IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received May 5, 2017, accepted June 14, 2017, date of publication June 30, 2017, date of current version September 19, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2720160

An Android Malware Detection Approach Using Community Structures of Weighted Function Call Graphs

YAO DU[1], JUNFENG WANG[2], AND QI LI[3]
[1]College of Computer Science, Sichuan University, Chengdu 610065, China
[2]School of Aeronautics and Astronautics and College of Computer Science, Sichuan University, Chengdu 610065, China
[3]College of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Junfeng Wang (wangjf@scu.edu.cn)

This work was supported in part by the National Key Research and Development Program under Grant 2016QY06X1205 and Grant 2016YFB0800605, in part by the National Natural Science Foundation of China under Grant 91438120, Grant 91338107, and Grant 91438119, and in part by the Technology Research and Development Program of Sichuan, China, under Grant 2016ZR0087.

ABSTRACT With the development of code obfuscation and application repackaging technologies, an increasing number of structural information-based methods have been proposed for malware detection. Although, many offer improved detection accuracy via a similarity comparison of specific graphs, they still face limitations in terms of computation time and the need for manual operation. In this paper, we present a new malware detection method that automatically divides a function call graph into community structures. The features of these community structures can then be used to detect malware. Our method reduces the computation time by improving the Girvan–Newman algorithm and using machine learning classification instead of a similarity comparison of subgraphs. To evaluate our method, 5040 malware samples and 8750 benign samples were collected as an experimental data set. The evaluation results show that the detection accuracy of our method is higher than that of three well-known anti-virus software and two previous control flow graph-based methods for many malware families. The runtime performance of our method exhibits a clear improvement over the GN algorithm for community structure generation.

INDEX TERMS Malware, community structures, machine learning.

I. INTRODUCTION

With the development of mobile devices, the Android operating system has become one of the world's most popular smartphone platforms, offering an excellent hardware basis and low cost. Market research by Gartner [1] found that more than 298 million Android smartphones were purchased globally in the third quarter of 2015, rising to 352 million in the fourth quarter of 2016, indicating an 81.7% share of the global smartphone market [2].

Because of the large number of users, the Android platform has become the main target of attacks. A report [3] shows that Android was the target of 97% of global mobile malware in 2013. In 2016, the number of Trojans contained in Android adware increased by nearly 700,000 [4].

Existing signature-based malware detection methods face constant challenges. Their manual analysis patterns always lag behind the malware and are inefficient for zero-day malware detection. Moreover, the repackaging technology allows hackers to quickly generate a large number of malware variants. This is also a big challenge for signature-based detection methods.

To solve these problems and improve the detection accuracy, many prior studies have focused on the structural features of applications [5]–[8]. Researchers have shown that a large number of new malware variations are similar to existing applications, and thus comparing the similarity of specific graphs extracted from applications seems a feasible approach [9], [10]. However, the similarity comparison of graphs is a complicated task. It involves restoring a significant number of subgraphs as features, and so matching the target graphs with these features can be very time consuming.

In this paper, we present a new malware detection method based on the analysis of community structures of function call graphs. First, our method applies reverse engineering to obtain the function call graph of an Android application. Second, the function call graph is weighted by sensitive permissions and application programming interfaces (APIs). Third, the community structures are extracted from the weighted function call graph. Finally, a machine learning classification process is used to detect malware. In an evaluation of 13,790 Android applications, our method achieves 96.5% accuracy in detecting unknown malware.

The main contributions of our method are as follows.

(1) A new community structure generation model that is suitable for Android malware detection is proposed. In our model, community structures are generated according to structural information of the function call graph as well as sensitive static characteristics of the Android application. This process makes the community structures more meaningful in reflecting malicious behavior.

(2) Several effective improvements are designed to improve the GN algorithm [11]. These improvements are shown to make the community structure generation process more efficient.

(3) Machine learning classification is used for malware detection. This means we abandon the complex comparison of the similarities between a large amount of subgraphs to identify malware, and treat the malware detection as a machine learning classification problem.

The remainder of this paper is organized as follows. Some relevant work is discussed in section II. The architecture and specific steps of our method are introduced in section III. The analysis of function call graphs and community structures is an important part of this paper, and is discussed in detail in section IV. Section V describes several experiments to evaluate our model. Finally, a general summary and ideas for future work are presented in section VI.

II. RELATED WORK

In recent years, the explosion of Android viruses has made malware detection more difficult. A comprehensive and systematic virus database is a key factor in security research. Zhou et al. [12] described a wide-ranging malware collection and analysis approach. They collected more than 1,200 malware samples and systematically analyzed them. Then, four well-known mobile security applications were chosen to test these samples, achieving detection rates ranging from 20.2-79.6%. The low detection rates indicate that their malware samples require further research. Thus, we added them to our experimental dataset.

