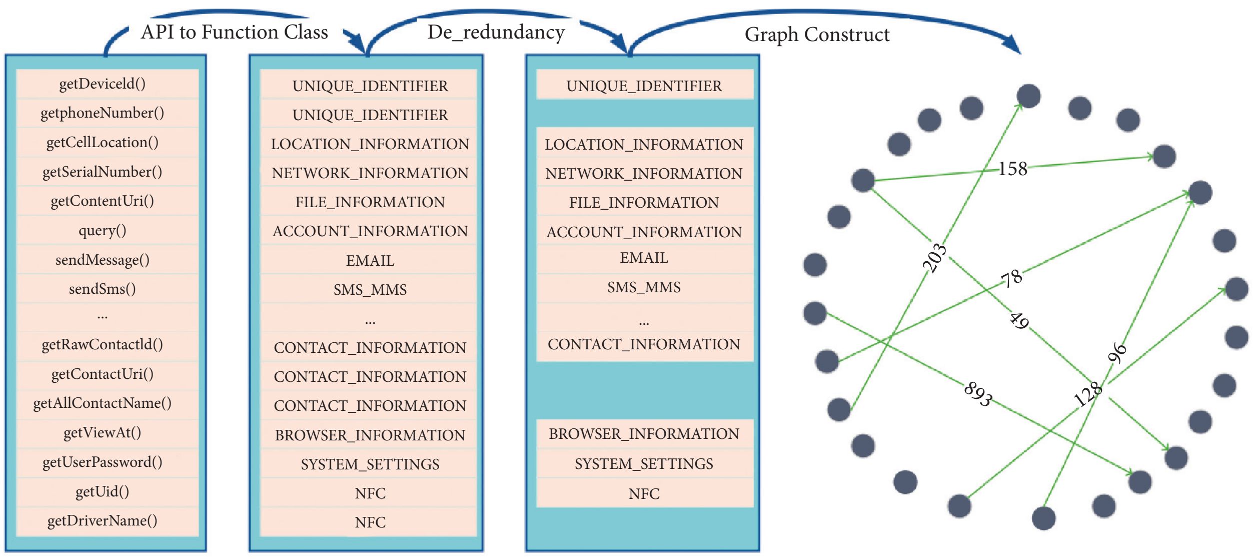

Figure 3: The process of converting dynamic API call sequence into function call graph. The figure only shows the first three steps.

```
(1) Input: Dynamic API Call Sequence(S) and SUSI's API Category($D$)
(2) Output: Function Call Graph F_Graph $(V, E, W)$
(3) Initially: let $V = (F1, F2, F3, \ldots, F26)$ be all the function category in $D$ and $E \leftarrow \varnothing$
(i) //Step 1: replace the API with corresponding function class
(4) for API $\in S$ do
(5)         Change API to its class in $D$
(6)         let $F(f1, f2, f3, \ldots f_n)$ be the Function call Sequence
(7) end for
(ii) //Step 2: delete adjacent duplicate function classes
(8) while $i <$ length($F$) do
(9)         if $F[i] = F[i-1]$ then
(10)                delete $F[i]$
(11)     end if
(12) end while
(iii) //Step 3: convert the execution sequence of function classes to function call graph
(13) while $i <$ length($F$) do
(14)         if $(F[i], F[i+1]) \notin E$ then
(15)                Add an edge $(F[i], F[i+1])$ to $E$
(16)             $W(F[i], F[i+1]) = 1$
(17)     else
(18)             $W(F[i], F[i+1]) += 1$
(19)     end if
(20) end while
(iv) //Step 4: normalization
(21) maxWeight $\leftarrow$ max$(W)$, minWeight $\leftarrow$ min $(W)$
(22) for $w \in W$:
(23)         $w \leftarrow (w -$ minWeight$)/($maxWeight $-$ minWeight$)$
(24) end for
(25) Return F_Graph $(V, E, W)$
```

Algorithm 1: Graph construction algorithm. Our algorithm includes four steps, which are (1) replace the API with corresponding function class, (2) delete adjacent duplicate function classes, (3) convert the execution sequence of function classes to function call graph, and (4) use max-min algorithm method for normalization.

obtain account information (*ACCOUNT_INFORMATION*) and then send it to the server through network (*NETWORK*); malicious download malware first needs to obtain the memory status of the phone (*UNIQUE_IDENTIFIER*) and then attach the information to the designated server (*NETWORK*) and finally download the malicious application to the designated memory through the network and decompress and install it (*FILE*). Therefore, in our GCN model, we use a double-layer GCN network. Each node merges the information of the nodes that are within two hops from the node, which can completely cover all the malicious behaviours.

