Hindawi
Scientific Programming
Volume 2020, Article ID 5190138, 13 pages
https://doi.org/10.1155/2020/5190138

Research Article

Android Malware Detection Using Fine-Grained Features

Xu Jiang,[1] **Baolei Mao**,[2] **Jun Guan**,[1] **and Xingli Huang**[3]

[1]*School of Automation, Northwestern Polytechnical University, Xi'an, China*
[2]*Cooperative Innovation Center of Internet Healthcare, Zhengzhou University, Zhengzhou, China*
[3]*College of Computer Science, Wenzhou University, Wenzhou, China*

Correspondence should be addressed to Baolei Mao; maobaolei524@gmail.com

Received 16 October 2019; Revised 25 November 2019; Accepted 18 December 2019; Published 25 January 2020

Guest Editor: Zhiang Wu

Nowadays, Android applications declare as many permissions as possible to provide more function for the users, which also poses severe security threat to them. Although many Android malware detection methods based on permissions have been developed, they are ineffective when malicious applications declare few dangerous permissions or when the dangerous permissions declared by malicious applications are similar with those declared by benign applications. This limitation is attributed to the use of too few information for classification. We propose a new method named fine-grained dangerous permission (FDP) method for detecting Android malicious applications, which gathers features that better represent the difference between malicious applications and benign applications. Among these features, the fine-grained feature of dangerous permissions applied in components is proposed for the first time. We evaluate 1700 benign applications and 1600 malicious applications and demonstrate that FDP achieves a TP rate of 94.5%. Furthermore, compared with other related detection approaches, FDP can detect more malware families and only requires 15.205 s to analyze one application on average, which demonstrates its applicability for practical implementation.

1. Introduction

Smartphones have become an integral part of our day-to-day life. New data for December 2018 shows that Android remains the most popular mobile operating system, with a worldwide market share of 75.16% [1]. With over one million Android applications in major app stores, applications such as WeChat, TikTok, and mobile banking applications are used in our daily life and continue to play an increasingly important role. Most of these applications have access to users' private information such as their location, credit card, and contact information. Almost all applications access the users' private data, although this provides users with better personalized services [2]. It may also result in information leakage of private data and economic loss [3]. Further, Android malicious applications keep emerging endlessly, and this security issue has gained increasing attention in the industry and academic fields [4].

A large body of research against Android malware has been proposed. Currently, static analysis and dynamic analysis are the two main types of detection methods. Each approach has its merits and shortcomings. The static analysis methods such as Kirin [5], PApriori [6], and DREBIN [7] analyze applications without executing the program requiring low overheads. However, the methods cannot defend against antidecompiling and obfuscation. On the contrary, dynamic analysis methods such as TaintDroid [8] and VetDroid [9] execute the application in real time to detect malware, but it is difficult to acquire all the execution paths. With malware being rapidly evolving, the machine learning method is used to perform Android malware detection. Consequently, gathering features that better represent malicious behavior as the features of machine learning is beneficial to improve the performance of malware detection.

By itself, Android has several security mechanisms in its different layers. The permission mechanism applied in the application layer is an important defence mechanism to protect sensitive resources on the Android platform. Applications must declare dangerous permissions to access sensitive data [10, 11]. Several studies have investigated Android malicious applications based on the declared

permissions, using permission-based methods [5–7, 12–14]. Although these methods avoid high overhead, they consider the declared permissions as the features of machine learning, which cannot truly reflect the difference between benign applications and malicious applications. Thus, they cannot detect malicious applications that declare only a few, or dangerous permissions, which are also always declared by benign applications.

Compared with permissions as features, application programming interfaces (APIs) represent the entire picture of application behavior provided by the Android system [15]. DroidAPIMiner [16] exploits data flow analysis to extract the numbers of APIs used in malicious applications and benign applications to analyze the difference between them, which is similar to these methods in [17, 18]. In general, the API feature set contains a very large number of features, and therefore, detection methods need extra time to extract the API features and to train detection models. It is worth noting that there is a corresponding relation between permissions and APIs.

In this paper, we present FDP, a lightweight Android malware detection method to mine hidden patterns of malware. According to previous studies, there is considerable difference between malicious applications and benign applications in terms of the declared permissions [19–22]. In addition, some dangerous permissions declared for different components reflect the purpose of the developers [23, 24]. It can be used to distinguish different purposes of the same dangerous permission. Therefore, FDP exploits static analysis to collect more fine-grained and representative features from AndroidManifest.xml and decompiled code; these features include fine-grained permissions, intent filter, and the code features of malicious applications. These features can represent the difference between benign applications and malicious applications.

Experiments with 1700 benign applications from Xiaomi markets and 1600 malicious applications demonstrate the effectiveness of FDP, which achieves a TP rate of 94.5% and only requires 15.052 s to analyze an application on average. The main contributions of our work can be summarized as follows:

(1) We propose a new method to perform Android malware detection based on fine-grained permission mechanism, which represents the difference between malicious applications and benign applications as the features of machine learning, including the information of the dangerous permissions used in the components for the first time.

(2) We present a thorough study on the permissions frequently used by malicious applications and evaluate the importance of permission features used to classify malicious applications and benign applications.

