Hindawi
Mobile Information Systems
Volume 2018, Article ID 1649703, 12 pages
https://doi.org/10.1155/2018/1649703

Research Article

Detecting Android Malwares with High-Efficient Hybrid Analyzing Methods

Yu Liu,[1] **Kai Guo**,[1] **Xiangdong Huang**,[1] **Zhou Zhou**,[2] **and Yichi Zhang**[1]

[1]*School of Electronic Information Engineering, Tianjin University, Tianjin 30072, China*
[2]*Teachers College of Columbia University, New York, NY 10027, USA*

Correspondence should be addressed to Xiangdong Huang; xdhuang@tju.edu.cn

Received 30 July 2017; Revised 8 October 2017; Accepted 29 October 2017; Published 13 March 2018

Academic Editor: Xiapu Luo

In order to tackle the security issues caused by malwares of Android OS, we proposed a high-efficient hybrid-detecting scheme for Android malwares. Our scheme employed different analyzing methods (static and dynamic methods) to construct a flexible detecting scheme. In this paper, we proposed some detecting techniques such as Com+ feature based on traditional Permission and API call features to improve the performance of static detection. The collapsing issue of traditional function call graph-based malware detection was also avoided, as we adopted feature selection and clustering method to unify function call graph features of various dimensions into same dimension. In order to verify the performance of our scheme, we built an open-access malware dataset in our experiments. The experimental results showed that the suggested scheme achieved high malware-detecting accuracy, and the scheme could be used to establish Android malware-detecting cloud services, which can automatically adopt high-efficiency analyzing methods according to the properties of the Android applications.

1. Introduction

According to IDC Mobility Research [1], the total shipments of smartphones and tablet computers had reached 1.84 billion in 2016. Among these devices, 81.2% of them are running on Android OS. Due to the openness of Android OS, Android-based devices are the major target of malwares. According to the investigations of third parties, the number of discovered Android malwares increased around 750,000 in Q1/2017, and about 3.5 million new malwares may be shared by various malware providers by the end of 2017 [2]. Meanwhile, many rogue companies keep developing Android Applications (Apps) that can perform malicious actions such as stealing credit card information [3]. Therefore, the detection of Android malwares has been a critical task in mobile security for both the consumers and service providers.

Given this serious threat, the Google Play App Store performs security scans on every submitted App. Unfortunately, this safety mechanism is not as effective as expected, and the market has not blocked the malware delivering. Besides, many third-party App markets without any supervision also provide conveniences for attackers to spread their malwares. Consequently, the Android malwares are constantly evolving.

As the Android Apps are always shared and downloaded with cloud service providers, cloud-based Android malware detection will be used by cloud management to improve the service qualities. A typical malware detection cloud service is shown in Figure 1. The cloud can work as file-sharing services which use software tools and machine-learning techniques to detect the Android malwares. The uploading and delivering process of Apps can be realized with common frameworks of file-sharing cloud, and the main issue of the malware-detecting cloud is how to complete accurate malware detection with high efficiency.

Constructing comprehensive and accurate Android malware detecting solutions is a hard and long-term work. In this paper, we suggest a hybrid malware-detecting scheme to improve the traditional malware-detecting methods. Some new techniques are proposed in our scheme, for example, we adopted clustering methods to unify the dimension of calling graph features into 100, so that the dimension collapsing of calling graph-based detecting analysis can be avoided, we proposed a new analyzing feature which is named as Com+ for

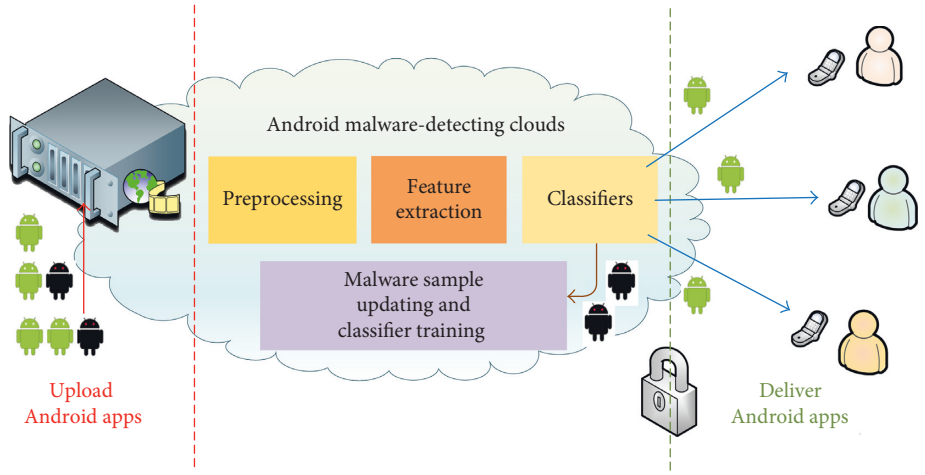

Figure 1: Common framework of Android malware-detecting cloud.

Apps without obfuscation strategy in static analysis, and we also designed soft tools ("*Detector*" detailed in Table 1) to improve the efficiency of the detection. With integrating and choosing proper detecting methods, our malware-detecting scheme is more comprehensive and efficient than other traditional detecting methods.

2. Relative Works

Plenty of researches have been carried out on the detection of malwares. Some of the research paid attention to the analysis of the evolution of malware ecosystem. Concerning the working mechanism and behaviors of malwares, Felt et al. [4] made a comprehensive survey on malwares of the three main platforms (Android, iOS, and Symbian) and indicated the potential bugs in the temporal counting mechanism of those systems. Di Cerbo et al. [5] suggested to classify mobile malwares as four types (viruses, botnets, worms, and Trojan horses) and further summarized six main malware behaviors such as stealing user information, intercepting/sending SMS, etc. For the security issues, factors of Android security were summarized and analyzed by Faruki et al. [6], and Sadeghi et al. [7] suggested a solution to evaluate the security of Android Apps.

