IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 27, 2020, accepted September 13, 2020, date of publication September 23, 2020, date of current version October 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3026052

Malicious Code Detection Based on Code Semantic Features

YU ZHANG AND BINGLONG LI
College of Cryptographic Engineering, Information Engineering University, Zhengzhou 450000, China
Corresponding author: Binglong Li (lbl2017@163.com)

ABSTRACT With the development of smart phones, malicious applications for the Android platform have increased dramatically. The existing Android malicious code analysis methods majorly focus on detection based on signatures, inter-component communication, and other configuration information features. Such methods ignore the effect of the semantic features of the malicious code. Even a few such studies that exist are based on the statistical features of the code for malicious code detection. To address these shortcomings, we (1) use the code semantic structure features to reflect deep semantic information, (2) propose a preprocessing method of APK files to generate graphics that reflect the code semantic features, and (3) introduce the advanced graphical semantics for a graph convolutional network (GCN) model to automatically identify and learn semantics and extract features for malicious code detection. Experiments on a dataset confirm that the proposed method can achieve 95.8% detection accuracy. Compared with the existing methods that adopt configuration information features or statistical features of codes, our method shows higher accuracy.

INDEX TERMS Deep learning, malicious code detection, semantic features.

I. INTRODUCTION

At present, the Android operating system has firmly occupied more than half of the market and is one of the most popular smart mobile platforms. However, the Android operating system has considerable hidden security risks. Although Android's open ecological environment provides convenience for application programming, it increases the number of vulnerabilities that malware can exploit. In recent years, the installation and use of malware has posed a security threat to mobile devices, including theft of privacy of smartphone users and repackaging benign applications.

The development of malware has prompted researchers to develop malware detection techniques. The current malware detection technology is mainly divided into static analysis and dynamic analysis. Dynamic analysis tracks the behavior of the Android application while it is running, whereas static analysis analyzes the Android application package (APK) file of the application. In this study, we focus on static analysis techniques.

Many malware detection systems use static analysis techniques to detect malware. They mainly focus on the static features of malware, such as permissions required by the APK, services [1], sensitive application programming interface (API) [2], and program functions [3]. However, these features are only the information extracted from the configuration file, ignoring the dex source code of the APK. These malware detection methods ignore the semantic features of APK files, thereby reducing the recognition rate of malicious code detection. Some studies have reported on this problem. Pei *et al.* [4] focuses on the semantic information of dalvik opcodes, mainly collecting statistical features of a malware code, such as code length and the number of occurrences of special characters. Yen and Sun [5] converts the classes.dex file in the APK into images to extract semantic features, where the image is composed of multiple color blocks. The size of each color block is determined by the corresponding characters in the code; that is, the greater the number of characters appearing in the code, the larger is the color block of the corresponding image. These studies extract the statistical features of the code, thereby improving their effect compared to other features [4]. However, by considering only the statistical features of the code, the potential semantic information such as the structural features of the code is overlooked. In addition, these methods are not valid when malware developers use different special characters. To address this issue, we propose a scheme for extracting structural features from the source code in order to provide deep code

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

semantic information for subsequent model learning. More specifically, we transform the code into a data flow graph that reflects semantic information through preprocessing and then use the data flow graph as an input to the subsequent model.

In recent years, the field of deep learning algorithms such as recurrent neural networks and convolutional neural networks (CNNs) has been combined with malware detection technologies. Among them, CNNs are widely used in the field of malware detection. Ren *et al.* [6] converts APK binary files into grayscale images and then uses CNNs to extract features. McLaughlin *et al.* [7] use recurrent neural network research to classify the features of the opcode sequence extracted from APK. However, the objects of the studies are Euclidean space data and structural rules irrespective of whether the search object is an opcode sequence or a grayscale image converted from a binary file. In this study, the structure around the nodes of the extracted data flow graph is unique, which renders the traditional CNN and RNN instantaneously invalid. To meet this challenge, we propose a model based on graph convolutional networks (GCNs) [8] to detect Android malware. We use the GCN to assemble each node with a fixed number of ordered neighbors and summarize the features of all neighboring nodes of each node.

Specifically, in this article, we propose a deep learning framework that can extract semantic features of the code, which are then used in malware code detection with improved accuracy. We first perform data flow analysis to extract the data flow graph of the Android application, and then add data flow attributes to the nodes of this graph. The structural information and attribute information of the node collected by the GCN model are used as the input of the subsequent classifier.

In general, our study contributes in the following aspects:

1) The feature extraction of the code by traditional deep learning is improved by introducing the semantic features of the code for malicious code detection.

2) A preprocessing method for APK files is proposed to generate a data flow graph that reflects semantic information.

3) A deep learning framework based on the GCN is proposed for malicious code detection. Experimental results show (Chapter 4) that compared with existing methods, our proposed features can improve the detection accuracy of malicious code.

Chapter 2 introduces related work, Chapter 3 introduces the pre-processing methods and deep learning framework used for malicious code detection, Chapter 4 presents the experimental results, and Chapter 5 finally discusses the conclusions.