Static analysis [13]–[16] is widely used in Android malware detection for two main reasons. First, the detection processes will cover all codes, including seldom-used functions. Second, the detection speed is fast, as it does not rely on a virtual execution environment. In static analysis, permissions and APIs are useful features. For example, Talha et al. [17] proposed a permission-based malicious code testing tool called APK Auditor. This offers a user client for granting application analysis requests, an independent database to store application features, and a central server to manage the database and user client. In their experiments, APK Auditor analyzed more than 10,000 Android applications with 88% detection accuracy. Qiao et al. [18] proposed a malware detection method based on permissions and APIs. They extracted numerical and binary features from Android applications for both quantitative and qualitative evaluation. In addition, the classification efficiency of several frequently used machine learning algorithms was compared.

Kate et al. [19] described a two-step malware detection method. In step one, two kinds of Bloom filters are used to classify samples as malware or normal based on permissions. In step two, more code features are added to the permissions. These features are classified by a Naive Bayes algorithm to identify malware.

Arp et al. [20] proposed a lightweight detection method called DREBIN. This method can extract eight types of static features from Android applications. These features are then input to vector space models for classification by a support vector machine (SVM) algorithm. Their method achieved 94% detection accuracy on 123,453 Android applications.

Schmeelk et al. [21], analyzed the architecture of Android applications and summarized the main technologies of static malware detection methods. Among these technologies, they focused on permission leakage and privacy concerns.

However, many static malware detection methods are easily influenced by code confusion mechanisms. To solve this problem, researchers began to focus on the internal structure of Android applications. Suarez-Tangil et al. [22] analyzed the code structures of Android applications and proposed a new detection mechanism called Dendroid. This adopts text mining and information retrieval technologies, allowing malicious applications to be identified and automatically classified into the appropriate virus family. Kwon et al. [23] proposed a graph-based mechanism, called DroidGraph, which can extract APIs from decompilation code and build an API call graph of the application. The API call graph is then used to build a semantic graph that better reflects the malicious features and improves the recognition efficiency. DroidGraph can achieve 87% detection accuracy without any false positives for benign samples.

Atici et al. [24] proposed a control flow graph-based malware detection method. They extract features named "code chunks" from the control flow graph, then use the classification and regression tree (CART) algorithm to classify these features. The method can obtain 96.3% classification accuracy.

Zhang et al. [25] proposed an Android malware detection method based on the features of weighted contextual API dependency graphs. In their method, graph similarities are used to identify malware with 93% detection accuracy.

Another effective method that is resistant to code confusion mechanisms is dynamic analysis. In dynamic analysis, applications are executed in a virtual environment. Runtime information is recorded to generate dynamic features for malware detection. For example, Enck et al. [26] proposed TaintDroid, a virtualization-based malware detection method that

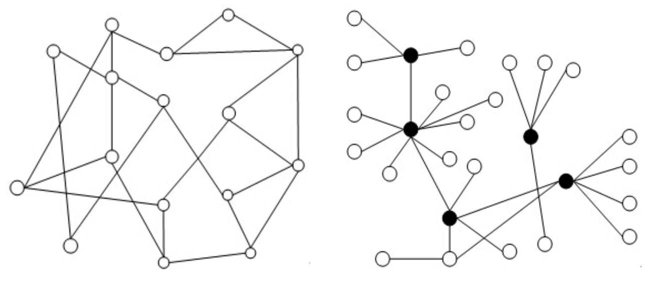


FIGURE 1. Graph structure comparison. (a) The strucutre of random networks. (b) A part of function call graph of 53dc08f08005f374a957afa44607ab52f205b684.apk.

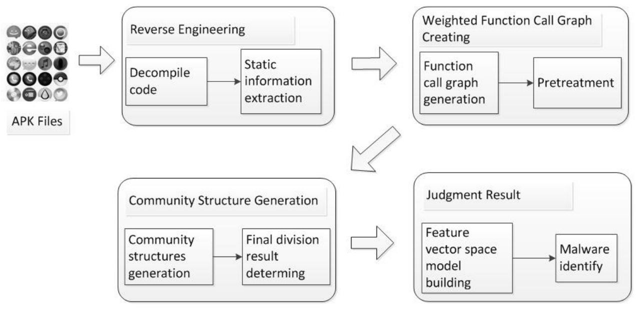

APK Files
Reverse Engineering
Decompile code
Static information extraction
Weighted Function Call Graph Creating
Function call graph generation
Pretreatment
Community Structure Generation
Community structures generation
Final division result determing
Judgment Result
Feature vector space model building
Malware identify

FIGURE 2. The architecture of our malware detection model.

can trace the flow of sensitive information. In an evaluation of 30 Android applications, TaintDroid found 20 applications had misused users' private information. DroidScope [27] is also a virtualization-based malicious code detection method. It can identify malwares by analyzing semantics features of operating system level and Java level.