For the techniques of Android malware detection, many methods have been proposed. Depending on whether the Apps need be executed in detection, the malware-detecting methods can be classified into two types: static analysis and dynamic analysis. In static analysis, researchers usually decompile the App and extract some particular features without executing them. For example, since the App Permission can reflect the App's access to the software and hardware resources of the mobile devices, Wang et al. [8] used Permission requests as the features and ranked the risk degree of samples from the malware library based on the requesting rates of Permission. Besides, API calls (officially defined packages and classes of Android OS) could be chosen as another feature for static analysis. Using API calls as features, Nishimoto et al. [9] recorded all the invocations in the Apps by inserting Log.v method into API calls. Using API calls as the features, Chan and Song [10] selected 19 typical API calls and achieved good malware-detecting results. Aafer et al. [11] proposed API feature with tuning machine learning model to classify Apps as normal (benign) or abnormal (malicious), and the performance of this solution could be improved by constant updating of malware samples. Except using Permission and API calls as features for malware detection, some researchers also used function-call relationships in static malware analysis. Wei et al. [12] used function-relation graphs to measure the structural similarity of Apps and used *DroidExec* to recognize malwares. Gascon et al. [13] generated the embedded function call graph and proved the effectiveness of this method in malware detection. In our previous work [14], we also suggested a method for static analysis based on the API calls.

Table 1: Analyzing software tools.

Name	Function	Platform
Apk tool	Most popular decompiling tool for Android Apps	Windows
Androguard	A powerful and comprehensive integrated analyzing and operating tool based on python	Linux
Monkey	Generates pseudorandom streams of user events such as clicks and touches	All
Strace	A system call tracer which can print out all the Linux kernel system calls of Android	All
TaintDroid	Modified Android OS which can monitor private information by attaching tags	Linux
DroidBox	Dynamic analysis tool for Android Apps working on emulator	Linux
Detector	A self-designed soft tool to find newly executed App and trace and log the operations of Apps	Linux

Dynamic analysis involves executing the Apps in a controlled and isolated environment so that the operations of the Apps can be traced. Many schemes based on different dynamic analysis tools have been proposed. Burguera et al. [15] designed the Crowdroid in 2011, which used the *Strace* tool to collect App's system call logs and send them to the cloud server for further analysis. Ham et al. [16] analyzed App's system call patterns and established a pattern library of benign and malicious Apps, so that malwares could be recognized by calculating the matching patterns. Zou et al. [17] used shared library injection and hooking techniques to intercept corresponding libc function calls. Besides, taint tracking was usually used as another way of dynamic analysis. In 2010, Enck et al. [18] designed and built the *TaintDroid*, which was a customized Android OS that could attach tags on the user privacy information, and these tags could be recognized by the system when the privacy information was leaked.

The abovementioned detecting techniques can be deemed as traditional malware-detecting methods which are mainly dependent on analyzing of the App data. Recently, subjective information such as user comments and ranking was used by cloud processing to improve the performance of the malware detection. Rahman et al. [19] presented a system named as FairPlay, which could detect malware based on the analysis of review activities and search rank fraud; Narayanan et al. [20] suggested a malware-detecting scheme using the security-sensitive information of Apps. Compared with traditional detecting methods, these solutions could achieve good performance in malware detection. But, they always demand time-consuming data collection and complex computation. Thus, improvement of traditional detecting methods is still desired by practical applications.

As the Android platform develops rapidly, the types and patterns of malwares are constantly evolving to avoid detections. Plenty of the newly generated malwares can invalidate existing malware-detecting methods. Thus, malware-detecting techniques should be constantly improving with the evolution of malwares, and an up-to-date malware dataset should also be maintained to evaluate the performance of the detecting techniques.

A new Android malware-detecting scheme is proposed by this paper. Compared with some existing traditional methods which only apply unified processing for each App (e.g., [8] only can be used for Apps without obfuscation strategy and [12, 13] only use complex call graph data for analysis), our scheme could provide more comprehensive analysis for any Android Apps. Besides, as our scheme provide some new detecting techniques and different processing methods are chosen according to the properties of each App, the analyzing efficiency is also enhanced. The rest of this paper is organized as follows: In Section 3, we introduce our malware-detecting scheme and corresponding analyzing tools. Section 4 presents the methods of feature extraction and malware classification. Section 5 presents the malware dataset and corresponding experimental results, Section 6 discusses the limitation and future works of the proposed scheme, and Section 7 concludes the paper.

3. The Proposed Android Malware-Detection Scheme

As discussed in the previous section, malware detection can be classified as static or dynamic analysis. As executing the Apps is not required by static analysis, this approach is simpler than dynamic analysis. However, if the App cannot be decompiled, static analysis becomes ineffective because the features such as Permission cannot be extracted from the Apk file (execution file of Android Apps). In this case, dynamic analysis should be used by executing the App in a sandbox. Dynamic analysis can record log files for some operations of the App running in the sandbox, and the data of log files can be used as features for malware detection. Though dynamic analysis outperforms static analysis in terms of applicability, it also has drawbacks such as computational complexity and low efficiency. Thus, the malware detection should choose proper analyzing methods based on the properties of Apps to enhance the detecting efficiency.

In this paper, we propose a hybrid-detecting scheme for Android malwares that employs several static/dynamic analysis methods to complete malware detection. Our hybrid scheme can choose the appropriate method according to the properties of the App so that a good balance can be achieved between the detecting accuracy and complexity. Figure 2 shows the overview process of our malware-detecting scheme which employs many software tools, and Table 1 summarizes those tools with descriptions. As shown in Figure 2, our scheme extracts the features of the malware classifiers via four paths with different complexities, and the classifiers such as k-NN are used to judge whether the App is malicious. As the detecting accuracy is highly related to the effectiveness of the features, our scheme employs four types of methods (denoted by color and index) to derive the features according to the attributes of the Apps.

Table 2 lists the application scopes and descriptions of the four paths. The four paths are constructed by two static (Path A and Path B) and two dynamic (Path C and Path D) detecting methods. For App which does not adopt obfuscation strategy (such as malwares of [21] dataset which are delivered before 2012), our Com+ feature (new feature proposed by our scheme, detailed in following sections) can efficiently reflect the behaviors of the App, and Path A in Figure 2 should be selected for feature extraction as static analysis. In Path A, the analyzing complexity is very low as the Com+ features can be extracted directly from the Apk files (source file of Apps). With the updating of Android OS, many Apps adopt obfuscation strategies to avoid clearly declaring Com+ feature. For Apps with obfuscation strategy, Path B which adopts function call graph as features should be used. As the complexity of analyzing call graph may be very high, we also proposed clustering methods (detailed in following sections) to decrease the complexity and avoid the analyzing collapsing caused by the unlimited feature dimensions.