II. RELATED WORK

Compared with traditional malware, mobile platform malware has special features. For example, it does not require large-scale self-replication and distribution like viruses or worms, but only needs to be placed in an application store to be downloaded on a large scale. In addition, the signature of mobile phone malware is relatively concealed, and the signature codes of most malware are not the same; moreover, a new malware cannot be detected promptly, so a detection technology based on signature matching is not suitable for detecting malicious codes in Android phones. The well-known open source tool Androguard [9], implemented in academia in 2013, is a representative method based on developer signature matching. By establishing a large and effective database, we can quickly and effectively discover known malicious applications but not new ones. Another method is behavior-based detection technology in which the mobile phone malware behavior is accurately detected by monitoring the behavior of an Android code (for example, through dynamic interception or static analysis to obtain the system call sequence of the program) by comparing it with known malicious behavior patterns (malware system call sequence) [10]. The most prominent advantage of behavior-based detection technology is that its feature library is small and does not need to be updated frequently. It can also detect unknown malware with similar behavior patterns. However, it requires high real-time performance, and it must be ensured that malicious programs are detected before they damage the system. Threats are relatively expensive, and the usual solution is to use sandboxes to close operations and monitor programs.

These shortcomings of the traditional methods have motivated research into new technologies. In recent years, data mining based on machine learning is the research hotspot of the development of Android malicious code detection technology [11]. Machine learning algorithms mainly include the application of common methods such as k-nearest neighbors, support vector machines, neural networks, Bayesian classification, and clustering; feature processing modes mainly include source code feature extraction and mobile application configuration file analysis, API call analysis, application permission monitoring, abnormal behavior analysis, and other multi-level and multi-dimensional methods [12]–[14]. However, machine learning is usually based on feature engineering and relies on complex or expert features to complete learning tasks. Tobiyama *et al.* [15] proposed malware process detection based on the process behavior. In this method, long-term short-term memory (LSTM) was used for feature extraction and the CNN was used for classification. The process behavior is a series of API calls. Features are extracted from the process behavior log file and transferred to an image that contains local features, which mainly represent process activities. Therefore, the CNN can be applied to capture these local features and correctly classify the images. Rhode *et al.* [16] studied the possibility of detecting malware executable files on the basis of behavioral data. In their study, the selected features are 10 consecutive machine activity data indicators instead of classification API calls. However, since API calls can be manipulated, the classification of input samples may be incorrect. Ilham *et al.* [1] and Yuan *et al.* [17] proposed a deep learning-based extraction method of three types of features from the malware: (1) required permissions, (2) sensitive APIs, and (3) dynamic behavior. These features

were then used for complex pattern and feature recognition. Ren *et al.* [6] used an end-to-end architecture to build deep learning models and characterize Android applications. Kolosnjaji *et al.* [18] proposed the classification of malware system call sequences using convolution and recurrent network layers, in which the convolutional layer is used for feature extraction.

Although these methods have higher detection accuracy compared to machine learning-based methods, they still show some weaknesses when expressing complex features. These methods cannot fully utilize the semantic information contained in the malware and lack the ability to integrate the semantic information into the neural network. Mercaldo and Santone [19] extracted features on the basis of the semantic information of the code, converted the binary file of the code into a grayscale image, and then used a CNN to extract features. Yen and Sun [5] used Simhash and Djb2 algorithms to convert APK dex files into images based on the statistical features of the code as well as used CNNs to extract features. Similarly, Azmoodeh *et al.* [20] proposed a deep feature space learning method to classify malicious and emerging Internet of Things (IoT) applications. They extracted 1078 benign and 128 malware opcode sequences from IoT applications. The selected features (opcode) of each sample were converted into a graph for classification using the deep convolutional network. These methods only used the statistical features of the code whereas ignoring the internal structural information, and they cannot fully integrate the semantic information into the neural network.

Therefore, we consider generating a data flow graph based on the structure of the code and extracting semantic feature information from it. However, the traditional deep convolutional neural network is not suitable for graphs with non-Euclidean structures. Some studies have tried to extract features from non-Euclidean structures. Shu *et al.* [21] used graph long short-term memory (G-LSTM) to extract features from the graph. G-LSTM includes a control unit to filter the graph node, select important information, and learn features. Tang *et al.* [22] used coherence constrained graph long short-term memory (CCG-LSTM) to extract features from the graph, thereby strengthening the associated routines and suppressing weak references associated with the graph to learn the graph features. The GCN model [8] extends the traditional Fourier transform to the graph; therefore, the convolution operation can be represented by the product of two Fourier transforms. Thus, the convolution of graphs with non-Euclidean structures is achieved.

III. METHOD

A. PREPROCESSING

1) DATA FLOW CHAIN ACQUISITION

As shown in Figure 1, the APK is essentially a compressed package, and multiple files and folders can be obtained from it after decompressing it. An APK file usually includes a META-INF folder (which stores program signatures and certificates), a res folder (which stores application resources), AndroidMainfest.xml (which stores application configuration information), and classes.dex (which are class files on the DEX compiler file format used to execute on the Dalvik virtual machine). In this study, the preprocessing of the APK is mainly aimed at classes.dex.

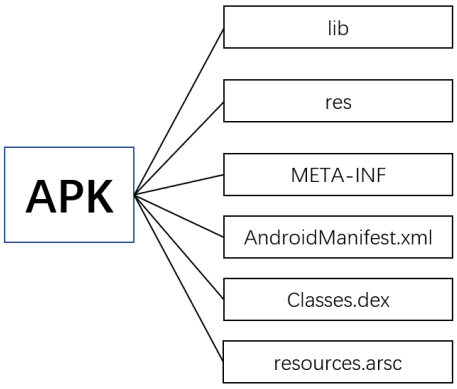

FIGURE 1. APK file structure.