(3) In terms of efficiency, FDP uses static methods to gather all features and analyzes an application in a reasonable time.

(4) The experiments demonstrate that FDP is more effective for detecting more malware families and malicious applications that declare few dangerous permissions or dangerous permissions as those declared by benign applications.

The remainder of this paper is organized as follows. Section 2 introduces the features of malicious applications. Section 3 covers the detection framework, including feature selection, feature extraction, and the fine-grained handle on the features. Section 4 introduces the key points for backtracking. Section 5 discusses the experiments and the results. Section 6 describes related work, and we conclude the paper in Section 7.

2. The Features of Malicious Applications

According to Google's definition, permissions are divided into several protection levels: normal, signature, dangerous, and special permissions. Normal permissions have very little risk to the user's privacy. Signature permissions are granted by Android during installation. Dangerous permissions refer to resources that involve the user's private information, which are shown in Table 1. Special permissions do not work like normal and dangerous permissions, which are particularly sensitive; therefore, most applications do not use them [25]. Malicious applications must exploit dangerous permissions to execute malicious behavior. According to the previous work, the research studies about dangerous permissions play an important role for the detection of malicious applications.

Moreover, malicious applications have some code features. AndroMalShare [26] holds 86,798 malicious applications and presents statistical information of the samples, which includes dynamic loading, native code, and reflection, as shown in Figure 1.

The reasons why malicious applications have the code features are provided as follows:

Dynamic loading: Android supports applications to load additional binaries at run time by dynamic loading. In order to evade static detection, malicious application developers separate the core functionality of the application into independent libraries and load them dynamically [27]. Therefore, malicious applications always employ dynamic loading to hide malicious behavior.

Native code: many developers use the native code as part of their applications to improve the execution efficiency of the code and to increase the difficulty of decompilation. Afonso et al. [28] estimate that 37% of all Android applications have the native code. Most malware is restricted to the detection of bytecode; therefore, malware developers use the native code to implement malicious behavior lest the code of the application is reversed, which means that static analysis cannot analyze the applications including the native code completely [29].

Reflection: since Android applications are composed of the Java code and native code, they have the ability to leverage the reflection mechanism. Malicious applications leverage reflection to invoke APIs corresponding

Table 1: Dangerous permissions.

Permission group	Permission
CALENDAR	READ_CALENDAR
	WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS
	WRITE_CONTACTS
	GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION
	ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE
	CALL_PHONE
	READ_CALL_LOG
	WRITE_CALL_LOG
	ADD_VOICEMAIL
	USE_SIP
	PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS
	RECEIVE_SMS
	READ_SMS
	RECEIVE_WAP_PUSH
STORAGE	READ_EXTERNAL_STORAGE
	WRITE_EXTERNAL_STORAGE

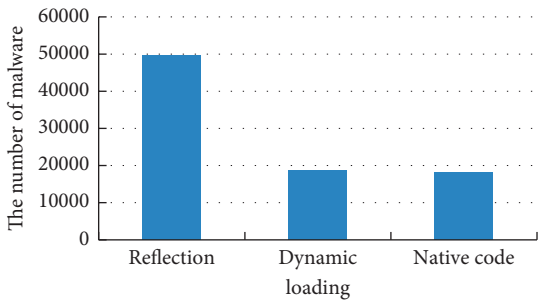

Figure 1: Code feature of malicious applications.

to dangerous permissions to perform malicious behavior, in an attempt to evade static detection [30].

3. Detection Methodology

3.1. Overall Framework. In order to extract features that can better represent application behavior, FDP exploits static analysis to extract key features by decompiling applications and analyzing them using the optimal algorithm of machine learning. The process is illustrated in Figure 2 and the details are provided as follows:

(a) Static analysis: FDP employs the apktool to decompile the application and acquires different feature sets from the AndroidManifest file and the decompiled source code.

(b) Embedding in vector space: all extracted features are mapped to a vector space that can be analyzed as features in machine learning. If the application has the features, the corresponding dimensions are set to 1. Instead, the corresponding dimensions of the other features are set to 0. In order to prevent the problem of overfitting, we take the action of "RemoveDuplicates" to remove the same instances.

(c) Optimal classifier: we use J48 (decision tree algorithm), K-Nearest Neighbor (KNN), Naïve Beyesian (NB), and Support Vector Machines (SVM) to analyze the same training set and choose the classifier with the best indicators as the optimal classifier for FDP.

3.2. Fine-Grained Permissions. The permissions mechanism restricts access to a series of critical APIs [31]. This paper refers to the APIs corresponding to dangerous permissions as sensitive APIs. Several studies choose APIs as the features of machine learning. The classification information is relatively rich, provided that all sensitive APIs are selected as features. However, it is time consuming to process with huge resource consumption and there must exist redundant and meaningless features. Nevertheless, the premise of invoking sensitive APIs is that the applications must declare permissions in the AndroidManifest file. For example, applications must declare the SEND_SMS permission to invoke sendTextMessage() and sendMultipartTextMessage() to send messages. Therefore, FDP selects permissions instead of APIs as the features of machine learning to ensure it is lightweight.