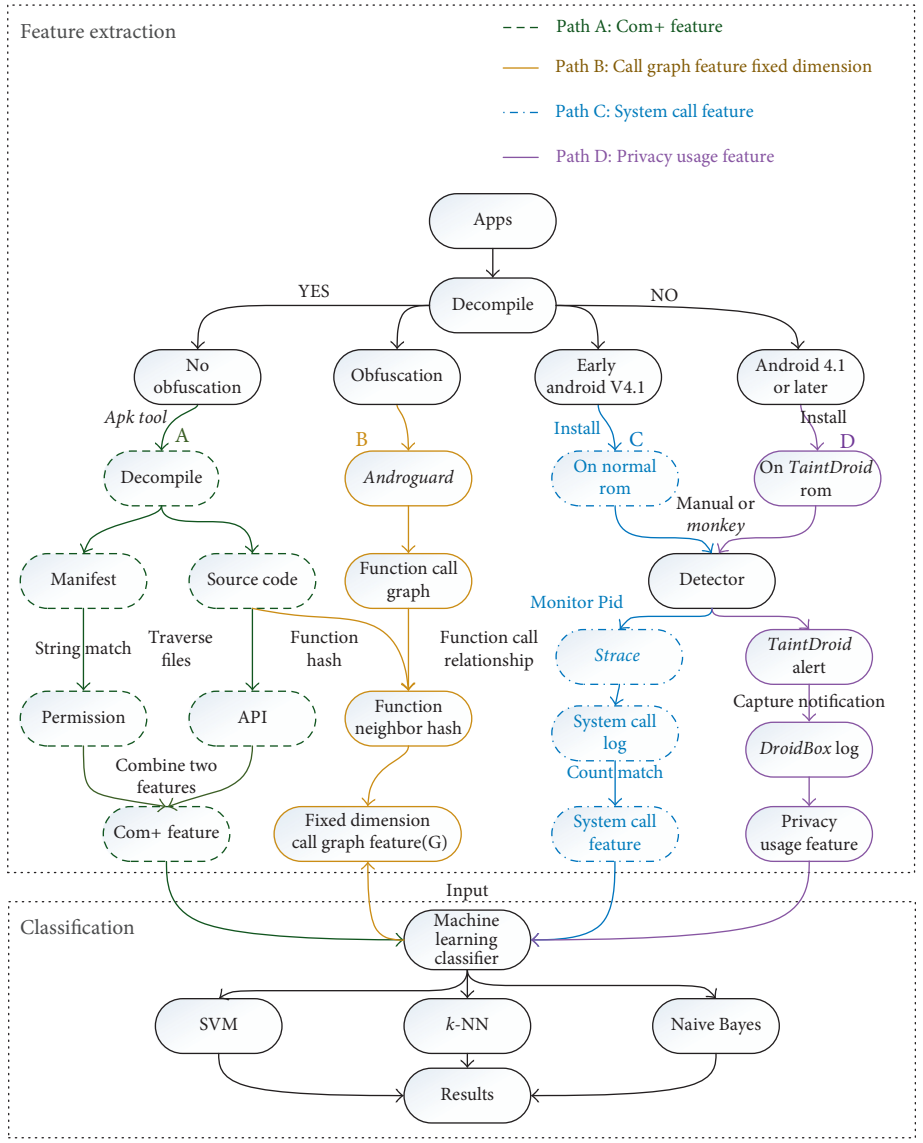

Figure 2: The proposed hybrid-detecting scheme.

Using Path A and Path B, malwares which could be decompiled can be analyzed with low complexity. However, Apps may use encryption techniques which prevent the App from being decompiled. If the malwares are encrypted, dynamic analysis (Path C and Path D) should be applied in malware detection. In order to monitor the behaviors of Apps, a soft tool named as *DroidBox* which can be used for Apps delivered with Android 4.1 or later is adopted by our scheme to extract running features of Apps. For Apps delivered before Android 4.1, Linux soft tool *Strace* should be adopted to extract features, because any Android versions are running on Linux kernel. Compared with static analysis, though dynamic analysis can complete malware detection for any Apps, it also demands more complex processing. Thus, the malware-detecting scheme should complete the analysis with flexible methods to obtain high analyzing efficiency.

Using the four paths of Figure 2, our malware-detecting scheme can complete the malware detection of any Apps. Thus, our scheme is more comprehensive than methods such as in [8–10, 12, 13], which can be used under some special conditions. Besides, our scheme is more efficient than that in [11, 15–20, 22], as our scheme can choose proper processing path according to the properties of each App. Generally speaking, our scheme can obtain good balance between the detecting accuracy and computation complexity, and it is a good solution for cloud-based Android malware-detecting services.

4. Features Extraction and Classification Methods of the Proposed Scheme

As shown in Figure 2, our scheme adopts four paths which are the Com+ feature (Path A), the function call graph feature (Path B), the system call feature (Path C), and the privacy usage feature (Path D) to extract features of the Apps. In the classification processes, *k*-NN, SVM, and Naive

Table 2: Four paths of feature extraction in our malware-detecting scheme.

Path	Description	Complexity	Conditions
Com+ feature	A classical presentation of Apk file	Low	Decompilable, nonobfuscation
Function call graph	A structural-level feature containing the invoking relationships	Super high	Decompilable
System call feature	A Linux kernel level dynamic feature suited for most Android versions	High	Apps earlier than Android 4.1
Privacy usage	An Android-specified dynamic feature indicating privacy leakage based on TaintDroid	High	Apps of Android 4.1 or later

Input: $S=\{s_i\}$, the set of all experimental n Apps.
Output: $C=\{c_{i,j}\}$, the feature set of experimental Apps, the ith row vector present the feature of the ith App.
(1) **begin procedure**
(2) **for** $i = 1$ to n **do**
(3) decompile the s_i
(4) generate the Permission feature p(151-bit) from Manifest;
(5) generate the API feature a(3262-bit) from source codes;
(6) **end for**
(7) **for** $j = 1$ to 151 **do**
(8) $c_{i,j}=a_j$;
(9) **end for**
(10) **for** $j = 152$ to 3413 **do**
(11) $c_{i,j}=p_j$;
(12) **end for**
(13) **end procedure**

Algorithm 1: Com+ feature extraction.

Bayes are used as classifiers. The details of the processing are presented below.

4.1. Feature Extraction

4.1.1. The Com+ Feature. Android OS has its own security approach based on Permission mechanism. For example, Android 4.1 adopts totally 151 kinds of Permissions which allow Apps to access software and hardware resources of the devices [22]. Normal Apps can retrieve all the messages using READ_SMS Permission while malwares may misuse this Permission to perform malicious actions. Therefore, some Permissions can be used as features for malware detection.