For code preprocessing, we adopt the data flow analysis method, which is a commonly used method for describing software based on semantics. In this study, to divide the code into fragments carrying semantic information like biological genes, information is extracted in the form of a data flow chain. The data flow chain refers to the flow of data from the use to the definition in the software, including a data activity in the entire life cycle. The life cycle of a data is from the definition to the redefinition of the data or the end of the basic block. The basic block is a sequence of sequentially executed statements in the code segment. In general, there is only one entry and one exit, at which a statement jump occurs. In this study, the executable bytecode is analyzed to obtain the data flow chain. First, we use decompilation tools to extract code from classes.dex. Then, to convert executable bytecode into an intermediate language, we use the Soot tool [23], a commonly used JAVA program analysis tool, which can also now be used in Android software. Soot's analysis is based on several intermediate languages. In our experiment, we use Jimple language, a three-address statement, which is more convenient for analyzing and obtaining data flow chains. The data flow chain is obtained by Algorithm 1.

In Algorithm 1, the control flow graph of basic blocks generated by the Soot tool is re-analyzed to obtain a data flow chain. Through the analysis of the basic block control flow graph, we can extract the definition and usage values of all data (called def-value or use-value, respectively). Lines 2-18 extract data flow chains in the basic block through traversal. The data flow chain is generated when each piece of data is defined by a value, and the first node of the chain is the definition statement. Whenever a value is used, the statement used for this value will be added to the end of the data flow chain. When a value is redefined, the data flow chain marked by this value is output and another data flow chain marked

Algorithm 1

Input: Android apk document
Output: Data flow chain

```
for each control flow graph of basic block do
    for Each statement in a basic block do
        Get the def-value and use-value in the statement
        Use the method name and method type of this sentence as the expression of this statement
        if There is a data flow chain marked with this def-value then
            Output this data flow chain;
            Recreate a data flow chain marked with this def-value
            use the expression of this statement as the head of the chain;
        else
            Create a data flow chain marked with this def-value
            Use the expression of this statement as the head of the chain;
        end if
        for Each use-value in this statement do
            Find the data flow chain marked with this use-value;
            Add the expression of this statement to the end of the data flow chain;
        end for
    end for
    Output all data stream chains that are not currently ended in this basic block;
end for
```

by this value is opened simultaneously. In addition, when the basic block ends, all unfinished data streams generated at this time are generated. The chain will be terminated and output (i.e., line 18 in Algorithm 1).

To facilitate Android malware sample matching and comparison in subsequent research, the nodes of the generated data flow information graph must be embedded. We represent nodes by two features: the statement-type feature and the calling method name.

As an easy-to-analyze three-address statement, Jimple has only 15 statement types. In this study, we use a single letter to abstract the common statement types as shown in Table 1. In addition, the name of the calling method in the statement is obtained as an important feature of the code to express semantic information.

Thus, we use the statement type and calling method name to represent the nodes of the graph.Figure 2 is an example of two data flow chains. The code in Figure 2 is a piece of data flow chain in a malware sample widely named by the antivirus engine as the Plankton family. The malware of the Plankton family collects sensitive information such as bookmarks of infected device browser and phone numbers from various mobile devices and sends it to a remote server.

TABLE 1. Abstract representation of jimpele statement.

Statement type	Letter
Definition Statement	I
Assign Statement	A
Invoke Statement	V
Nop Statement	N
If Sentences	F
Goto Statement	G
Table Switch Statement	S
Lookup Switch Statement	L
Enter Monitor Statement	M
Exit Monitor Statement	W
Throw Statement	T
Return Statement	R

```
["I","<com.apperhand.common.dto.BaseBrowserItem:clone()>",
"V","< com.apperhand.common.dto.Bookmark:void<init>>",
"R""<null>"]
["A","<com.apperhand.common.dto.GetUserInfo:getDeviceId()>",
"V","< com.apperhand.common.dto.Bookmark: void<init>>",
"T""< com.apperhand.common.dto.Bgservice:Send()>"]
```

FIGURE 2. The example of two data flow chains.

2) DATA FLOW CHAIN MERGE

After all data flow chains are generated, the same nodes in the data flow chain are combined to form a graph that reflects the data flow information. Figure 3 is an example of the merging of two data flow chains shown in Figure 2. A graph of reaction data flow information thus generated is input to the following model.

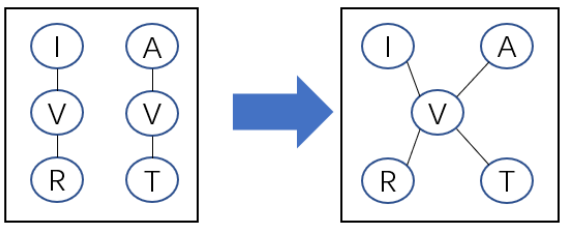

FIGURE 3. The merge of data flow chains.

3) EMBEDDED REPRESENTATION OF NODES

We use the statement type and calling method name to represent the nodes of the graph and embed the nodes into the vector space by applying an embedding method. Therefore, in the embedding of each node, the sentence type and calling method name from the vocabulary are mapped to the N-dimensional real number vector in the embedding vector space; $N = 200$ in this study. Figure 4 shows an example of encoding nodes.