In terms of how to select permissions, it is not suitable to select permissions that are commonly used by both malicious applications and benign applications. If all permissions are used, it will inevitably lead to a "curse of dimensionality." As shown in the previous work [19], malicious applications tend to declare dangerous permissions more often than benign applications. Thus, our research method considers the permissions that are frequently used by malware as the object of our study and extracts the features from the following aspects.

Declared permissions and intent: applications invoking sensitive APIs must declare the corresponding permissions in the AndroidManifest file. We extract these declared permissions after decompiling the application. Activities, Service, and Broadcast Receivers are activated by intent and they register their type of intent using intent-filters in the AndroidManifest file. Malicious applications can exploit the announcement from the Android system to trigger malicious behavior. Therefore, our method extracts the intent that is always used in malicious applications as features, e.g., BATTERY_CHANGE_ACTION and SMS_RECEIVED.

Unused permissions and root privilege: some applications declare dangerous permissions; however, there are no mapped APIs in the Smali code of the application because dangerous permissions are used in dynamic loading or abused by the developers.

Our method first extracts dangerous permissions listed in the AndroidManifest file. Then, we traverse the decompiled source codes based on the relation between the permissions and the APIs provided by PScout [32], to look for declared permissions for which the mapped APIs are not invoked. Such dangerous permissions are defined as unused

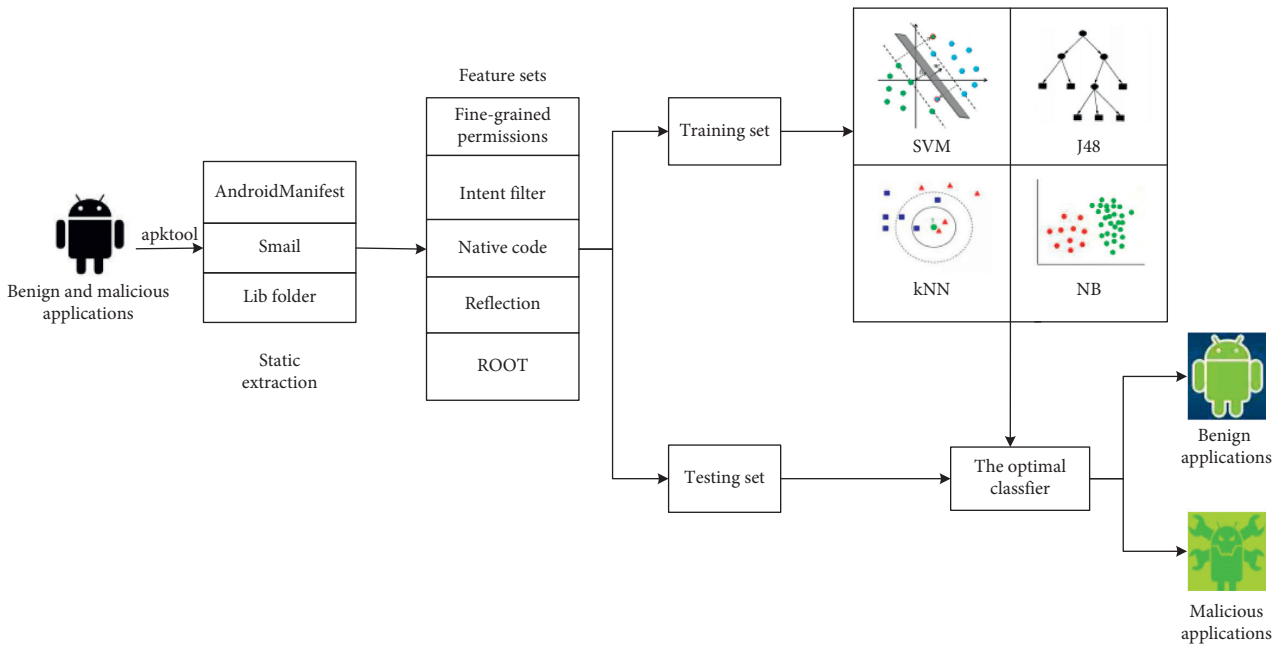

Figure 2: Framework of FDP.

permissions. If there exists sensitive APIs without mapped dangerous permissions in the AndroidManifest file, we suspect that the application is probably attempting to gain root privileges. Because the root privileges give the application absolute control over the device, it can execute any malicious behavior without permissions [33].

Permissions used in components: permissions used in components are the bright spot of our method. In this study, the sensitive APIs are used as the starting point to backtrack the generation of the call graph. According to the relation between the permissions and the APIs provided by PScout, our method scans the Smali code and looks for the mapped APIs. Then, with these APIs as a starting point, the process for the backtracking of parent function is iterated until the component can be identified through the files where a parent function is in.

The specific steps of the generation process of backtracking are as follows:

(1) Traverse all the Smali files in the folder after decompiling, and find out the sensitive API, such as *Landroid/telephony/TelephonyManager;>getDeviceId()Ljava/lang/String*. Then, look for the function that contains the sensitive API, which is the nearest function from the bottom to the top. Taking Figure 3 as an example, the function containing the sensitive API is *getDid()*.