4.2.2. Merging Dynamic Behaviour Feature. In order to reduce the amount of data and avoid the overfitting phenomenon caused by excessive feature dimension, we replaced API with the corresponding SUSI's function class to reduce the data dimension. Such coarse-grained method will inevitably lose some detailed information, resulting in a decrease in detection accuracy. We find Permission can be used to make up for the lost detail information. For example, sendTextMessage() and notifySendFailed() are assigned to the same functional class, losing the differences in their behaviours. However, sendTextMessage() requires Permission "SEND_SMS," but notifySendFailed() does not, so we can make up for lost details by introducing Permission. Due to the fact that the Permission information is an advanced feature, there is no need to apply complex deep neural network on this feature. In order to efficiently utilize both function call graph and Permission, a fusion model of GCN and LR is proposed, as shown in Figure 4. For complex graph data, we use double-layer GCN to extract the application behaviour information from the graph. Then apply Average_pooling and Max_pooling operations on each node and concatenate the two results together as the final graph representation Vector_Graph with length of 1150. We combine Vector_Graph with Permission features as Vector_mix.

4.2.3. Malicious Judgment. After getting the Vector_mix, we use a dense layer with 128 neural units to reduce the dimension of the intermediate vector to 128. A ReLU activation is adopted in this dense layer. Then we use a dropout layer with a rate of 0.6 to reduce overfitting. Finally, we use a full connection layer with an output dimension of 1 to obtain the malware probabilities. The binary cross-entropy function is adopted as our loss function, which can characterize the difference between the true sample label and the predicted probability. The loss function is

$$\text{Loss} = y \log\hat{y} + (1 - y)\log(1 - \hat{y}), \tag{3}$$

where y represents the true label and $\hat{y}$ represents the predicted label.

In addition, the optimization method we take is Adam, and the learning rate is 0.0005.

5. Results and Discussion

5.1. Datasets and Experimental Environment. The benchmark dataset used in this paper is shown in Table 2, including 4217 normal applications downloaded from the Google Play Store and 3950 malicious applications collected from Andro_dumpsys Project, which includes 13 malware families.

The configuration of the experimental equipment is shown in Table 3.

5.2. Effectiveness of Translating Dynamic APIs into Graph. To evaluate the effectiveness of translating dynamic APIs into function call graph, we measured the accuracies of the other two mainstream process methods: using the statistical features of dynamic API call sequences and intercepting a fix_length subsequence of the dynamic API call sequence:

(a) SF: using the statistical features of dynamic API call sequence. We count the number of occurrences of an API in dynamic API call sequence and then use DNN (deep neural network) to detect the maliciousness of the application. What we use is a three-layer DNN with 4500, 500, and 1 neural unit, respectively, and the activation function is ReLU. During the training process, we choose the learning rate of 0.003, the optimizer is the Adam optimizer, and the loss function is the cross-entropy loss function.

(b) FLS: intercepting a fix_length subsequence of the dynamic API call sequence. We use the first 1000 APIs of dynamic API call sequences and then use LSTM (Long Short-Term Memory) network to detect malware. What we use is a single-layer LSTM network, and the training process is consistent with the above parameters.

The results are shown in Table 4. Significantly, our process method of translating dynamic APIs into function call graph is superior to the other two processing methods in four evaluation indicators. As for the metric MTTD (mean time to detect), since we transformed the original sequence into a graph with only 26 vectors, our schema also outperforms the other two.