Our goal is to obtain a matrix expression $x \in R^{M \times N}$,where instance x consists of N-dimensional embedded vectors $x_j, j = 1, \ldots, M$, corresponding to a series of nodes. M represents the number of nodes in the graph.

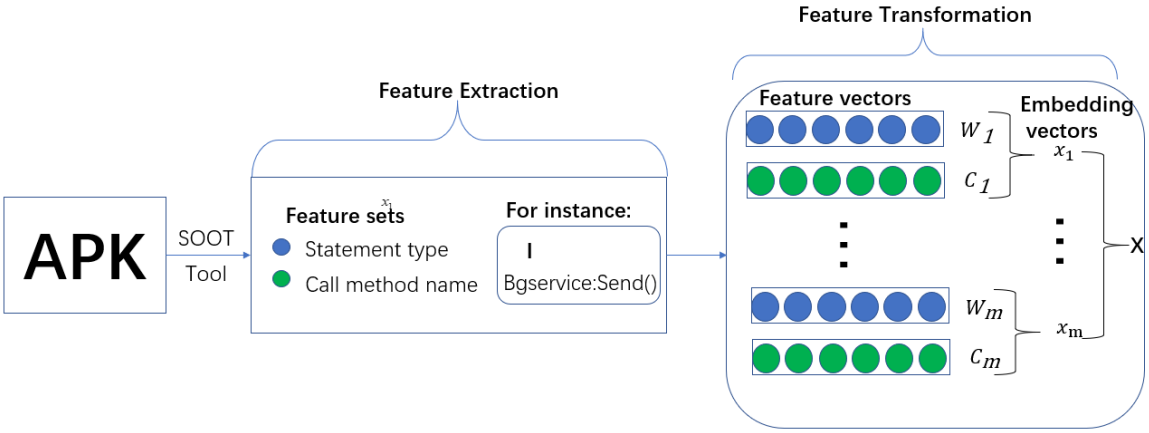

FIGURE 4. An example of encoding nodes.

x_j is generated is as follows:

$$x_j = \{emb(W_j \oplus C_j)\}_{j=1}^{M} \tag{1}$$

Here,W_j and C_j represent the type and name of the method called by node j, respectively; *emb* means the embedding operation; and $\oplus$ means the connection operation.

B. MALICIOUS CODE DETECTION BASED ON DEEP GRAPH CONVOLUTIONAL NETWORK

We use the GCN model to model the graph generated by the preprocessing. The GCN is a neural network that runs on the graph and summarizes node features based on the properties of its neighborhood. Depending on how many convolutional layers are used, the GCN can capture information about neighboring neighbors (with a graph convolutional layer) or any node with a maximum k-hop distance (where k represents the number of graph convolutional layers used).

The overall flow of the proposed algorithm is shown in Figure 5. The core of the network consists of a graph convolution layer and a classification layer. The graph convolution layer uses the GCN to aggregate nodes and extract features. In the classification layer, a softmax classifier is used to classify the input based on the features extracted by the graph convolution layer.

The graph is represented by (*VER*, *EDG*), which is a set of nodes on the graph $VER = \{VER_1, \ldots, VER_n\}$ and a set of edges $EDG \in VER \times VER$.

The GCN uses the following variables as input based on (*VER*, *EDG*):

1) A feature matrix composed of k-dimensional embedded vectors $x_j, j = 1, \ldots, M$ corresponding to a series of nodes (already described in the previous section).

2) The adjacency matrix A that reflects the graph structure can be obtained from the graph. Finally, the label probability Z is output through the classification layer, which is divided into two categories, benign and malicious.

The GCN obtains node vectors by iteratively aggregating vectors from its neighbor nodes. We try to obtain graph representation by learning to transform the entire graph into a vector space graph. In this space, the geometric relationship between the learning vectors reflects the graph structure information, which can be used as the input for the next classification layer.

Multi-layer changes are prevented by the large scale difference between the output and input. The adjacency matrix A is reconstructed by the normalization operation, described as follows:

$$\hat{A} = A + I \tag{2}$$

$$\tilde{A} = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}} \tag{3}$$

where I is the identity matrix used to add the self-loop connection. $\hat{A} = A + I$ ensures that when the adjacency matrix is multiplied, the node itself can also be added when the feature vectors of all adjacent nodes of each node are added; D is the diagonal of nodal degree matrix $\hat{A}$; $D^{-\frac{1}{2}}\hat{A}D^{-\frac{1}{2}}$ is the normalization operation performed to avoid the problem of data instability and gradient explosion or disappearance caused by repeated operations.

Each graph convolution layer of the GCN model has a nonlinear activation function, defined as follows:

$$x_v{}^{(k+1)} = ReLU\left(\sum_{u \in N(v)} \left(\tilde{A}x_v{}^{(k)}W^{(k)} + b^{(k)}\right)\right) \tag{4}$$

where k is the number of layers and $x_v{}^{(1)}$ is the embedded matrix X composed of the embedded vectors corresponding to the nodes in the graph, $W^{(k)}$ is the weight matrix of the kth layer, and $b^{(k)}$ is the intercept of the kth layer. Specifically, $N(v)$ is represented as a group of neighbors of node Ver, where Ver belongs to $N(v)$ due to self-loops [24]. *RELU* is the activation function and $\sum_{u \in N(v)} \left(\tilde{A}x_v{}^{(k)}W^{(k)} + b^{(k)}\right)$ represents the convolution operation adopted.