(2) Generate a new parent function according to the sensitive API and the class it is in. We take the new parent function as the starting point, traverse all the Smali files in the folder again and find out the function which calls the newly generated parent function. Taking Figure 3 as an example, it means that we should find out where *getDid()* is called, i.e., *Lcom/qq/e/v2/managers/status/DeviceStatus;->getDid()Ljava/lang/String*.

```
.class public Lcom/qq/e/v2/managers/status/DeviceStatus;
.super Ljava/lang/Object;
.....
.method public getDid()Ljava/lang/String;
    .local 2
    .....
    move-result-object v0
    check-cast v0, Landroid/telephony/TelephonyManager;
    invoke-virtual {v0},Landroid/telephony/TelephonyMan-
    ager;>getDeviceId()Ljava/lang/String;
```

Figure 3: Code snippet of Lcom/qq/e/v2/managers/status/DeviceStatus.

(3) Iterate the methods of steps 1 and 2 until we get the component information where the parent function is located. The iteration result of *getDid()* is shown in Figure 4. The statement of "*.super*" displays the component information. The "*.super Landroid/app/Activity*" in the Smali file indicates that the parent function called *Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/Lang/String* belongs to the Activity component.

3.3. Native Code and Reflection. After decompilation, our method checks whether the application has the lib folder that stores the libraries. If the lib folder exists, the feature of the native code is set to 1. In order to alleviate the absence of reflection, FDP extracts APIs related to reflection, such as *Class.forName()*. The feature enables us to know that the hidden code is executed. Therefore, the feature of reflection is set to 1 or 0 depending on the code related to the reflection.

4. The Key of Backtracking

The feature of permissions used in components contributes a lot for FDP to improve the detection performance. Therefore,

```
.class public Lcom/qq/e/ads/AdActivity;
.super Landroid/app/Activity;
# instance fields
.field private a:Lcom/qq/e/v2/plugininterfaces/ActivityDelegate
#direct methods
.method public construction<init>()v
    .locals 0
    invoke-direct {p0}, Landroid/app/Activity;-><init>()v
    Return-void
end method
```

Figure 4: Code snippet of Lcom/qq/e/ads/AdActivity.

our method must backtrack the parent function calling the sensitive API to get the component information. Moreover, the key to the whole backtracking process is that the building process of the call graph cannot be interrupted. However, owing to the communication between components, callback, and other reasons, Android applications have implicit calls, which cause the interruption of a call graph depending on the conventional method. Some situations and solutions are presented as follows.

4.1. Annotations. Annotations are a language feature of Java widely used in the development of Android applications [34]. There are two types of annotations: the dalvik.Annotation package is not open to the public, which is only used for the core library and code testing, and it is under the *Dalvik\src\main\Java\dalvik\annotation* directory; the other one is android.Annotation and the corresponding annotation declaration is under the framework *\base\core\Java\android\annotation* directory. Annotation often appears in the process of backtracking.

Taking the app "net.maxicom.android.snake" as an example, the code snippet is shown in Figure 5. When the detection backtracks the parent function, FDP encounters the annotations code, in which "EnclosingMethod" specifies the scope of its own class. "Method" shows that the annotation acts on a function, and the value tells us that it is located in onCreate () of SnakeService. Therefore, FDP opens the SnakeService file, where the second line is "*.Super Landroid/app/Service.*" Thus, it can be inferred that the sensitive API is in the Service component.

4.2. Multithread Communications. Android is message-driven, and the essence of the message-handling mechanism is that one thread opens a loop to continuously listen to and process messages sent by other threads in turn. If a new thread is created to operate on the main thread, the system will throw an exception. Therefore, an asynchronous callback mechanism Handler is provided in the Android system for communication between threads [35].

As shown in Figure 6, the UI thread is the main thread. The system initializes a Looper object and also creates a MessageQueue associated with it. The Message is the object that the Handler receives and processes. The MessageQueue is the Message queue. Each UI thread can only have one Looper, continually pulling the Message out of the MessageQueue and distributing it to the corresponding Handler.

When Looper gets a new message, handleMessage() is called to process the new message. This Android message-handling mechanism causes the call graph to break, which leads to the failure of backtracking. Therefore, our method constructs an instance initialization method that has the special name "*<init>*," which is supplied by a compiler. Then, we use the method as the new starting point of backtracking.

```
# annotation
.annotation system Ldalvik/annotation/EnclosingMethod;
  value=Lnet/maxicom/android/snake/SnakeService;->onCreate
  ()V
.end annotation
```

Figure 5: Code snippet of "net.maxicom.android.snake."

Taking "ServiceCommunication1.apk" as an example, we backtrack the parent function calling sensitive API until the parent function is "*handleMessage*," as shown in Figure 7(a). Then, we find that the class inherits the *"Landroid/os/Handler"* based on the codes. If one method constructs "*Ledu/mit/icc_service_message/MessengerService$IncomingHandler;->HandleMessage(Landroid/os/Message;)V,"* the backtracking fails. Our method constructs "*Ledu/mit/icc_service_messages/MessengerService$IncomingHandler;-><init>(Ledu/mit/icc_service_messages/MessengerService;)V"* as the new start point and looks for its location. As shown in Figure 7(b), the second line "*.super Landroid/app/Service;*" shows that the dangerous permission is used in the service component.