In the process of the experiment, we find that there is obvious overfitting phenomenon in schemes which use statistical features of dynamic API call sequences. On the contrary, our method does not have this problem. We record the change of loss value in the training process of using statistical features and Ours. As shown in Figure 5, we can clearly see that loss has been decreasing on the training set, but, on the test set, there has been a large increase after a small decrease. Obviously, there is an overfitting phenomenon when we use the statistical features of dynamic API call sequences to judge malicious.

Using the frequency of each API that appears in dynamic API call sequence as features will cause the data dimension to be too high. Google officially provides 25834 APIs, so the dimension of the feature is 25834. In some papers, they not only use API but also include Permissions and opcodes. The final feature dimension can reach tens of thousands of dimensions. The feature dimension is too high, so it is easy to suffer from overfitting. Our processing method converts the API call sequence into a graph with only 26 nodes, thus reducing the data dimension and avoiding the occurrence of overfitting.

5.3. Effect of the Layer of GCN. In this part, we verify the influence of different layers of GCN on the detection result. We conduct experiment under different layers of GCN from 1 to 5. The results are shown in Table 5. The best detection

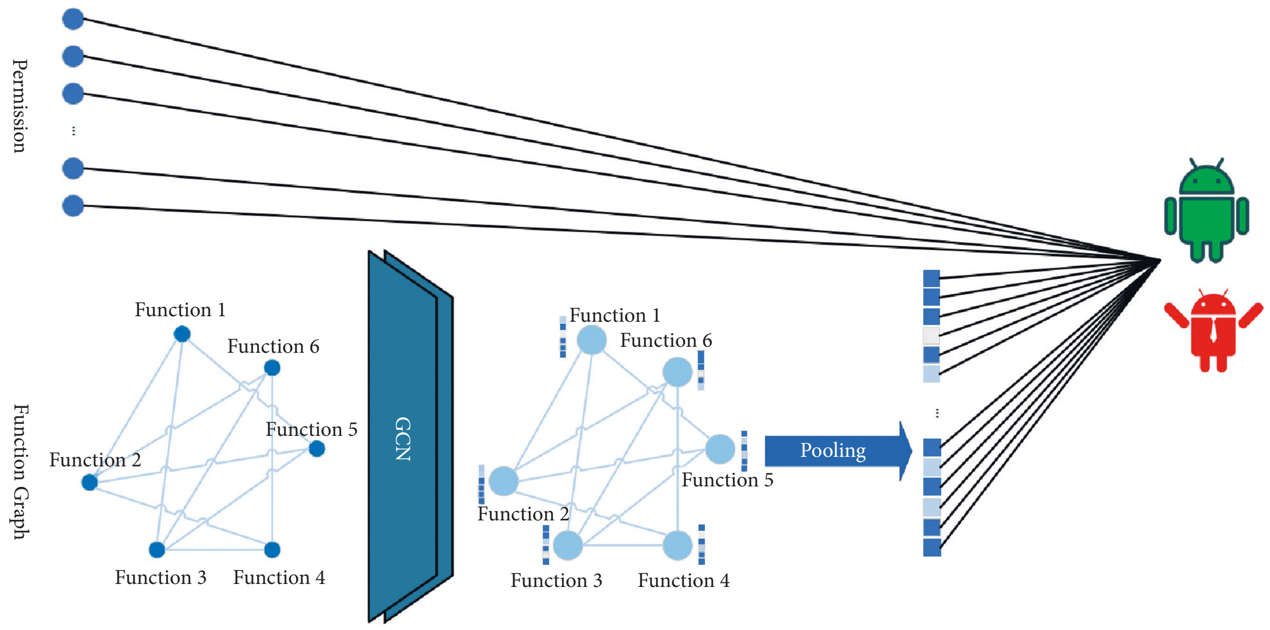

Figure 4: Fusion model of GCN and LR. Use GCN to extract the application behaviour pattern from the function call graph and then splice it with the Permission feature, and then use the LR model for malware detection.

Table 2: Summary of the dataset.