For our classification experiment, a two-layer GCN model is used, and the output of the graph convolution layer is input into the classifier. The objective function of the classification

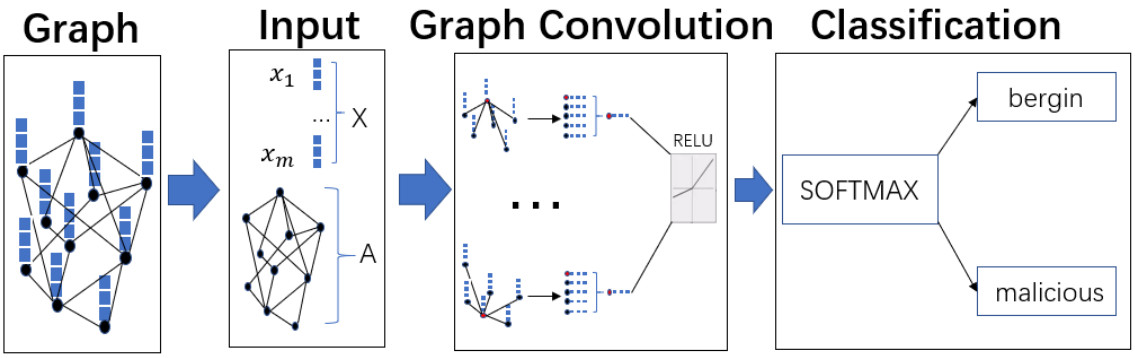

FIGURE 5. Architecture of our model.

is as follows:

$$Z = soft\max(x_v^{(2)}) \tag{5}$$

After Z provides the probability distribution of the label, cross-entropy is used as the loss function. Next, we use backpropagation loss and the Adam algorithm to update the weight matrix $W^{(k)}$ and intercept $b^{(k)}(k = 1, 2)$ in the convolutional layer of the graph.

The GCN basically collects information from neighbors and learn neighbor representations. The embedding matrix X and the adjacency matrix A constitute the GCN input. The graph structure framework can inductively learn the embedding of each node. The aggregation operation adopted can be described as follows:

$$x_v^{(k+1)} = ReLU\left(\sum_{u \in N(v)} \left(\tilde{A} x_v^{(k)} W^{(k)} + b^{(k)}\right)\right) \tag{6}$$

More formally, through learning, our model integrates the node structure (structure information) and node attributes (attribute information) on the last layer of the GCN, so that these two parts interact closely with each other.

IV. RESULTS

We conducted extensive comparison experiments, analysis, and research to demonstrate that the code semantic features are effective for improving malicious code detection. We used accuracy (ACC), accuracy (P), recall (R), and F-score (F) as parameters to quantitatively evaluate the performance of the classifier. In addition, we used five-fold cross-validation. In the malware detection experiment, we randomly retained four-fifths of the data for training and verified the model using the remaining one-fifth of the samples. All experimental results were obtained on the same evaluation dataset. Finally, the performance was optimized by adjusting variable hyperparameters.

A. DATASETS

The dataset consists of malware and benign applications, including 8,000 benign APKs collected from Google Play Store [25] and 8,000 malicious APKs collected from Virusshare [26]. The APK files we collected are between 20 KB and 50 MB.

B. EVALUATION METRICS AND EXPERIMENT SETTINGS

1) EVALUATION METRICS

To quantitatively evaluate the performance of the classifier, we used the following common machine learning performance evaluation indicators: True positive (TP) is the number of samples correctly classified as malicious, true negative (TN) is the number of samples correctly classified as benign, false positive (FP) is the number of samples misclassified as malicious, and false negative (FN) is the number of samples that are misclassified as benign.

Accuracy (Acc) represents the number of samples correctly classified as benign or malicious.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

Precision (p) is the ratio of correct positively labeled instances to all positively labeled instances.

$$Precision = \frac{TP}{TP + FP} \tag{8}$$

Recall (R) is the ratio of correct positively labeled instances to all instances that should have been labeled positive.

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

F-score (F) is the harmonic mean of precision and recall.

$$F - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{10}$$

2) EXPERIMENT SETTINGS

As shown in Figure 5, our model is divided into an input layer, a convolution layer and a classification layer. The convolution layer uses GCN model for graph convolution. To optimize the performance of our model, we study the hyperparameter settings of the GCN model in this section.

The setting of model parameters plays a vital role in the effectiveness of malware detection, such as the number of GCN layers, optimizers, and epochs. These factors are analyzed below.

TABLE 2. Experimental results of GCN layers comparisons.

Layers	Acc	P	R	F
G=1	98.15%	98.36%	98.48%	0.984
G=2	98.57%	98.82%	98.71%	0.987
G=3	98.28%	98.78%	98.23%	0.985
G=4	98.12%	97.99%	98.64%	0.983
G=5	98.02%	97.58%	98.05%	0.978

The hidden layer of the GCN plays a vital role in the performance of the model. As shown in Table 2, G represents the number of graph convolutional layers.