4.3. Process and Thread. A process may contain several threads, which can be used as the basic unit of independent operation and independent scheduling. Because threads are smaller than processes and do not own system resources, their scheduling costs are low, which effectively improves the concurrent execution of programs between multiple programs of the system.

There are two ways to implement threads in an Android system. One is to extend the *"java.lang.threads,"* rewrite "*run(),*" and use multiple threads to complete their tasks separately. The other is to implement the Runnable interface and instantiate the thread class. The difference is that the threads created with the Runnable interface can share resources when multiple threads have access to the same resource, whereas the threads created by inheriting the thread class have their own resources.

When the parent function of the sensitive API is "*run(),*" it is necessary to check the keywords "*.Super*" and "*.Implement*" in the Smali file. If there exists "*.super Ljava/lang/Thread*" or "*.Implement Ljava/lang/Runnable*," we need to traverse all the Smali files again, instantiate the thread and use the function that starts the thread as the starting point for backtracking, and search upward for the parent function. Note that the thread class has the methods of "*run()*" and "*start(),*" whereas the implementation of the Runnable interface only has the "*run()*" method.

Taking "net.maxicom.android.snake" as an example, the application invokes location/LocationManager;->requestLocationUpdates(Ljava/lang/String; JFLandroid/location/LocationListener)V corresponding to the ACCESS_FINE_LOCATION permission, and its parent

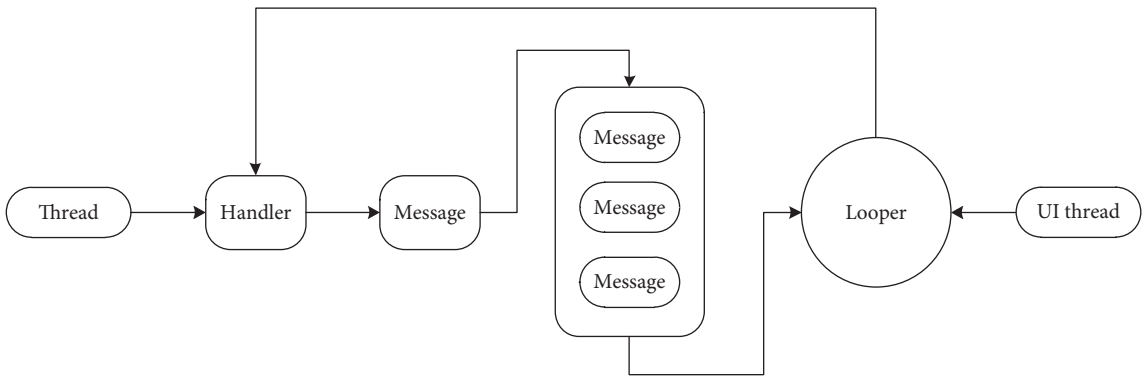

Figure 6: Sketch map of communication with UI thread.

```
.class Ledu/mit/icc_service_message/MessengerService$Incoming
Handler
.super Landroid/os/Handler
.source "MessengerService.java"
..... .
# virtual methods
.method public handleMessage(Landroid/os/Message;)V
```

(a)

```
.class public Ledu/mit/icc_service_messages/MessengerService;
.super Landroid/app/Service;
.source "MessengerService.java"
......
# direct methods
.method public constructor <init>()V
   new-instance v0, Landroid/os/Messenger;
   new-instance v1, Ledu/mit/icc_service_messages/MessengerService$IncomingHandler;
   invoke-direct {v1, p0}, Ledu/mit/icc_service_messages/MessengerService$IncomingHandler;->
   <init>(Ledu/mit/icc_service_messages/MessengerService;)V
```

(b)

Figure 7: Code snippet of SeviceCommunication1.apk.

function is Lnet/max-icom/android/snake/SnakeService$1;->run(V), where the second line of SnakeService$1 shows "*.super Ljava/lang/Thread*" in the "*net/maxicom/android/snake/SnakeService$1*" file. Provided that the method constructs its parent class function by class, method, and parameter as previously addressed, the backtrack fails. We take *Lnet/maxicom/android/snake/SnakeService$1;-> start (V)* as the new starting point for backtracking. Eventually, after scanning and matching, the parent function is *Lnet/maxicm/android/snake/SnakeService;->onCreate()*, and the file is located in *net/maxicom/android/snake/snakeService* where the second line shows that the sensitive API is called in the Service component, as shown in Figure 8.

5. Experimental Results and Discussion

5.1. Experiment Setup

Step 1: build a sample dataset. The malicious application samples come from MalGenome and AndroMalShare [26]. MalGenome is widely used by many researchers which help us compare the results with related approaches. As a remedy, AndroMalShare provides the latest malware samples which can be classified by malware families. We collect 1600 malicious applications as samples, which are classified according to Android malware families. The advantage of the dataset is that we can select specific Android malware families to build the detection model and examine the ability of the detection model detecting unknown applications from the other Android malware families. In order to ensure the balance of the data sets, we employ the crawler program to download 1700 benign applications from the Xiaomi App Store. To further check benign applications and malicious applications, we validate these applications with Virustotal, which is the website that analyzes suspicious files and URLs to detect types of malware, automatically sharing them with the security community.