Vidas and Christin [28] illustrated the fundamental limitations of virtualization-based malware detection methods. They offered several Android malware detection techniques to prove these limitations may be evaded.

StaDynA [29], MARVIN [30] and the method proposed by Zeng et al. [31] are methods that can combine static and dynamic analysis for Android malware detection.

Although dynamic analysis is also effective in malware detection, the virtual environment is very resource consuming. This shortcoming means that large-scale malware detection through dynamic analysis is a significant challenge.

Different from the above studies, our approach analyzes Android applications' structural information based on community structures rather than directly using function call graphs or control flow graph, etc. As these community structures contain useful information about closely related functions, we extract features from them for malware detection. In addition, our method places particular emphasis on optimizing the GN algorithm and machine learning classification process, whilst also reducing the computational load.

III. MECHANISM OVERVIEW

Compared to random networks, the function call graph of an Android application has its own structural characteristics, as shown in Fig. 1. In function call graphs, the different function call frequencies of each method lead to a nonuniform distribution of edges across the whole graph. This situation exists in many node sets of the graph, indicating that the nodes within a common node set are closely connected, but have few connections with nodes in other node sets. This structural characteristic indicates the existence of potential community structures in the graph. In Android applications, such community structures may contain a wealth of information. The functions of each community structure are expected to have common attributes and behaviors that can be used to distinguish malicious codes. To find these community structures and identify malware, a new malware detection architecture is proposed. As shown in Fig. 2, the execution process has the following steps:

(1) **Reverse engineering.** The Android package (APK) file is an Android application's execution file. It is essentially a zip (compressed) file with a different suffix. APK files are composed of several subfiles, including ".dex" files, "AndroidManifest.xml" file, and many resource files. To analyze these files, the Baksmali tool is used to decompile them into ".smali" files.

(2) **Weighted function call graph creating.** Functions and their call relationships can be extracted from the ".smali" codes. These are used to build the function call graph. Many common Android malicious attack modes, such as repackaging technology and malicious code injection, are less likely to occur in isolated nodes. Thus, they are removed to reduce the computational complexity. Every node of the function call graph is then weighted according to sensitive permissions and APIs.

(3) **Community structure generation.** In this phase, community structures are generated from the weighted function call graph. The GN algorithm is used to finish this task because of its high accuracy. GN algorithm can divide the whole function call graph into community structures by constantly deleting edges with the highest betweenness. However, the GN algorithm involves complex computations and can be time consuming. Hence, we improve the GN algorithm to make it more efficient.

(4) **Malicious code judgment.** Features can be extracted from the community structures generated in step (3). These features are input to a vector space model to build classification rules using machine learning algorithms. The final malware identification results are obtained using these rules.

IV. GRAPH ANALYSIS AND MALWARE DETECTION

In this section, the details of graph generation and the malware detection strategy are discussed. The content contains three main parts: (1) Construct a weighted function call graph; (2) improve the GN algorithm for community structure generation; and (3) identify malware using the features extracted from the community structures.

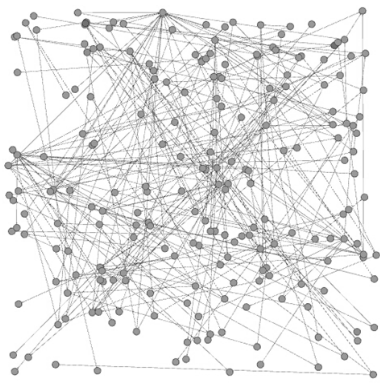


FIGURE 3. **Function call graph of 53dc08f08005f374a957afa44607ab52f205b684.apk.**

A. WEIGHTED FUNCTION CALL GRAPH

In reverse engineering, methods and their call relations can be extracted from ".smali" codes. All call sequences for each method are collected to build a function call graph, as shown in Fig. 3. Then, a two-step pretreatment is applied. The first step is to delete isolated nodes. There are many nodes with very few edges that are isolated in the function call graph. It is not only difficult to reflect the behavior characteristics of the application in these nodes, but they also increase the complexity of the calculation. Thus, isolated nodes are first removed. The second step is to add a weight to each node. We analyzed the applications in our experimental dataset and obtained statistics on the appearance frequency of permissions and APIs. The malicious behavior value M can be calculated by the following formula:

$$M = F1/(F1 + F2). \tag{1}$$

where F1 represents a permission or system API's appearance frequency in malware and F2 represents a permission or system API's appearance frequency in benign applications. A node's weight is the sum of the malicious behavior values of its related sensitive permissions and system APIs. Once this pretreatment work has been completed, a weighted function call graph is constructed, and this graph is used in the following community structure generation step.

B. COMMUNITY STRUCTURE

Our community structure generation model is based on the GN algorithm, which is a classic community structure generation algorithm. The main idea of the GN algorithm is to continuously remove the edges with the highest betweenness, which is defined as the number of shortest paths between pairs of vertices that run along a particular edge. The betweenness of the remaining edges is then recalculated, and the process is repeated until all edges have been deleted.