Dataset	Number	Source
Malware	3950	Andro_dumpsys
Benign	4219	Google Play Store

Table 3: Experiment environment.

Environment	Configuration
Operating system	Android 6.0.1
Phone model	LG Nexus 5

performance is achieved when there are 2 layers. For two-layer GCN, each node merges the information of the nodes that are within two hops from the node, which can completely cover all the malicious behaviours. For example, information leakage malware usually only needs to obtain account information (ACCOUNT_INFORMATION) and then send it to the server through network (NETWOR). When the layer of GCN is greater than 2, the representation of each node will tend to homogenize, which decreases the accuracy of detection.

5.4. Effectiveness of Normalizing the Edges of Function Call Graph. Android malware usually hides malicious code in normal code, and most of the dynamic API call sequences we extracted are normal call relationships. Therefore, the weight of the normal call relationship in the function call graph obtained from the dynamic API call sequence transformation is much larger than the malicious call relationship, which has a greater impact on the final judgment result. In order to reduce misjudgment caused by the large difference of edge weights, we normalized the edges.

To show the effect of normalizing the edges of the function call graph, we added experiments to verify the impact of edge normalization on the proposed scheme. The WW is a weighted graph without normalization, and WNW is a weighted graph with normalization. As shown in Table 6, the detection accuracy of WNW is significantly superior than the other method, reaching 0.975.

5.5. Effectiveness of the Integration Model. Due to the fact that the method of transforming dynamic API call sequences into function call graph loses some details, we propose a way to make up for the loss of details by merging the function call graph and Permission. As mentioned in the previous section, both *sendTextMessage()* and *notifySendFailed()* are assigned to the same functional class "*SMS_MMS*," losing the differences in their behaviours. However, *sendTextMessage()* requires Permission "*SEND_SMS*," but *notifySendFailed()* does not, so we can make up for lost details by introducing Permission. In order to evaluate the effectiveness of fusion features, we conduct the three following groups of experiments:

(a) Permission: use Permission only for maliciousness determination
(b) Function call graph: use only the function call graph for maliciousness determination
(c) Fusion feature: use fusion information to determine maliciousness

Table 7 shows the experimental results. It can be seen from the experimental results that our scheme is significantly better than only using Permission or function call graph, which indicates that our fusion scheme is indeed effective in the determination of maliciousness.

Table 4: Effectiveness of translating dynamic APIs into graph.

Process method	Approach	ACC	Precision	Recall	F1	MTTD (ms)
SF	DNN	0.925	0.933	0.933	0.933	0.208
FLS	LSTM	0.702	0.952	0.662	0.781	1.406
FCG + Permission	**GCN + LR**	**0.975**	**0.979**	**0.976**	**0.978**	**0.141**

SF: statistical features of dynamic API call sequence; FLS: fix_length subsequence of the dynamic API call sequence; FCG: function call graph.