Each Apk file has a Manifest.xml file, which declares all the Permissions that the App needs to use. So we can extract Permissions from that file and generate corresponding Permission features. Each App can be represented by a binary vector, namely, P, where $P_i = 1$ if the ith Permission is used and $P_i = 0$ if the ith Permission is not declared in the Manifest.xml file. Thus, any App can be represented by a 151-dimension vector which is named as Permission feature.

However, using Permissions as features may encounter errors as some Permissions are not really employed during the execution of the App (i.e., fake Permission declaration). Thus, our scheme also employs API calls as a feature of static analysis to decrease the impact of fake Permission declaration.

Android OS provides the framework API that can be invoked by Apps so that Apps can interact with the kernel of Android OS. The official framework API consists of a set of packages and classes such as Android/app/Notification/Action and Android/view/Window Manage, which conveys substantial semantics about the App's behaviors [11]. The framework API provides 3262 kinds of API calls in total. If one App adopts an API call in the code, it means this App will execute the action of the corresponding API.

Since API calls reliably reflect the behaviors of the App, we can combine API calls together with Permissions to form a more representative and comprehensive feature, which is called the Com+ feature in this paper. The procedure of the Com+ feature extraction for one App is summarized in Algorithm 1. Using this procedure of Path A, a 3413-bit (151 Permissions features + 3262 API call features) Com+ feature can be formed for each App in our malware-detecting scheme with very low complexity.

4.1.2. Function Call Graph Feature with Fixed Dimension. With the development of malicious technology, obfuscation strategies have been widely used by malwares which can obscure the Permission and API call features. The detecting methods used in virus detection of personal computers show that structural-level properties, such as function call graphs, can offer a robust representation of virus actions [23]. Thus, we believe that function call graphs may also be

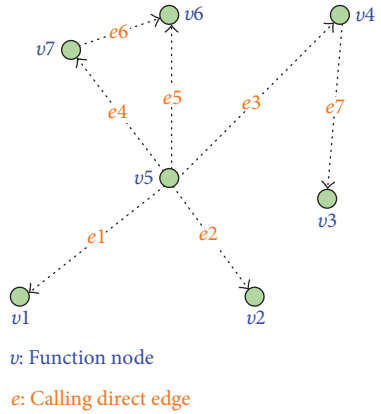

Figure 3: An example of the function call graph.

effective in Android malware detection even when obfuscation strategies are used by malwares. Using some software tools, the function call graph feature with limited dimension can be extracted from the Apps in our malware detecting scheme (Path B of Figure 2).

(a) *Function call graph information extraction*: We can get a ".gexf" file from the Apk using the "*Androguard*" tool of Table 1. This directed graph contains nodes of every function and edge direction from callers to callees. Then, the function call graph can be represented as a 4-tuple $G = \{V, E, L, l\}$, where V is a finite set of nodes, and each node is associated with one of the App functions. $E \subseteq V \times V$ denotes the set of calling directed edges, where an edge from a node v_1 to a node v_2 indicates a call from the function represented by v_1 to the function represented by v_2. As shown in Figure 3, the structural-level properties of the function call graphs can be expressed by labeling V and E.

Further, labeling function $l : V \to L$ can be used to label each node with the properties of the function. According to Dalvik [10] specification of Android OS, we chose 15 distinct categories of instructions based on their functionalities as shown in Table 3, and the l can be defined as

$$l(v) = \{b_1(v), b_2(v), b_3(v), \ldots, b_m(v)\}, m = 15. \quad (1)$$

If node v contains the ith instruction in Table 3, b_i is set to 1; otherwise b_i is set to 0. Then, each function node of Apps can be labeled by a 15-bit vector.

However, only considering single function is not enough for malware detection. So, we further explore the invoking relations among all the functions. Reference [24] suggested a kernel operating on an enumerable set of subgraphs in a labeled graph. Each given node v and the set of neighbor nodes V_v contain its own subgraph. The generation of the subgraph is defined by the operation as

$$F = r(l(v)), \left(\bigoplus_{z \forall V_v} l(z) \right), \quad (2)$$

where $\oplus$ represents a bit-wise XOR on the binary labels and r denotes a 15-bit left-shifting operation. Using (2), we can generate a new 30-bit feature F which contains the subgraph features, and the F could represent both the content and invoking information of a function node.

Table 3: List of instruction categories and corresponding bit.

Category	Bit
nop	1
move	2
return	3
monitor	4
test	5
new	6
throw	7
jump	8
branch	9
arrayop	10
instanceop	11
staticop	12
invoke	13
unop	14
binop	15

(b) *Unify call graph features into fixed dimension*: For each App, we can extract n nodes and corresponding F features. As the number of subgraphs (n) is not certain, different Apps contain unequal amount of F features. If we directly use the $n \times F$ data as features in the machine learning process, the analysis may be collapsed by the high dimension of $n \times F$. As many tranditional softwares are transferring from the PC to Android devices, Android Apps are providing more and more functions. For example, if one App contains 10,000 functions, the dimension of $n \times F$ will reach 300,000. As huge computation will be demanded in the machine learning process which uses the data of $n \times F$ as features, the malware-detecting service may be collapsed. Hence, we need to unify the dimension of these features to meet the requirement of the classification algorithms.

In our scheme, we adopt Laplacian score feature selecting method of [25] to select 100 typical F features as the centers of k-means clustering. Then, k-means which can group objects into k classes according to the measured intrinsic characteristics is used to complete the feature unification [26]. Using the k-means clustering method, our scheme can convert the $n \times F$ features into a fixed 100-dimension feature for classification, and the analysis collapsing caused by the high dimension of function call graph can be avoided. In this paper, we named the new call graph feature with fixed dimension as G features in this paper. Figure 4 presents the processing of $n \times F$ features $\to G$ features. For an App which contained n nodes, $n \times F$ features could be extracted from this App using labeling and hashing methods presented before. Then, these $n \times F$ features are clustered by the 100 centering F features which are generated by feature selection, and the number of F which are clustered as F_i is recorded as g_i. Further, feature G could be constructed by g_i as follows:

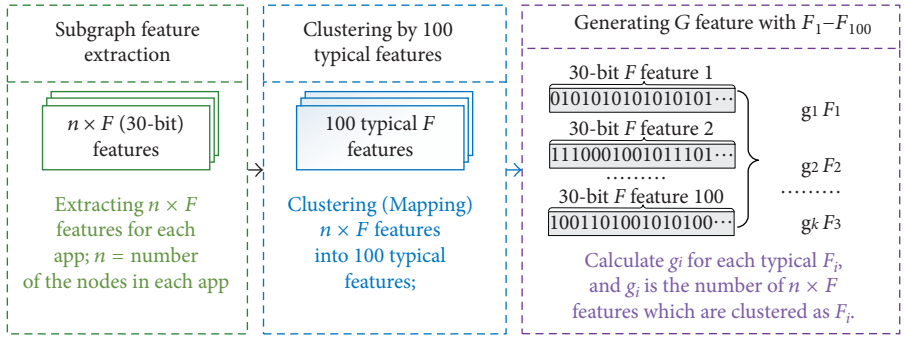

Figure 4: Unification of $n \times F$ features into G feature.