However, the GN algorithm suffers from a long computation time. According to its definition, the repeated calculation process gives the GN algorithm a time complexity of O(m^2n), where m represents the number of edges and n represents the number of nodes. In addition, the community structures obtained by a pure structural generation process struggle to reflect the behavior of the application. Thus, we improve the GN algorithm as follows:

1) FAST BETWEENNESS CALCULATION

The calculation of an edge's betweenness can be adjusted by the following two steps: First, our method identifies the key node of a graph. The key node is chosen according to the degree and weight of its directly connected nodes. It is defined as:

$$C = C_i * w(i) = (D_i/n - 1) * ((\sum_{m=1}^{k} w(q_m)) - k + 1). \tag{2}$$

where n represents the number of nodes in the whole graph, D_i is the degree of node i, k is the number of nodes that are directly connected to node i, and w(q_m) represents the weight of node m in node set k.

Second, an edge's betweenness is calculated as the number of shortest paths between the key node and the other nodes that run along that edge. To ensure that important edges are not easily deleted, the edge weights must be considered. This depends on the weight of the two nodes attached to the edge. Suppose a graph is defined as G = (V, E), where V is the set of nodes and E is the set of edges. The betweenness B_b can be calculated as:

$$B_b = (\sum_{v_i \epsilon V, v_j \epsilon V, e \epsilon E} \wp_{v_i v_j}(e))/g_e. \tag{3}$$

where $\wp_{v_i v_j}(e)$ is the shortest path between node i and node j that contains edge e. g_e is the weight of edge e.

2) REDEFINITION OF TERMINATION CONDITION

The GN algorithm does not define a termination condition to achieve the best graph division result. Therefore, the function Q [32] is proposed as the termination criterion:

$$Q = \sum \left(e_{mm} - {a_m}^2\right). \tag{4}$$

Suppose the number of community structures of a function call graph is k. A k×k symmetric matrix can be defined. $\sum e_{mm}$ (m<k) is the sum of the diagonal elements of the symmetric matrix, and represents the fraction of edges that connect each node in every community structure to all edges of the function call graph. $\sum {a_m}^2$ represents the fraction of edges that connect each node in community structure m to all edges of the function call graph.

In general, Q ranges from 0.3-0.7. Higher values of Q represent better graph division results. The upper limit of Q is 1. For example, Fig. 4 shows the change in Q during the community structure generation process of sample "4de0d8997949265a4b5647bb9f9d42926bd88191.apk." The value of Q increases from 0.4 to 0.7 before falling back to 0.4.

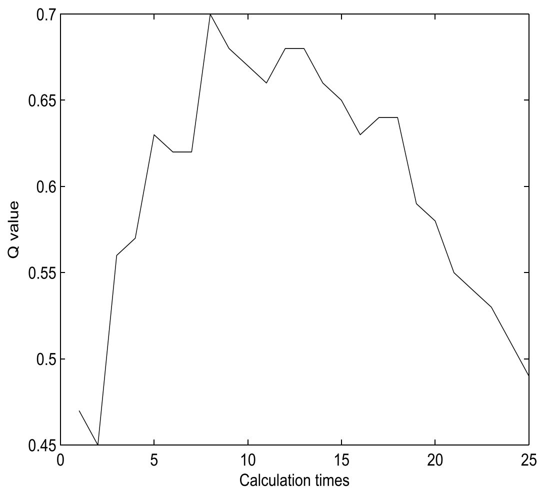


FIGURE 4. The change of Q in the community structure generation of 4de0d8997949265a4b5647bb9f9d42926bd88191.apk.

In our method, the stop condition is determined by calculating ΔQ, which is defined as:

$$\Delta Q = maxQ - currentQ. \tag{5}$$

If $\Delta Q >= 0.1$, terminate the algorithm.

If $0.1 > \Delta Q >= 0$, continue the algorithm.

If $\Delta Q < 0$, then $maxQ = currentQ$, and the algorithm will continue.

3) DETERMINATION OF FINAL DIVISION RESULT

New community structures are generated for each of the multiple computations of Q. These community structures are recorded as one division result. An ideal division result should satisfy the following two conditions at the same time: (1) the nodes in the same community structure are closely connected; (2) a single community structure contains more nodes with high malicious behavior values and fewer packages. To achieve this goal, an automatic evaluation model is established. Suppose a single community structure can be described as:

$$C_i = \Big\{ V = (v_1, v_2, \ldots, v_m), N = (n_1, n_2, \ldots, n_n), \\ P = (p_1, p_2, \ldots, p_j) \Big\}. \tag{6}$$

where C_i represents community structure i, n is the number of nodes in the community structure, V denotes the nodes with malicious behavior values in community structure i, N denotes all nodes in community structure i, and P denotes the packages in community structure i.