We select 20 malware families shown in Table 2, including 673 malicious applications and 700 benign applications for building the detection model. The remaining applications are used as the testing set to examine the detection model.

```
.class public Lnet/maxicom/android/snake/SnakeService;
.super Landroid/app/Service;
.source "SnakeService.java"
```

Figure 8: Code snippet of SnakeService.

Step 2: decompile the application using the *apktool* tool to get the Smali code and AndroidManifest file. Then, we extract all features from the Smali code and AndroidManifest file. After further analysis, the fine-grained features are acquired.

Step 3: use the program written in Python to unify the features extracted in Step 2, and unify it into the arff format, which is convenient for machine learning by WEKA tools.

Step 4: compare the classification results of two different sets of permission features to select the features that can better represent the malicious behavior and exploit different machine learning algorithms to learn the data and select the optimal classifier according to the TP Rate, *F*-measure, and Receiver Operating Characteristic curve (ROC) area.

All experiments are carried out on a machine with a memory of 16 GB and Intel (R) Core(TM) i7-4720hq 2.60 GHz processor.

5.2. Evaluation Metrics. In this work, TP Rate, *F*-measure, and ROC area are employed to evaluate the performance of the detection model. As malicious applications are positive samples and benign applications are negative samples in our evaluation, we present four types of values. t_pos is the number of malicious applications correctly identified as malicious applications; f_neg is the number of malicious applications incorrectly identified as benign applications; t_neg is the number of benign applications correctly identified as benign applications; and f_pos is the number of benign applications incorrectly identified as malicious applications:

$$\begin{aligned}
\text{TP Rate} = \text{Recall} &= \frac{t_pos}{t_pos + f_neg},\\
\text{FP Rate} &= \frac{f_pos}{t_neg + f_pos},\\
\text{Precision} &= \frac{t_pos}{t_pos + f_pos},\\
F\text{-measure} &= \frac{2 * \text{recall} * \text{precision}}{\text{recall} + \text{precision}},\\
\text{Accuracy} &= \frac{t_{pos} + t_{neg}}{t_{pos} + t_{neg} + f_{pos} + f_{neg}}.
\end{aligned} \tag{1}$$

In these metrics, *F*-measure is an indicator referring to precision and recall. The ROC area is one of the most important evaluation metrics for checking any classification model's performance. The higher the ROC area, the better the classification.

Table 2: Malware families used in the detection model.

Family	Quantity
BaseBridge	60
BaseBird	39
BeanBot	6
KMin	40
GoneSixty	6
Fakeinst	16
DroidDream	45
Lotoor	8
Pjapps	44
SendPay	34
Geinimi	48
Bgserv	8
DDLight	40
HippoSMS	4
GoldDream	10
GingerMaster	7
DroidKungFu1	30
DroidKungFu2	20
DroidKungFu3	152
DroidKungFu4	56

5.3. Selection of Features. In order to achieve a better detection performance, choosing informative, discriminating, and independent features is a crucial step for classification. We design two experiments to compare the permission sets and use the testing set to validate. Experiment I: select all dangerous permissions as features of machine learning; Experiment II: select the top 20 used permissions of 86798 malware samples collected by AndroMalShare as the features of machine learning, as shown in Figure 9.

Experiment I chooses 927 malicious applications and 1000 benign applications that do not contribute to the detection model as a testing set. The result of dangerous permissions as features of machine learning is shown in Table 3. KNN and SVM are relatively better than other classifiers, and they have the highest TP Rate in malware detection. However, the performances of the four classifiers are not very satisfactory.

In Experiment II, we use the same training set to build the detection model and analyze the same testing set by the same algorithms. The only difference is that the features are the top 20 used permissions of 86,798 malicious applications collected by AndroMalShare. The experimental result is shown in Table 4.

It can be seen from the results that the SVM algorithm is the best in the TP Rate, reaching 90.9%. To compare the results in Tables 3 and 4, similar performances are observed for the same testing set and classifier. It is difficult to evaluate which permission set is better; however, we can identify the features which are more important for classification depending on the information gain of the features. The two sets of results are shown in Tables 5 and 6, respectively.