The results indicate that when the number of hidden layers increases, the GCN can retrieve deeper semantic information, thereby improving the accuracy. Moreover, when the size of the graph convolutional layer is 2, F-score is the highest. However, when the number of hidden layers is extremely large, the semantic information extracted by the GCN network will also increase through the increase in the number of layers, and thus overfitting occurs, which leads to a decrease in F-score.

The model selects six optimizers, namely stochastic gradient descent (SGD), Momentum, adaptive gradient (AdaGrad), Adam, AdaDelta, and root mean square prop (RMSProp) for comparison experiments. The corresponding accuracy rates of different optimizers are shown in Figure 6.

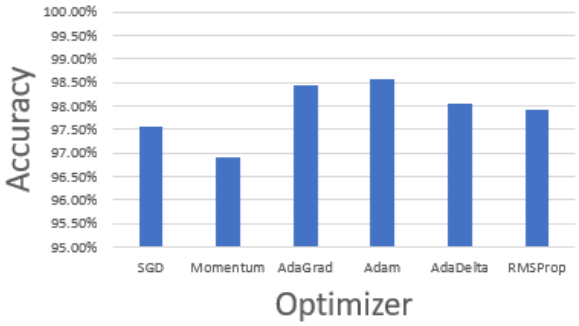

FIGURE 6. Experimental results of optimizers comparisons.

The results indicate that the model training effect is best when the optimizer selects Adam.

The learning rate is usually set within 0.001–10. In this study, we select five learning rates, namely 0.001, 0.01, 0.1, 1, and 10, for comparison experiments. The corresponding accuracy rates of the different learning rates are shown in Figure 7.

The results indicate the best training effect of the model for a learning rate of 0.01.

With repeated experimental tests, after 27 training iterations, the highest accuracy of the overall test set of the model is achieved.

With increasing number of training iterations, the accuracy of the test set improves and eventually stabilizes, as shown in Figure 8.

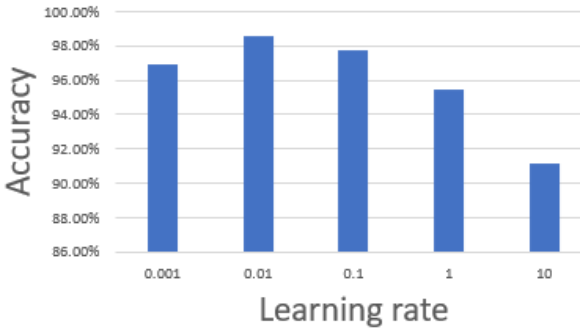

FIGURE 7. Experimental results of learning rates comparisons.

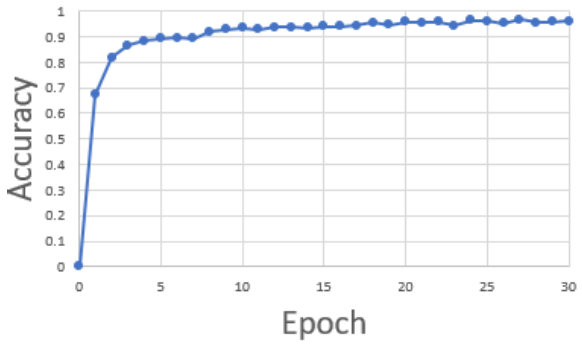

FIGURE 8. Experimental results of epochs comparisons.

TABLE 3. Experiment settings.

Parameters	Values
Embedding	200
GCN-layers	2
GCN-filters	50
GCN-neighbors	25
Dropout	50%
Batch-size	128
Epochs	27
Optimizer	Adam
Learning rate	0.01

In Table 3, "GCN-filters" indicates the number of output channels in the convolutional layer of the graph, "GCN-neighbors" indicates the number of neighbors of each node and "batch-size" indicates the batch quantity. In addition, Dropout is a technique to overcome overfitting by randomly excluding nodes during training. In our experiment, a dropout rate of 50% was applied. We experimented with these settings and found that minor changes did not considerably change the results.

3) EVALUATION METRICS

To quantitatively evaluate the performance of the classifier, we used the following common machine learning performance evaluation indicators: True positive (TP) is the number of samples correctly classified as malicious, true negative (TN) is the number of samples correctly classified

as benign, false positive (FP) is the number of samples misclassified as malicious, and false negative (FN) is the number of samples that are misclassified as benign.

Accuracy (Acc) represents the number of samples correctly classified as benign or malicious.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{11}$$

Precision (p) is the ratio of correct positively labeled instances to all positively labeled instances.

$$Precision = \frac{TP}{TP + FP} \tag{12}$$

Recall (R) is the ratio of correct positively labeled instances to all instances that should have been labeled positive.

$$Recall = \frac{TP}{TP + FN} \tag{13}$$

F-score (F) is the harmonic mean of precision and recall.

$$F - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{14}$$

C. EFFECT OF DATASETS ON MODEL PERFORMANCE

1) EFFECT OF TRAINING SET SIZE ON MODEL PERFORMANCE

This section mainly discusses the effect of the size of training data on the performance of the model. The data set is divided into training and test sets with the same structure. Keeping the test set size constant, the training set is randomly sampled from the previous training set to construct multiple training sets with different degrees of reduction. We retrain the network on the reduced training sets and evaluate the performance of the model on the training and test sets. The performance evaluation standard is F-score. We plot the results on the Figure 9. The ordinate is F-score, and the abscissa is the number of training sets.