According to the information about C_i, an evaluation score S can be calculated and assigned to each division result, as shown in (7). The division result with the highest evaluation score is selected as the ideal division result.

$$S = \sum_{1}^{k} (\frac{V}{N})^2 \cdot \frac{\alpha}{P} \cdot (1 + \beta Q). \tag{7}$$

where k is the number of community structures in a division result, V is the number of nodes with malicious behavior values in each community structure, N is the number of nodes in each community structure, P is the number of packages in each community structure, and α, β are constant coefficients.

In summary, the community structure generation process can be described as Algorithm 1.

Algorithm 1 Community Structure Generation

```
1:  Input: Weighted function call graph
2:  Repeat:
3:      Input nodes and their call relations of the weighted
        function call graph (for the first time) or community
        structures
4:      if (the graphs are not composed of isolated nodes)
5:      then
6:          Obtain key nodes of graphs
7:          Calculate edges' betweenness and Q
8:          Delete the edges with the highest betweenness
9:          Generate community structures and record them
            as one division result
10:     else
11:         Break the loop
12: Until $\Delta Q >= 0.1$
13: Determine the final division result
14: Output: Community structures
```

C. MALWARE DETECTION

A comprehensive malware detection model is established based on machine learning classification. Its essential features can be extracted from the community structures generated in the steps of subsection IV.B. These features are as follows: (1) The number of community structures with malicious features; (2) The highest malicious behavior value of community structures, and (3) The sum of malicious behavior values of community structures in the application. The core process of the machine learning classification is to establish linkages between the features and malicious behavior (or benign behavior). The machine learning classification consists of a training phase and a testing phase.

In the training phase, every application in the training sample dataset is assigned a label that indicates whether it is malware or benign. The features and labels of each application are then input to a vector space model to train the classifier. In the testing phase, features are extracted from unlabeled applications and classified by the trained classifier to identify malware.

To evaluate the performance of the classifier, four metrics are used: the true positive rate (TPR), false positive rate (FPR), precision, and receiver operating characteristic (ROC) curve. TPR represents the proportion of positive samples recognized by the classifier, whereas FPR represents the proportion of negative samples that are wrongly judged as positive samples by the classifier. Precision represents the proportion of true positive samples out of all the positive

TABLE 1. Information of malware samples.

Number	Source	Malware family name
1260	Genome Project	ADRD, Fakeplayer, DroidKungFu, CoinPirate, Asroot, AnserverBot, DroidDream, P-japps, Zsone, etc.
3780	The third party market	FakeInst, sysService, Jxt, sex-player, GingerMast, Faker91, SkullKey, Smishing, ZooTiger, etc.

TABLE 2. Information of benign samples.

Number	Source	Category name
4200	Google play	Security, Tools, Travel, Health, Video, Netgames, etc.
4550	The third party market	Office Business, Sports, Health, Browsers, Shopping, Payment, etc.

samples that are recognized by the classifier. The ROC curve reflects the relationship between TPR and FPR. Higher values of the area under the ROC curve (AUC) indicate higher TPR and lower FPR. The closer the value of AUC is to 1, the better the classification efficiency.

All classifications were performed by WEKA [33], which is a free and open source data mining software based on the Java development environment. WEKA can process multiple batches of data in parallel and view the models generated by the classifier for each step of cross-validation. It also can visualize the effectiveness of classifiers.

V. EXPERIMENTS AND EVALUATION

A. EXPERIMENTAL DATASET

The experimental dataset contains 13,790 Android applications in total, with 5040 malware samples and 8750 benign samples. Of these, 1260 malware samples were collected from the Android Malware Genome Project [34], and the others were obtained from other websites, such as [35]. The benign samples were collected from Google Play [36] and some official third-party application markets. The categories of our application samples are listed in Tables 1 and 2.

B. EFFECTIVENESS OF CLASSIFICATION PROCESS

In the malware detection phase, the efficiency of the classifier is an important factor in the final detection accuracy. To obtain a suitable classifier, several common classification algorithms were compared. As mentioned in section II, many previous studies have applied machine learning classification to Android malware detection, with decision trees, Bayesian

TABLE 3. The comparison of different classification algorithms.

Classifier	TP Rate	FP Rate	Precision	ROC Area
NaiveBayes	0.976	0.034	0.976	0.992
J48	0.968	0.053	0.968	0.98
BayesNet	0.98	0.024	0.98	0.995
SVM	0.964	0.07	0.964	0.947

approaches, and SVM algorithms the most frequently used [37], [38].

Decision tree algorithms are suitable for handling samples with missing attributes or irrelevant features. Such methods are fast, but may lead to overfitting. The SVM algorithm provides a good theoretical guarantee against overfitting. It is good at solving high-dimensional problems (large feature vector spaces). However, its efficiency is poor when the number of samples is large. Moreover, the SVM algorithm has no general solution for nonlinear problems, and sometimes it is difficult to find a suitable kernel function. Bayesian algorithms converge faster than discriminant models such as decision trees and SVM. Thus, this method should achieve good performance, even with small sets of training data.