According to the ranked features and the actual usage status of permissions, we choose the 24 permissions as features from the two sets, as shown in Table 7. And we find that the permissions, that is, the requirement of executing malicious behavior may not be the significant like

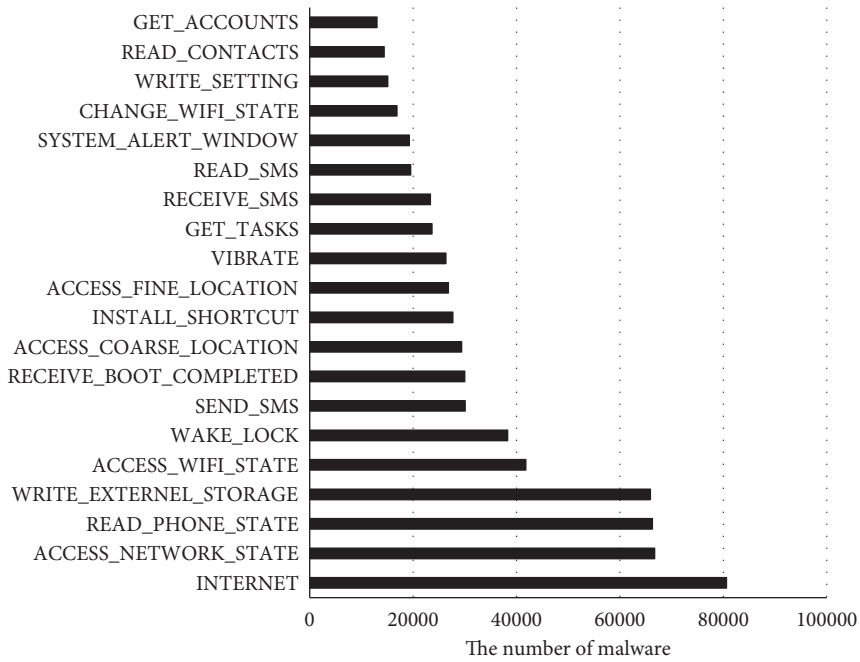

Figure 9: The top 20 used permissions of 86798 malicious applications.

Table 3: Detection rates with dangerous permissions of features.

	TP rate	FP rate	*F*-measure	ROC area
J48	0.877	0.149	0.877	0.813
KNN	0.899	0.091	0.901	0.937
SVM	0.899	0.099	0.900	0.900
NB	0.888	0.105	0.890	0.946

Table 4: Detection rates with the top 20 permissions of malware samples of features.

	TP rate	FP rate	*F*-measure	ROC area
J48	0.867	0.185	0.847	0.892
KNN	0.895	0.144	0.886	0.946
SVM	0.909	0.124	0.909	0.948
NB	0.888	0.103	0.891	0.942

INTERNET permission, which is used frequently by malware and benign applications.

Then, we employ the selected permission features as features of machine learning to learn the training set and analyze the same applications from the test set with the same classifiers. The result is shown in Table 8, with all indicators being superior to those in the above experiments. In these algorithms, the highest TP Rate achieved is 93.8% and the *F*-measure is the best with the J48 classifier.

5.4. FDP. Through further analysis on classifier errors, we find that most of these applications are wrongly classified as they declare dangerous permissions that are frequently declared by malware applications and benign applications, especially, READ_PHONE_STATE, ACCESS_COARSE_-LOCATION, ACCESS_FINE_LOCATION, SEND_SMS, and READ_SMS. Although these dangerous permissions are exactly same, the usage of these permissions is different between benign applications and malicious application. For example, the information gain of READ_PHONE_STATE is relatively low, but malicious applications often employ it in service component in order to avoid any users' attention which is different from benign applications.

Based on the previous work [23, 24], we further refine the above permissions and extract the information of

Table 5: Information gain of dangerous permissions.

Information gain	Permission
0.45761	CAMERA
0.38322	READ_EXTERNAL_STORAGE
0.23817	RECORD_AUDIO
0.18864	ACCESS_COARSE_LOCATION
0.16828	ACCESS_FINE_LOCATION
0.14783	GET_ACCOUNTS
0.12662	WRITE_EXTERNAL_STORAGE
0.09001	READ_CALENDAR
0.07381	READ_CALL_LOG
0.05988	WRITE_CALENDAR
0.02238	READ_PHONE_STATE
0.01669	READ_CONTACTS
0.01604	PROCESS_OUTGOING_CALLS
0.01390	WRITE_CALL_LOG
0.01322	CALL_PHONE
0.01233	READ_SMS
0.00565	SEND_SMS
0.00553	RECEIVE_WAP_PUSH
0.00343	BODY_SENSORS
0.00343	USE_SIP
0.00343	ADD_VOICEMAIL
0.00329	RECEIVE_SMS
0.00001	WRITE_CONTACTS

Table 6: Information gain of the top 20 permissions of 86,798 malware samples.

Information gain	Permission
0.33442	GET_TASKS
0.32002	SYSTEM_ALERT_WINDOW
0.29962	WRITE_SETTING
0.21801	WAKE_LOCK
0.1655	ACCESS_WIFI_STATE
0.1588	CHANGE_WIFI_STATE
0.14685	ACCESS_COARSE_LOCATION
0.12524	ACCESS_FINE_LOCATION
0.11073	GET_ACCOUNTS
0.09772	ACCESS_NETWORK_STATE
0.07362	VIBRATE
0.06159	WRITE_EXTERNAL_STORAGE
0.03961	READ_SMS
0.02997	INSTALL_SHORTCUT
0.00644	READ_CONTACTS
0.00546	READ_PHONE_STATE
0.00531	SEND_SMS
0.00399	RECEIVE_BOOT_COMPLETED
0.00314	RECEIVE_SMS
0.00154	INTERNET

Table 7: The 24 selected permissions.