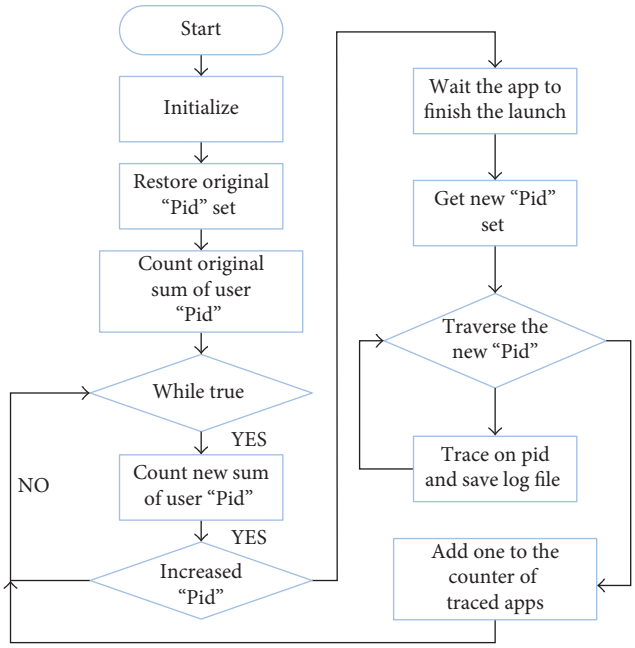

Figure 5: Flow of the *Strace*-based analysis.

$$G = \{g_1, g_2, \ldots, g_k\}, s.t. \sum_{i=1}^{k} g_i = n, k = 100. \quad (3)$$

4.1.3. Feature Extraction with Detector in Dynamic Analysis. If Apps use encrypting techniques, these Apps could not be decompiled, and dynamic analysis should be applied in the malware detection. In our malware-detecting scheme, we designed a special software tool *Detector* to monitor the running Apps' status and the newly started testing Apps. Besides, our *Detector* also collects and formats the log data provided by tracing soft tools, so that the dynamic features can be sent to the machine learning process of our scheme (descriptions about the *Detector* are listed in Table 1). In our scheme, Apps which prepared for malware detection are run with interval of 30 seconds, and the *Detector* can create feature files (log files) from the tracing data of *Strace* or *TaintDroid*.

The processing flow diagram of the *Detector* is shown in Figure 5. The initializing work includes setting root path and some counters. The original "Pid" [27] setting of all running user Apps is restored. The amount of "Pid" is monitored and counted in every iteration of the Apps. If the amount of "Pid" is increasing, *Detector* will deem that the testing App is turned on. Then, recording actions will be executed to trace the testing Apps and generate log files. If Apps start several "Pid," our Detector will execute multiple instances to save all the log data for each "Pid."

4.1.4. Extracting Privacy Usage Feature with DroidBox and TaintDroid. DroidBox [28] is a dynamic analysis tool which can monitor the App's privacy usage such as network connection, reading/writing operations, information leaks, SMS messages, and phone calls. To our knowledge, the *DroidBox* obtained good performance for Android 4.1 or later. As the malwares always complete some actions related with privacy usage, for Apps which are delivered with

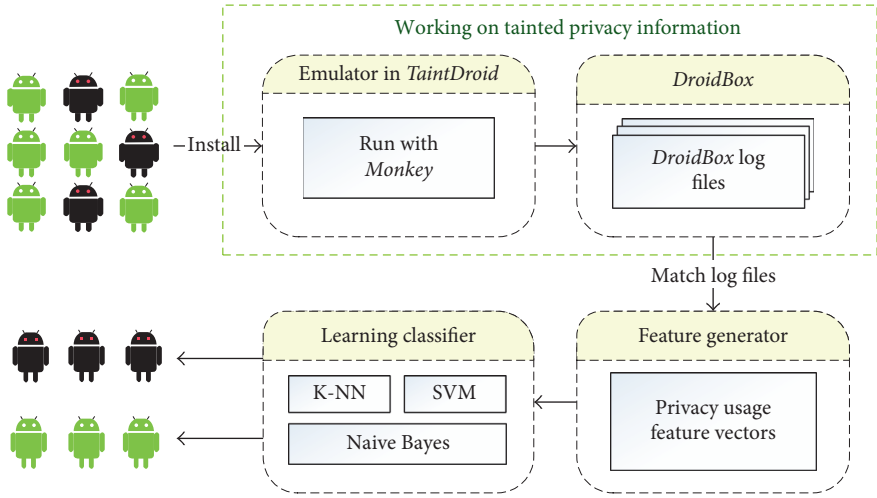

Figure 6: The framework of *DroidBox* analysis.

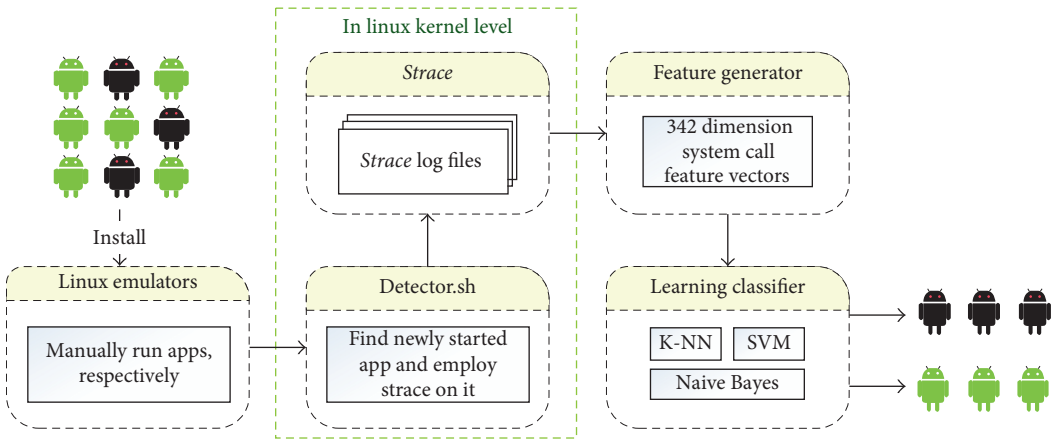

Figure 7: The framework of the *Strace* analysis.