Table 3 shows the classification performance of a decision tree (J48), SVM, NaiveBayes, and BayesNet. These classification results are based on the features extracted from the community structures. The experimental results indicate that the Bayes algorithms achieve better results than J48 and SVM in all metrics. Based on this dataset, BayesNet is the best algorithm.

C. COMPARISON EXPERIMENT OF MALWARE FAMILIES

An important part of our experimental database is the malware samples collected from the Genome Project. It consist of 49 malware families and 1260 typical malicious applications. By examining the detection performance of each family, we can identify the detection stability of our model. Note that several malware families contain very few samples, such as "Zitmo," which has only one sample. These malware families were removed from this experiment and placed into an isolated testing sample set for use in the experiment described in the next subsection.

The Genome Project samples have been used as test datasets in previous studies. We used our method to detect some of the main families of the dataset, and compared the detection results with those of several previous approaches. The first is composed of two anti-virus applications, AVG Antivirus Free (AVG) and Norton Mobile Security Lite (Norton) [12]. The second is Androguard [39], an excellent decompilation and virus detection tool that has been widely used in recent studies. The third is "Dendroid," [22] which was mentioned in section II. This is an efficient method for coping with repackaging technologies. The fourth [24], referred to as the control flow graph-based method, extracts

TABLE 4. The comparison of detection accuracy among former works and our method.

Malware family name	AVG(%)	Norton(%)	Androguard(%)	Dendroid(%)	Control flow graph-based method(%)	Our method(%)
ADRD	100.0	22.7	59.1	100.0	95.5	100.0
AnserverBot	88.2	1.0	0.0	95.7	98.9	100.0
BeanBot	0.0	0.0	0.0	100.0	62.5	100.0
Bgserv	100.0	22.2	0.0	100.0	88.9	100.0
DroidDream	68.7	56.2	93.8	100.0	93.8	93.8
DroidDreamLight	30.4	23.9	28.2	100.0	100.0	100.0
DroidKungFu1	100.0	5.8	0.0	88.24	97.1	97.1
DroidKungFu3	0.0	0.3	0.0	91.56	100	98.7
Geinimi	100.0	55.0	97.1	100.0	100.0	100.0
GoldDream	61.7	0.0	40.4	100.0	100.0	100.0
Pjapps	75.8	44.8	41.4	100.0	97.8	96.6
jSMSHider	68.7	81.2	0.0	100.0	100.0	100.0
Plankton	100.0	9.0	18.2	100.0	90.9	100.0
YZHC	4.5	13.6	95.5	100.0	95.5	100.0
Zsone	100.0	41.6	100.0	100.0	100.0	100.0
KMin	100.0	76.9	78.8	100.0	100.0	100.0
RoguePPush	100.0	0.0	100.0	100.0	98.9	100.0

features and builds control flow graphs from decompiled codes. Machine learning algorithms are then used for the classification and identification of malicious codes. This method is similar to ours, and offers high detection accuracy for the DroidKunfu malware family. Table 4 presents the detection accuracy results for each method. The experimental results show that the detection accuracy of our model ranges from 93.8-100%. Among most malware families, our method achieves better detection accuracy than AVG, Norton, Androguard, and the control flow graph-based method. The average detection accuracy of our method is 99.1%, which is the highest of all methods.

D. MALWARE DETECTION PERFORMANCE

In this subsection, the detection performance of our method is evaluated in terms of unknown malware detection with a compressed training set and through an analysis of the runtime improvement in community structure generation.

In the first experiment, an isolated testing sample set was established. This contains all categories of applications of our dataset that are not used for training. Thus, these applications can be treated as an unknown sample set. To evaluate the detection performance further, we compressed the size of the training set to 10% of the total samples. As shown in Fig. 5, our method achieves 81% detection accuracy with a training set of 10% of the total. The highest detection accuracy of 96.5% is reached when the training set size is 100% of the total. This result indicates that the detection accuracy is not over-reliant on the training set.

The second experiment examines the improvement in runtime performance given by the enhanced GN algorithm.

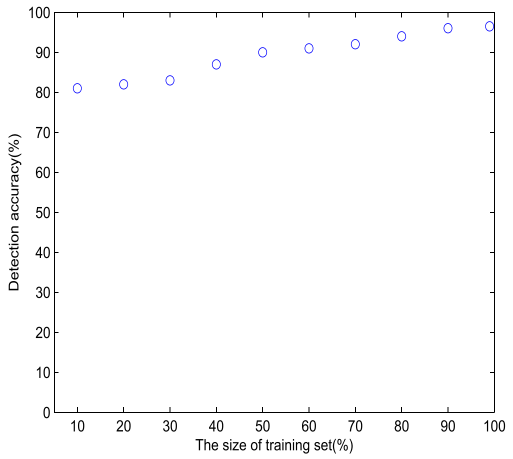


FIGURE 5. Detection accuracy of different sizes of training sets.