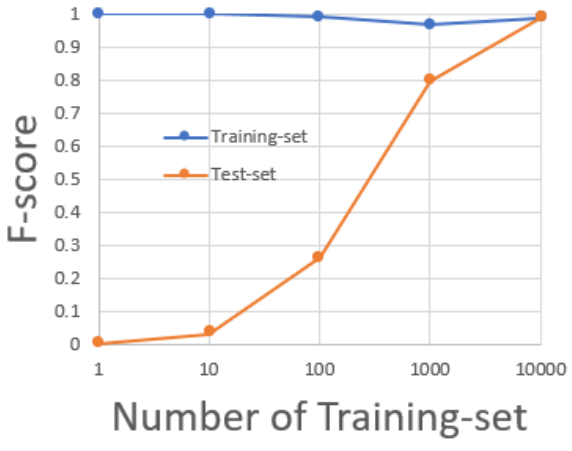

FIGURE 9. Effect of training set size on model performance.

As shown in Figure 9, when the training set size is small, the model performance on the training set is high, but that on the test set is poor, indicating overfitting of the model.

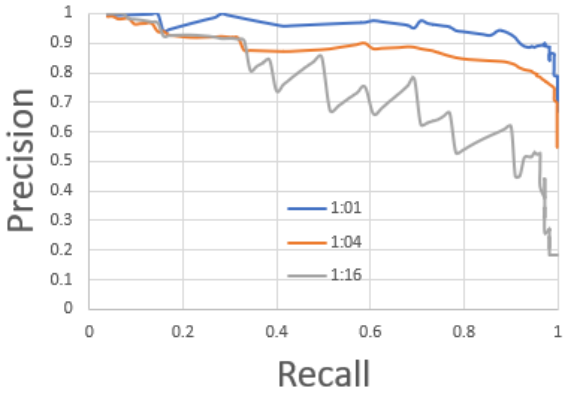

FIGURE 10. Effect of test set distribution on model performance.

Further, as the size of the training set increases, the model performance on the test set continues to improve, indicating that the model has learned to generalize from the training set. Thus, it can be inferred that the model performance will improve with larger training set size.

2) EFFECT OF TEST SET DISTRIBUTION ON MODEL PERFORMANCE

This section discusses the effect of the distribution of positive and negative samples in the test set on the performance of the model. We conducted experiments using three ratios of benign and malware samples, namely 1:1, 1:4, and 1:16, to evaluate the stability of the model while keeping the size of test set constant.

In actual malware detection tasks, the distribution of positive and negative samples is usually not fixed. We use the precision–recall (PR) curve to evaluate the classification performance of the machine learning algorithm. Figure 10 shows the PR curve of the model for varying distribution of the test set. The closer the curve is to (1,1), the more accurate the model classification. However, when the sample distribution in the test set is varied, the PR curve usually changes considerably. The greater the difference in the ratio, the greater the drop in the PR curve. The figure indicates that even in the extreme distribution of the test set, the model has good stability.

D. COMPARISON WITH OTHER METHODS

In this section, we present the results of our proposed method and compare them with the results of other frontier methods. All evaluation results shown in Table 4 are the best results reported in the respective papers. Yen and Sun [5] used the term frequency-inverse document frequency (TF-IDF) method to generate images according to the frequency of characters appearing in the code, and then used the CNN to extract features for analysis. Mclaughlin *et al.* [7] used the opcode call sequence as a feature input into the model. Ren *et al.* [6] resampled the original bytecode of the Android application classes.dex file as an end-to-end Android malware detection method based on deep learning. The com-

TABLE 4. Experimental results of different algorithms comparisons.

Model	Acc	P	R	F
CNN [7]	69.92%	67.34%	74.99%	0.716
DexCNN [6]	93.6%	91.8%	95.7%	0.937
CNN [5]	92.67%	91.99%	92.87%	0.924
AMalNet [4]	99.69%	99.57%	99.82%	0.997
Our method	98.57%	98.71%	98.82%	0.987

TABLE 5. Experimental results of different features comparisons.

Features	Acc	P	R	F
(AMalNet)Api	88.92%	87.34%	89.99%	0.886
(AMalNet)Permission	85.66%	85.35%	88.16%	0.867
(AMalNet)Opcode	93.7%	92.51%	93.19%	0.928
(AMalNet)All features	99.69%	99.57%	99.82%	0.997
Semantic feature	98.57%	98.82%	98.71%	0.987

parison of the methods shows that the proposed algorithm is superior over the previous research methods. Compared with the accuracy of the method of [4], the accuracy of our method is lower because [4] adopts a multi-feature detection method, which uses a combination of the opcode call sequence, permissions required by the APK, and sensitive APIs as features for malicious code detection. Although the required permissions, services, and sensitive APIs of the APK cannot express semantic information, they are very important for malicious code detection.

We varied some of the input features of the study of [4] to use a single feature for malicious code detection. The comparison of those results with the results obtained by the proposed method is shown in Table 5. Comparison of some features indicates that the detection accuracy of [4] is much lower than that of our method; however, if these features are combined, the detection accuracy is relatively high. Therefore, in our future work, we will integrate the semantic features of the code with other features to further improve the effectiveness of malicious code detection.