Android 4.1 or later, *DroidBox* and *TaintDroid* in Table 1 should be used to trace the behaviors of Apps. Figure 6 shows how the *DroidBox* (Path D in Figure 2) path extracts the privacy usage features. It is achieved by shell scripts and python programs working together on the Linux platform. The Linux emulator is running a *TaintDroid* rom which will clean all user data when it is restarted. The *DroidBox* tool will automatically install/start the Apps on the emulator, and the *Monkey* tool [29] in Table 1 is used to complete initial operations. During the execution of Apps, our scheme can catch all the Apps' privacy usage status and generate the log files with *Detector*.

4.1.5. Extracting System Call Feature with Strace. For Apps delivered before Android 4.1, the *DroidBox* and *TaintDroid* cannot be applied for the App behavior tracing. As Android OS is built on the Linux kernel, our scheme employed *Strace* to trace the behavior of the Apps. The process is illustrated in Figure 7. After the Apps are uploaded, we install and execute all the Apps on the Linux emulating environment. Then, *Strace* (Path C in Figure 2), which is a system call tracer and debugging tool, is used to trace the Linux kernel system calls made by processes or programs. After the system call information is extracted by *Strace*, the feature log files are generated with our *Detector*.

Using the Path A–Path D, our malware-detecting scheme can extract features from any Android Apps. According to the processing of the four Paths, the complexities of feature extraction are different. Normally, Path A demands the lowest computation, Path B and Path D require similar complexity, and Path C always demands the highest computation. As our scheme can automatically deliver the Apps to the four Paths according to the properties of Apps, we can achieve good performance on the analyzing complexity. Compared with some existing malware-detecting methods, our scheme is more comprehensive and efficient, and Table 4 lists the comparisons.

4.2. Classification Methods. After the feature extraction, our scheme analyzes different features by three classification

Table 4: Comparisons of our scheme with some existing malware-detecting methods.

Methods	Required using conditions	Efficiency
References [8–10]	Apps can be decompiled, nonobfuscation	Low
References [12, 23]	Apps can be decompiled, obfuscation strategy	High
References [10, 18]	Apps cannot be decompiled, Android 4.1 or later	High
References [16, 17]	Apps cannot be decompiled, before Android 4.1	High
Ours	None	Median

methods: k-NN, SVM, and Naive Bayes. We believe that different classifiers may obtain various results, and the results will be impacted by parameter tuning of different classifiers. In this paper, we only adopt three classifiers which are common in machine learning process. The k-Nearest Neighbors algorithm (k-NN) is a nonparametric method used for classification [30]. In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, so that it will be assigned to the most common class among its k nearest neighbors (k is a positive odd integer). If $k = 1$, the object is simply assigned to the nearest neighbor class. As the malware detection only need to divide the malwares from benigns, we set k as 1 to construct 2-type k-NN classifier.

SVM can divide the n-dimensional space representation of the data into different regions by using a hyperplane, which intends to maximize the margin between the two classes. The margin is computed using distances between the closest instances of both classes, which are called support vectors [31]. In our malware-detecting scheme, SVM with RBF kernel is also used in our malware classification process.

Naive Bayes classifier is a kind of simple probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Concerning the variety and independence of different Apps, Naive Bayes can also be used to classify the malicious and benign Apps.

The above three classified methods are adopted in our experiments without parameter tuning. As the performances of the three classifiers are similar, we believe the features extracted by our detecting scheme are effective.

5. Experimental Results and Analysis

5.1. Testing Malware Datasets. In order to evaluate the performance of the proposed malware detecting scheme, two datasets were applied in our experiments. The first dataset was an open-access dataset which was built by Jiang [21] in 2012. This dataset collected 1260 Apps from August 2010 to October 2011, and these samples were classified into 46 Android malware families. To our knowledge, the dataset of [21] was the most popular and open-access malware library so far, and lots of research used this library for evaluation. Thus, we randomly selected 1000 Apps from malware and benign libraries to form a dataset before 2012, which was referred to as the older dataset in our experiments.

Table 5: Information and comparison of experimental datasets.

Type	Older dataset		Newer dataset	
	Malicious	Benign	Malicious	Benign
Source	Reference [21]	Wandoujia	VirusShare	Wandoujia
Time	2011	2016	2016	2016
Amount	1000	1000	1000	1000

As the Android platform is updating rapidly, the security strategies of Android is also changing. So, old malicious actions may not be active any more, and new malwares may perform different malicious functions. Considering this, we established an up-to-date dataset which provided open-access in [25], and the details malicious actions of the malwares were also shared by excel format. The malicious Apps of our new dataset were collected from VirusShare [32], one of the major virus App-sharing websites. Because some Apps of VirusShare was ineffective, we spent several months to pick the appropriate malicious samples from top-ranked malwares. Eventually, about 1000 malicious Apps delivered between September 2012 and December 2016 were selected as testing dataset. Benign Apps of our new dataset were gathered from an App market called "Wandoujia" [33] in July 2016. We obtained more than 1000 benign Apps from the top popular ranking Apps, so that these samples could cover most types of Android Apps. We believed that top-ranked popular Apps would not perform any illegal activities, otherwise the management of Apps market and the App users would not rank the Apps as top popular applications. The diversity of the benign Apps could ensure the comprehensiveness of the behaviors and features of benign sample set. Using these up-to-date malicious and benign Apps, we formed a new testing dataset which was referred as newer dataset of Table 5.