As mentioned in section IV, the time complexity of our method is obviously reduced compared with that of the GN algorithm. Fig. 6 compares the execution times of the GN algorithm and our method for community structure generation. The testing samples were chosen based on the number of nodes, which ranged from 135-644. Experimental results show that the execution time of both algorithms is less than 30 s when the node number of the sample is less than 169. However, the execution time of the GN algorithm increases much faster than that of our method when the node number exceeds 216. Moreover, when the node number reaches 1000, the average execution time of the GN algorithm exceeds 30 min, compared with just 5 min for our method.

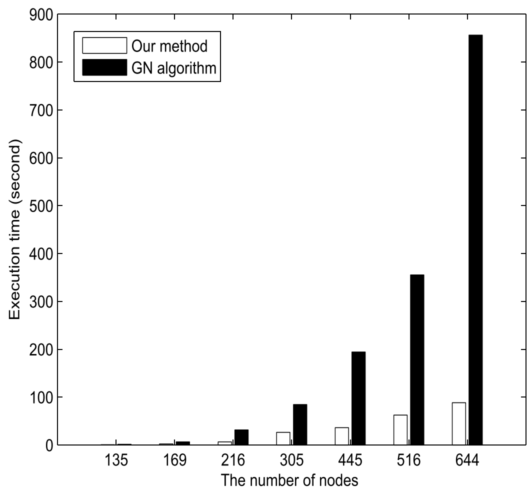


FIGURE 6. **The execution time comparison between our method and the GN algorithm.**

VI. CONCLUSIONS AND FUTURE WORK

In this paper, a new Android malware detection method that uses community structure analysis has been introduced. The proposed method adopts three features extracted from community structures to detect malware by machine learning classification. The method was shown to be efficient from three aspects. First, the malware detection accuracy of our method was compared with that of three well-known anti-virus programs and two previous detection techniques. The comparison results show that our method achieves the highest average detection accuracy across the 17 main malware families of our experimental dataset. Second, the execution time of our method is clearly less than that of the GN algorithm for community structure generation. Finally, an unknown application detection experiment was conducted with ten different sizes of training set. The experimental results indicate that our method can achieve stable and high detection accuracy even with small training sets. In addition, the classification performance of several common machine learning algorithms was compared.

Though our method achieves good malware detection performance, the experimental results indicate that the runtime performance of our method could be improved for large function call graphs. In future work, more efficient algorithms should be designed for community structure generation and feature extraction. At the same time, the rapid growth of new malware families presents another challenge. We plan to choose more semantic-based features to deal with this issue.

REFERENCES

[1] Egham. (Nov. 2015). *Gartner Says Emerging Markets Drove Worldwide Smartphone Sales to 15.5 Percent Growth in Third Quarter of 2015.* [Online]. Available: http://www.gartner.com/newsroom/id/3169417

[2] Egham. (Feb. 2017). *Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016. U.K.* [Online]. Available: http://www.gartner.com/newsroom/id/3609817

[3] Gordon Kelly. (Mar. 2014). *Report: 97% Of Mobile Malware Is On Android. This Is The Easy Way You Stay Safe*. [Online]. Available: 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/

[4] McAfee Labs. (Dec. 2016). *Threats Report*. [Online]. Available: https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-dec-2016.pdf

[5] G. Canfora, F. Mercaldo, and C. A. Visaggio, "An HMM and structural entropy based detector for Android malware: An empirical study," *Comput. Secur.*, vol. 61, pp. 1–18, Aug. 2016.

[6] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang, "Detect Android malware variants using component based topology graph," in *Proc. IEEE 13th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Apr. 2014, pp. 406–413.

[7] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "HI-CFG: Construction by binary analysis, and application to attack polymorphism," in *Proc. 18th Eur. Symp. Res. Comput. Secur.*, 2013, pp. 164–181.

[8] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. 35th IEEE Symp. Secur. Privacy*, May 2014, pp. 590–640.

[9] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Proc. Eur. Symp. Res. Comput. Secur. (ESORICS)*, 2012, pp. 2454–2456.

[10] S. Cesare and X. Yang, "Malware variant detection using similarity search over sets of control flow graphs," in *Proc. Int. Conf. Trust, Secur. Privacy Comput. Commun.*, 2011, pp. 181–189.

[11] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proc. Nat. Acad. Sci. USA*, vol. 99, no. 12, pp. 7821–7826, Apr. 2002.

[12] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, Apr. 2012, pp. 95–109.

[13] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1036–1046.