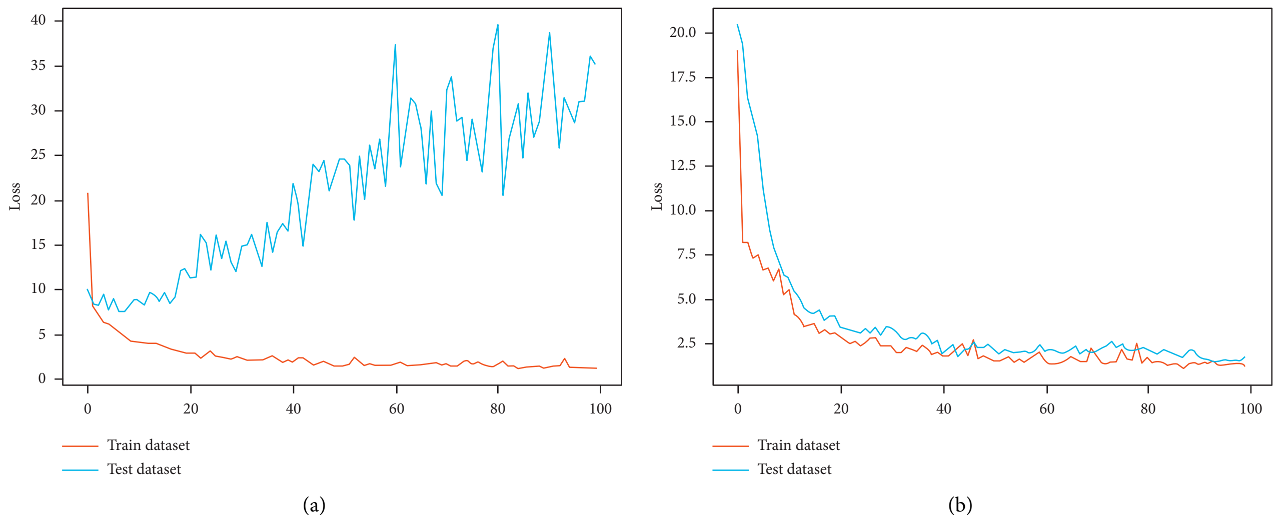

Figure 5: The change of loss value during model training and testing using different API call sequence processing methods. (a) Using statistical features of API call sequence. (b) Ours: using function call graph.

Table 5: Results under different numbers of GCN layers.

Layer	ACC	Precision	Recall	F1
1	0.957	0.963	0.940	0.951
2	**0.975**	**0.979**	**0.976**	**0.978**
3	0.968	0.977	0.966	0.972
4	0.954	0.969	0.954	0.961
5	0.953	0.962	0.934	0.948

Table 6: The performance comparison with other proposed systems.

Method	ACC	Precision	Recall	F1
FCG_WW	0.713	0.653	0.799	0.719
FCG_WNW	**0.975**	**0.979**	**0.976**	**0.978**

WW: function call graph with weights; WNW: function call graph with normalization weights.

Table 7: Evaluating the effectiveness of the integration model.

Feature	ACC	Precision	Recall	F1
Permission	0.748	0.665	0.999	0.799
Function call graph	0.920	0.904	0.906	0.930
Fusion feature	**0.975**	**0.979**	**0.976**	**0.978**

5.6. Model Evaluation. In order to study the performance improvement of FGL_Droid model, we compare the state-of-the-art scheme of malicious detection using dynamic API call sequences, which mainly includes two ways: using statistical features of API call sequences or intercepting the API call sequence to a certain length.

Table 8 shows the results of the comparative experiment. Our scheme is significantly better than other methods. The precision of FGL_Droid in the test set is 0.975, and the F-Measure is 0.978.

For the method of using the statistical features of the dynamic API call sequence [5, 17, 22], they lose the sequence

Table 8: The performance comparison with other proposed systems.

Approach	ACC	Precision	Recall	F1
Mohammed K. Alzaylaee [17]	0.950	0.941	0.978	0.959
Jagsir Singh [22]	0.855	0.769	0.949	0.849
Zhenlong Yuan [5]	0.966	0.966	0.966	0.966
Xi Xiao [9]	0.923	0.939	0.905	0.922
Bojan Kolosnjaji [12]	0.894	0.856	0.894	—
Rakshit Agrawal [6]	0.954	0.963	0.951	—
Ours	**0.975**	**0.979**	**0.976**	**0.978**

information of the API call sequence. For the method of intercepting a part of dynamic API call sequence [6, 9, 12] may lose malicious behaviour, our method not only preserves the complete sequence but also preserves the order of the sequence. Therefore, our method obtains better detection accuracy compared to other methods.

6. Conclusion

In this paper, we propose a new framework for detecting Android malicious applications. First, we transform the multimillion-length dynamic API call sequence into a directed and edge-weighted function call graph with only 26 nodes, which can greatly reduce the amount of data, while preserving the order information of dynamic API call sequence. Then we use a double-layer GCN network to extract the behaviour information of the application from the heterogeneous function call graph and concatenate the results with the advanced features Permission. Finally, we feed the results into an LR model for malicious detection. The experimental results show that the proposed method of transforming API call sequence into a graph can significantly improve the time efficiency of detection, while the loss of precision is very small. We combined the two types of information through the fusion model, which was significantly better than other baseline models.