5.2. Evaluation Matrices. The performance is evaluated by three metrics: accuracy (ACC), true positive rate (TPR), and false positive rate (FPR). These metrics are defined as

$$\begin{aligned} \mathrm{TPR} &= \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \\ \mathrm{FPR} &= \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}, \\ \mathrm{ACC} &= \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}}. \end{aligned} \tag{4}$$

TP (true positive) represents the number of malwares that are correctly detected. FN (false negative) denotes the number of Android malwares that are not detected (predicted as benign App). TN (true negative) is the number of benign Apps that are correctly classified, and FP (false positive) is the number of benign Apps that are incorrectly detected as Android malwares. The TPR can reflect the percentage of malwares in the whole samples which are detected as benign Apps, the FPR can reflect the percentage

Table 6: Results of static analysis with different feature sets on older dataset.

Algorithm	Permission feature			API feature			Permission and API		
	ACC	TPR	FPR	ACC	TPR	FPR	ACC	TPR	FPR
k-NN	95.76	97.31	5.21	98.42	97.30	0.43	98.61	97.21	0.14
SVM	96.65	95.31	2.65	99.10	98.51	0.34	99.21	98.38	0.14
Bayes	93.37	94.62	7.31	94.13	99.57	11.01	94.41	99.42	10.7

of benign Apps in the whole samples which are detected as malware, and ACC can evaluate the overall detecting performance. Details about these evaluating metrics could be found at [34].

For each classification method, we randomly selected the training and testing dataset at the ratio of 7 : 3. The malicious and benign features are labeled as 1 and 0, respectively. Then, we performed the classifying algorithm 100 times and calculated the average value of each measurement metrics as the final results, so that the reliability of the experiments is guaranteed.

5.3. Experiment Results and Discussion

5.3.1. Older Dataset. For the older dataset, we used both the Com+ feature for static analysis and the system call feature for dynamic analysis (Path A and Path C in Figure 2) to execute the detection [14].

Table 6 summarizes the classification results of the different static features, where the Com+ feature achieves a good result. The accuracy ranges from 93% to 99%. The best performance in ACC (99.28%) is achieved by the SVM classifier using the Com+ feature. Meanwhile, this algorithm has the second highest TPR (98.68%) and the lowest FPR (0.12%). Therefore, the SVM algorithm is a suitable method for the Com+ feature of static detection.

For Apps which cannot be decompiled, we used *Strace* to dynamically extract their system call features. As shown in the experimental results in Table 7, all three classifiers achieve 85% or higher in accuracy. And the Naive Bayes has the highest ACC rate (90.00%) and TPR rate (90.85%), as well as the lowest FPR (10.85%). So, we consider Naive Bayes as the good choice for the system call feature of dynamic analysis.

5.3.2. Newer Dataset. Since Android is updated continually and the latest Apps mostly adopt obfuscation on source code, it is difficult to extract the Com+ feature from Apps in the newer dataset. Meanwhile, the performance of the system call feature had also decreased. So, we employed new methods (Path B and Path D in Figure 2) to deal with Apps in the newer dataset.

The function call graph feature (Path B) is used as static analysis for the newer dataset, and Table 8 shows the comparison between different classifiers using the function call graph feature. The accuracy of all these algorithms is higher than 86%, which proves that the function call graph is an excellent feature for static analysis for the newer dataset. The *k*-NN and Naive Bayes classifications produced similar results. SVM obtained the highest TPR but the worst FPR, which means it detected 96.92% of the malwares but misjudged 24.54% of the benign Apps in the meantime.

Table 7: Results of *Strace* analysis on older dataset.

Algorithms	ACC	TPR	FPR
k-NN	87.82	89.51	13.32
SVM	85.41	83.88	11.36
Naive Bayes	90.31	90.41	10.65

As mentioned in Section 3, *Strace* can be used on most types of Linux kernel-based system (like Android), which means it can work on different Android versions. But its weakness is also obvious such as lacking of pertinence. In contrast, *DroidBox* (Path D), which is a software tool designed especially for Android version 4.1 or higher, can provide more specific analysis on privacy information leaking according to the Android mechanism.

The experimental results of *DroidBox* analysis on new dataset are shown in Table 9. Among the three different classifiers, the best accuracy (74%) of all these algorithms is achieved by the *k*-NN classifier, and the *k*-NN classifier also obtains the highest TPR and the lowest FPR. Therefore, it is the best classifier for the privacy leaking of dynamic analysis. However, as *DroidBox* employs an emulator which is not actual Android devices, the *Monkey* tool can only simulate parts of the human operations. The performance of *DroidBox* may be further improved in our future work.

5.3.3. Discussion. As shown by the experiments described above, our scheme provides four paths to complete malware detection, namely, the Com+ feature, function call graph with limited dimension, system call feature, and privacy usage feature. Given any Android App, our scheme can select the most suitable path according to its properties in order to achieve comprehensive and efficient analysis.

Many existing Android malware-detecting methods (such as [8–10, 19, 20]) were evaluated by their testing dataset which are not shared. Thus, it was very hard to complete fair comparison between different malware-detecting methods. To our knowledge, dataset of [21] are open-access and popular malware dataset, and we used this dataset to evaluate our scheme. For the new dataset, as other existing methods have not provided open-access detecting software, it is very hard for us to complete the experiments with other up-to-date approaches.

Table 8: Results of function call graph analysis on newer dataset.

Algorithms	ACC	TPR	FPR
k-NN	86.56	86.48	13.51
SVM	86.11	96.32	24.65
Naive Bayes	86.51	84.54	11.34

Table 9: Results of *DroidBox* analysis on newer dataset.

Algorithms	ACC	TPR	FPR
k-NN	74.23	77.77	28.41
SVM	66.33	73.87	41.74
Naive Bayes	71.54	73.31	30.12

We also checked the malware samples which were detected by our scheme (detecting errors). We found that most detecting errors occurred in the game Apps with malicious action of private information leaking. We believed that the reason of the wrong detecting may be caused by the complex operation of game Apps, as these Apps always require operations such as registering private user information and payment for game tools. Some illegal activities could be covered by the processing of purchasing tools of game (e.g.,, credit card information leaking). Thus, the illegal activities were hard to be detected.

As shown in Tables 6–8, the proposed Android malware-detecting scheme obtained good detecting accuracy. Besides, the average malware-detecting complexities are decreased by our flexible feature extracting Paths. Thus, our scheme achieved good balance between the malware detecting accuracy and complexity, and it is highly suitable for cloud service providers which need to improve the security of their cloud services.

6. Limitation and Future Work

The detecting scheme proposed in this paper could be applied to construct a basic malware-detecting platform. However, the scheme is limited in some aspects, and some issues should be further improved to make the detecting scheme more comprehensive.