V. CONCLUSION

This study investigated a new deep learning method based on code semantic features for malicious code detection. We proposed a preprocessing method of the APK file to generate graphics that reflect the semantic features of the code. We used the GCN to learn the features of graphics as code semantic features for malicious code detection. The neighborhood information extracted from the nodes of the graph according to the GCN maintains its original spatial structure and hierarchically expresses high-level semantic structural feature information. Experiments confirmed that the code semantic features are superior to other features. However, we revealed that the effect of the fusion of code semantic features and other features is better than that of only a single feature. Therefore, in our future work, we will integrate the semantic features of the code with other features to further improve the effectiveness of malicious code detection.

REFERENCES

[1] S. Ilham, G. Abderrahim, and B. A. Abdelhakim, "Permission based malware detection in Android devices," in *Proc. 3rd Int. Conf. Smart City Appl.*, 2018, pp. 1–6.
[2] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of Android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 1–29, Jan. 2018.
[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, vol. 14, 2014, pp. 23–26.
[4] X. Pei, L. Yu, and S. Tian, "AMalNet: A deep learning framework based on graph convolutional networks for malware detection," *Comput. Secur.*, vol. 93, Jun. 2020, Art. no. 101792.
[5] Y.-S. Yen and H.-M. Sun, "An Android mutation malware detection based on deep learning using visualization of importance from codes," *Microelectron. Rel.*, vol. 93, pp. 109–114, Feb. 2019.
[6] Z. Ren, H. Wu, Q. Ning, I. Hussain, and B. Chen, "End-to-end malware detection for Android IoT devices using deep learning," *Ad Hoc Netw.*, vol. 101, Apr. 2020, Art. no. 102098.
[7] N. McLaughlin, J. M. del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. J. Ahn, "Deep Android malware detection," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 301–308.
[8] Q. Zhang, J. Chang, G. Meng, S. Xu, S. Xiang, and C. Pan, "Learning graph structure via graph convolutional networks," *Pattern Recognit.*, vol. 95, pp. 308–318, Nov. 2019.
[9] *Androguard*. Accessed: Dec. 11, 2019. [Online]. Available: http://code.google.com/p/androguard/
[10] I. Burguera, U. Zurutuza, and S. Nadjmtehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.
[11] K. Allix, T. F. D. A. Bissyande, J. Klein, and Y. Le Traon, "Machine learning-based malware detection for Android applications: History matters!" Interdiscipl. Centre Secur., Rel. Trust, Univ. Luxembourg, Luxembourg City, Luxembourg, Tech. Rep. 978-2-87971-132-4, 2014.
[12] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Comput. Secur.*, vol. 28, nos. 1–2, pp. 18–28, Feb. 2009.
[13] J. Sahs and L. Khan, "A machine learning approach to Android malware detection," in *Proc. Eur. Intell. Secur. Inform. Conf.*, Aug. 2012, pp. 141–147.
[14] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of Android malware," in *Proc. 7th Eur. Workshop Syst. Secur.*, 2014, p. 5.
[15] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Jun. 2016, pp. 577–582.
[16] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Comput. Secur.*, vol. 77, pp. 578–594, Aug. 2018.
[17] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: Android malware characterization and detection using deep learning," *Tsinghua Sci. Technol.*, vol. 21, no. 1, pp. 114–123, Feb. 2016.
[18] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Proc. Australas. Joint Conf. Artif. Intell.* Cham, Switzerland: Springer, 2016, pp. 137–149.
[19] F. Mercaldo and A. Santone, "Deep learning for image-based mobile malware detection," *J. Comput. Virol. Hacking Techn.*, vol. 16, no. 2, pp. 1–15, Jan. 2020.
[20] A. Azmoodeh, A. Dehghantanha, and K.-K.-R. Choo, "Robust malware detection for Internet of (battlefield) things devices using deep eigenspace learning," *IEEE Trans. Sustain. Comput.*, vol. 4, no. 1, pp. 88–95, Jan. 2019.
[21] X. Shu, L. Zhang, Y. Sun, and J. Tang, "Host-parasite: Graph LSTM-in-LSTM for group activity recognition," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Apr. 2, 2020, doi: 10.1109/TNNLS.2020.2978942.
[22] J. Tang, X. Shu, R. Yan, and L. Zhang, "Coherence constrained graph LSTM for group activity recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, early access, Jul. 15, 2019, doi: 10.1109/TPAMI.2019.2928540.
[23] *Soot*. Accessed: Dec. 11, 2019. [Online]. Available: http://sable.github.io/soot/

[24] D. Marcheggiani and I. Titov, "Encoding sentences with graph convolutional networks for semantic role labeling," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2017, pp. 1506–1515.
[25] *Googleplay*. Accessed: Dec. 11, 2019. [Online]. Available: https://play.google.com/
[26] *Virusshare*. Accessed: Dec. 11, 2019. [Online]. Available: https://virusshare.com/

YU ZHANG was born in Lianyungang, Jiangsu, China, in 1996. He received the B.E. degree in electronics science and technology from Beijing Jiaotong University, in 2018. He is currently pursuing the M.E. degree with Information Engineering University.

His research interest includes forensics of mobile device.

BINGLONG LI received the B.E. and M.E. degrees in information security from Information Engineering University, Zhengzhou, Henan, China.

He is currently an Associate Professor of cyber security with Information Engineering University. He has been funded under the National Natural Science Foundation of China (NSFC). His main research interests include digital forensics and information security.