Data Availability

The data used to support the findings of this study are included within the article. The data presented in this study will be available upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant nos. 61672543 and 61772559.

References

[1] C. Secutiry, "Cyber secutiry report," 2021, https://www.ntsc.org/assets/pdfs/cyber-security-report-2020.pdf%20.

[2] Kaspersky, "Mobile malware evolution 2020," 2021, https://securelist.com/mobile-malware-evolution-2020/101029/.

[3] A. Arora, S. K. Peddoju, and M. Conti, "Permpair: android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2019.

[4] T. G. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, pp. 773–788, 2018.

[5] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

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

[7] M. Schofield, G. Alicioglu, R. Binaco et al., "Convolutional neural network for malware classification based on API call sequence," in *Proceedings of the 2021 the 14th International Conference on Network Security & Applications*, Computer Science & Information Technology (CS & IT), Zurich, Switzerland, January 2021.

[8] Y. Qiao, Y. Yang, L. Ji, and J. He, "Analyzing malware by abstracting the frequent itemsets in API call sequences," in *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, Melbourne, VIC, Australia, July 2013.

[9] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.

[10] W. Xie, S. Xu, S. Zou, and J. Xi, "A system-call behavior language system for malware detection using a sensitivity-based LSTM model," in *Proceedings of the 2020 3rd International Conference on Computer Science and Software Engineering*, Beijing, China, May 2020.

[11] Z. Zhang, P. Qi, and W. Wang, "Dynamic malware analysis with feature engineering and feature learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, NY, USA, February 2020.

[12] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, Springer, Hobart, TAS, Australia, December 2016.

[13] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *Proceedings of the 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, South Brisbane, QLD, Australia, April 2015.

[14] W. Wang, J. Wei, S. Zhang, and X. Luo, "LSCDroid: malware detection based on local sensitive API invocation sequences," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 174–187, 2019.

[15] Y. Qin, W. Wang, S. Zhang, and K. Chen, "An exploit kits detection approach based on HTTP message graph," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3387–3400, 2021.

[16] T. Gu, C. Sun, X. Ma et al., "Practical GUI testing of Android applications via model abstraction and refinement," in *Proceedings of the IEEE/ACM 41st International Conference on*

Software Engineering (ICSE), IEEE, Montreal, QC, Canada, May 2019.

[17] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, Article ID 101663, 2020.

[18] M. Ali, S. Shiaeles, G. Bendiab, and B. Ghita, "MALGRA: machine learning and N-gram malware feature extraction and detection system," *Electronics*, vol. 9, no. 11, p. 1777, 2020.

[19] S. Jha, D. Prashar, H. V. Long, and D. Taniar, "Recurrent neural network for detecting malware," *Computers & Security*, vol. 99, Article ID 102037, 2020.

[20] D. Uppal, R. Sinha, V. Mehra, and V. Jain, "Malware detection and classification based on extraction of API sequences," in *Proceedings of the 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, IEEE, Delhi, India, September 2014.

[21] Y. Fang, B. Yu, Y. Tang et al., "A new malware classification approach based on malware dynamic analysis," *Australasian Conference on Information Security and Privacy*, Springer, Cham, 2017.

[22] J. Singh and J. Singh, "Assessment of supervised machine learning algorithms using dynamic API calls for malware detection," *International Journal of Computers and Applications*, vol. 44, no. 3, pp. 270–277, 2022.

[23] J. Kim, S. Lee, J. M. Youn, and H. Choi, "A study of simple classification of malware based on the dynamic api call counts," in *Proceedings of the Advances in Computer Science and Ubiquitous Computing*, pp. 944–949, Bangkok, Thailand, December 2016.

[24] androguard, "reverse engineering, malware and goodware analysis of android applications," 2020, https://github.com/androguard/androguard/.

[25] oracle. Java, "Debug wire protocol," 2020, https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwpspec.html.

[26] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks," Rep. TUDCS-2013-0114, University of Darmstadt, Darmstadt, Germany, 2013.