[14] S. Arzt *et al.*, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[15] Z. Yang and M. Yang, "LeakMiner: Detect information leakage on Android with static taint analysis," in *Proc. 3rd World Congr. Softw. Eng.*, 2012, pp. 101–104.

[16] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Reflection-aware static analysis of Android apps," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Sep. 2016, pp. 756–761.

[17] K. A. Talha, D. I. Alper, and C. Aydin, "APK auditor: Permission-based Android malware detection system," *Digit. Investigat.*, vol. 13, pp. 1–14, Jun. 2015.

[18] M. Qiao, A. H. Sung, and Q. Liu, "Merging permission and API features for Android malware detection," in *Proc. Int. Congr. Adv. Appl. Informat. (IIAI)*, 2016, pp. 566–571.

[19] P. M. Kate and S. V. Dhavale, "Two phase static analysis technique for Android malware detection," in *Proc. 3rd Int. Symp. Women Comput. Informat.*, 2015, pp. 650–655.

[20] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of Android malware in your pocket," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.

[21] S. Schmeelk, J. Yang, and A. Aho, "Android malware static analysis techniques," in *Proc. 10th Annu. Cyber Inf. Secur. Res. Conf.*, 2015, p. 5.

[22] 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 Syst. Appl.*, vol. 41, no. 4, pp. 1104–1117, Mar. 2014.

[23] J. Kwon, J. Jeong, J. Lee, and H. Lee, "DroidGraph: Discovering Android malware by analyzing semantic behavior," in *Proc. IEEE Conf. Commun. Netw. Secur.*, Sep. 2014, pp. 498–499.

[24] M. A. Atici, S. Sagiroglu, and I. A. Dogru, "Android malware analysis approach based on control flow graphs and machine learning algorithms," in *Proc. Int. Symp. Digit. Forensic Secur.*, 2016, pp. 26–31.

[25] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1105–1116.

[26] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th Symp. Oper. Syst. Design Implement. (USENIX)*, 2010, pp. 1105–1116.

[27] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Secur. Symp.*, 2012, p. 29.

[28] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proc. 9th ACM Symp. Inf. Comput. Commun. Secur.*, 2014, pp. 447–458.

[29] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, 2015, pp. 37–48.

[30] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, 2015, pp. 422–433.

[31] H. Zeng, Y. Ren, Q. X. Wang, N. Q. He, and X. Y. Ding, "Detecting malware and evaluating risk of app using Android permission-API system," in *Proc. 11th Int. Comput. Conf. Wavelet Active Media Technol. Inf. Process.*, 2014, pp. 440–443.

[32] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 70, no. 6, p. 066111, Dec. 2004.

[33] (2016). *Weka 3: Data Mining Software in Java*. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/index.html

[34] (2013). *Android Malware Genome Project*. [Online]. Available: http://www.malgenomeproject.org/

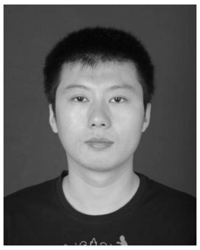
[35] (2015). *Contagio Mobile-Mobile Malware Mini Dump*. [Online]. Available: http://contagiominidump.blogspot.com/

[36] (2016). *Google Play Store*. [Online]. Available: https://play.google.com/store/

[37] Z. Zhao, J. Wang, and C. Wang, "An unknown malware detection scheme based on the features of graph," *Secur. Commun. Netw.*, vol. 6, no. 2, pp. 239–246, Feb. 2013.

[38] Y. Du, X. Wang, and J. Wang, "A static Android malicious code detection method based on multi-source fusion," *Secur. Commun. Netw.*, vol. 8, no. 17, pp. 3238–3246, Mar. 2015.

[39] (2016). *Androguard Project*. [Online]. Available: http://code.google.com/p/androguard/



YAO DU received the M.E. degree in software engineering from the University of Electronic Science and Technology of China, Sichuan, in 2008. He is currently pursuing the Ph.D. degree with the College of Computer Science, Sichuan University. His recent research interests include machine learning, information security, and systems engineering.

JUNFENG WANG received the M.S. degree in computer application technology from the Chongqing University of Posts and Telecommunications, Chongqing, in 2001, and the Ph.D. degree in computer science from the University of Electronic Science and Technology of China, Chengdu, in 2004. From 2004 to 2006, he held a postdoctoral position with the Institute of Software, Chinese Academy of Sciences. He is with the School of Aeronautics and Astronautics and College of Computer Science, Sichuan University, as a Professor. His recent research interests include spatial information networks, network and information security, and intelligent transportation system.

QI LI received the Ph.D. degree in computer science and technology from the Beijing University of Posts and Telecommunications, China, in 2010. She is currently an Associate Professor with the Information Security Center, State Key Laboratory of Networking and Switching Technology, School of Computer Science, Beijing University of Posts and Telecommunications, China. Her current research interests include on software security.