(i) Like many traditional malware detecting methods, the proposed scheme only can complete objective detection with software. However, some subjective methods which involve reviewing comments of App users have been used in malware detection, such as [19, 20]. With the developments of deep learning, effective user feedbacks may be extracted from huge number of comments, and we believe these feedbacks are very valuable for malware detection. Thus, our future malware detecting scheme should integrate the user's feedbacks as some features in the detecting process to achieve better detecting accuracy.

(ii) The proposed detecting scheme still demands some manual processing; thus, the scheme is hard to be applied in practical applications which demand real-time processing of huge number of App samples. As many Apps are delivered every day, more automatic processing should be developed to upgrade these manual processing, so that the proposed scheme could be adopted more easier in practical applications.

(iii) Our scheme still has not adopted self-updating mechanisms to constantly renew the testing dataset. The testing dataset used in our experiments are constructed by ourselves, and the samples only cover parts of the Android Apps. Meanwhile, the Android OS is continuously updating, and the malwares are also evolving. Thus, some auto-updating methods should be employed in our scheme to complete auto-updating of testing dataset and auto-training of classifiers.

(iv) The current detecting scheme only used some common Linux or Android soft tools to analyze the App samples. If malwares adopt some techniques to cheat these common soft tools, the detecting scheme may be invalided. Thus, some new tracing and analyzing soft tools should be integrated into our scheme.

7. Conclusion

Malware detection is very important for mobile applications. This paper proposed a high-efficiency detecting scheme for Android malwares using static and dynamic analyzing methods, such as Permission, API calls, function call graph with limited dimension, and privacy usages. Several software tools are designed and integrated to extract malware analyzing features including Com+ feature, system call, and privacy leaking information. We also proposed some new techniques such as Com+ feature and "*Detector*" to improve the performance of malware detection. Using these techniques, the proposed scheme can effectively detect malwares, regardless whether the encrypting or obscuring techniques are used by malwares. Besides, our scheme can achieve good balance between the complexity and detecting accuracy as several methods are applied to complete feature extracting based on the properties of the Apps. Plenty of experiments verified the high performance of the scheme with our new open-access up-to-date malware dataset.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work was supported in part by the National Natural Science Foundation of China (Grant nos. 61373102 and 61771338) and by Yunnan Academician Funding.

References

[1] http://www.idc.com/prodserv/4Pillars/mobility#Press.

[2] https://www.magzter.com/news/488/1242/052017/0jt0j.

[3] "Kaspersky: forget lone hackers, mobile malware is serious business," February 2014.

[4] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 3–14, Swissôtel Chicago, Chicago, IL, USA, October 2011.

[5] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, "Detection of malicious applications on Android OS," in *Proceedings of the 4th International Workshop on Computational Forensics (IWCF)*, pp. 138–149, Springer, Berlin, Heidelberg, Germany, November 2010.

[6] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[7] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, 2017.

[8] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in Android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.

[9] P. P. Chan and W. K. Song, "Static detection of Android malware by using permissions and API calls," in *Proceedings of Security and Privacy in Communication Networks. SecureComm 2013*, vol. 1, pp. 82–87, Sydney, Australia, September 2013.

[10] P. P. Chan and W. K. Song, "Static detection of Android malware by using permissions and API calls," in *Proceedings of International Conference on Machine Learning and Cybernetics*, vol. 1, pp. 82–87, Lanzhou, China, July 2014.

[11] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: mining API-level features for robust malware detection in Android," in *Proceedings of Security and Privacy in Communication Networks. SecureComm 2013*, pp. 86–103, Sydney, Australia, September 2013.

[12] T. E. Wei, H. R. Tyan, A. B. Jeng, H. M. Lee, H. Y. M. Liao, and J. C. Wang, "DroidExec: root exploit malware recognition against wide variability via folding redundant function-relation graph," in *Proceedings of 17th International Conference on Advanced Communication Technology (ICACT)*, pp. 161–169, Seoul, South Korea, July 2015.

[13] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pp. 45–54, Berlin, Germany, November 2013.

[14] Y. Liu, Y. Zhang, H. Li, and X. Chen, "A hybrid malware detecting scheme for mobile Android applications," in *Proceedings of the 2016 IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, USA, pp. 155–156, January 2016.

[15] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for Android," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 15–26, Swissôtel Chicago, Chicago, IL, USA, October 2011.

[16] Y. J. Ham, D. Moon, H. W. Lee, J. D. Lim, and J. N. Kim, "Android mobile application system call event pattern analysis for determination of malicious attack," *International Journal of Security and Its Applications*, vol. 8, no. 1, pp. 231–246, 2014.

[17] S. Zou, J. Zhang, and X. Lin, "An effective behavior-based Android malware detection system," *Security and Communication Networks*, vol. 8, no. 12, pp. 2079–2089, 2015.

[18] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, 29 pages, 2014.

[19] M. Rahman, M. Rahman, B. Carbunar, and D. H. Chau, "Search rank fraud and malware detection in Google Play," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 6, pp. 1329–1342, 2017.

[20] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "Context-aware, adaptive, and scalable Android malware detection through online learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 157–175, 2017.

[21] X. Jiang, *An Evaluation of the Application ("App") Vertification Service in Android 4.2*, December 2012, http://www.malgenomeproject.org/.

[22] N. Peiravian and X. Zhu, "Machine learning for Android malware detection using permission and API calls," in *Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence*, pp. 300–305, Herndon, VA, USA, November 2013.

[23] X. Hu, T. C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of ACM Conference on Computer and Communications Security*, pp. 611–620, Chicago, IL, USA, November 2009.

[24] D. Haussler, "Convolution kernels on discrete structures," Tech. Rep., vol. 646, Department of Computer Science, University of California at Santa Cruz, Santa Cruz, CA, USA, 1999.

[25] X. He, D. Cai, and P. Niyogi, "Laplacian score for feature selection," in *Proceedings of 18th International Conference on Neural Information Processing Systems*, Vancouver, BC, Canada, December 2005.

[26] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010.

[27] https://en.wikipedia.org/wiki/Process_identifier.

[28] https://github.com/pjlantz/droidbox.

[29] http://developer.Android.com/intl/zh-cn/tools/help/monkey.html.

[30] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[31] V. N. Vapnik, *The Nature of Statistical Learning Theory*, Springer Science & Business Media, New York, NY, USA, 2013.

[32] https://virusshare.com.

[33] https://www.wandoujia.com.

[34] https://en.wikipedia.org/wiki/Sensitivity_and_specificity.