CAMERA	GET_TASKS
WAKE_LOCK	READ_SMS
VIBRATE	WRITE_CALENDAR
READ_PHONE_STATE	READ_CONTACTS
PROCESS_OUTGOING_CALLS	WRITE_CALL_LOG
SYSTEM_ALERT_WINDOWS	WRITE_SETTING
READ_CALENDAR	READ_EXTERNAL_STORAGE
ACCESS_WIFI_STATE	ACCESS_COARSE_LOCATION
RECORD_AUDIO	WRITE_EXTERNAL_STORAGE
GET_ACCOUNTS	READ_CALL_LOG
CALL_PHONE	ACCESS_NETWORK_STATE
ACCESS_FINE_LOCATION	CHANGE_WIFI_STATE

components where the dangerous permissions used as the feature for the first time. And taking the code feature of malicious applications, intent that malicious applications are frequently used, and whether the application exploits native code, reflection mechanism, and root into consideration, the features of FDP are shown in Table 9, in which we add SEND_SMS and merge ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION into LOCATION, SEND_SMS and READ_SMS into SMS in view of curse of dimensionality.

FDP also uses the same data sets and classifiers for analysis. The result is shown in Table 10. Every algorithm has better performance than before. In these classifiers, J48 is the best and its TP Rate and *F*-measure are 0.945. It demonstrates that the features of FDP can better represent the malicious behavior.

Besides, the results of FDP are also shown as a ROC area in Figure 10. The area below the ROC curve is the value of the area under the curve (AUC). The larger the area, the better the classification. The results are obvious that these four classifiers have good performance in detecting malicious applications and all ROC area are over 0.9.

Table 8: Detection rates of the 24 selected permissions.

	TP rate	FP rate	*F*-measure	ROC area
J48	0.938	0.062	0.938	0.968
KNN	0.921	0.101	0.920	0.970
SVM	0.916	0.074	0.916	0.921
NB	0.918	0.107	0.917	0.949

5.5. The Evaluation of Dangerous Permissions Used in Component. In order to evaluate the importance of the feature of dangerous permissions used in components, we collect malicious applications from Android malware families, such as FakePlayer, DroidCoupon, TapSnake, and Plankton. These applications only declare few dangerous permissions such as READ_PHONE_STATE, ACCESS_COARSE_LOCATION, and READ_SMS which are frequently used by benign applications. This means there is not enough available information for classification if the detection methods are based on the declared permissions.

For comparison, we employ two sets of the permission features. The features in Table 7 are taken as one of two sets, and the other is that the features of dangerous permissions applied in components replace the declared permissions based on the features of Table 7, which involve READ_PHONE_STATE, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, and READ_SMS. For example, READ_PHONE_STATE is replaced by A-READ_PHONE_STATE, B-READ_PHONE_STATE, and S-READ_PHONE_STATE.

By testing 63 malicious applications and 70 benign applications that declare similar permissions, we perform 10-fold cross-validation using SVM algorithm. The accuracy of detection is 77.8% with the declared permissions, while the accuracy of detection reaches 92.5% with the fine-grained permissions. The scale of two data sets is relatively small because there are too few eligible benign applications.

In addition, we use information gain to evaluate the importance of features including the fine-grained permissions. As shown in Table 11, there are three features of dangerous permissions used in components ranked in the top fourteen of permissions. The result demonstrates the feature of permissions used in components contributes to the classification.

5.6. Detection of Unknown Malware Families. In order to examine the prediction ability of FDP, we collect the malware families as the unknown malware families shown in Table 12. These applications are not in the training set for constructing the detection model, which include the malware families that PApriori and DREBIN cannot detect.

We use three detection models to identify the same applications from unknown malicious families. The three detection models employ dangerous permissions as the features of machine learning (i.e., Experiment I), the top 20 used permissions of malicious applications declared as the features of machine learning (i.e., Experiment II), and FDP, respectively. The experimental results show that FDP can accurately detect more malicious families, especially with few dangerous permissions, which are shown in Figure 11. For instance, all the malicious

Table 9: The features of FDP.

CAMERA	D-CAMERA	WAKE_LOCK	REFLECTION
RECORD_AUDIO	D-RECORD_AUDIO	GET_TASKS	NATIVE CODE
READ_CONTACTS	D-READ_CONTACTS	VIBRATE	ROOT
READ_CALL_LOG	D-READ_CALL_LOG	PROCESS_OUTGOING_CALLS	I-BOOT_COMPLETED
CALL_PHONE	D-CALL_PHONE	SYSTEM_ALERT_WINDOWS	I-SMS_RECEIVED
WRITE_CALL_LOG	D-WRITE_CALL_LOG	WRITE_SETTING	I-BATTERY_CHANGE_ACTION
CHANGE_WIFI_STATE	D-CHANGE_WIFI_STATE	D- SMS	A-LOCATION
READ_CALENDAR	D-READ_CALENDAR	A- SMS	D-LOCATION
ACCESS_WIFI_STATE	D-ACCESS_WIFI_STATE	B- SMS	B-LOCATION
GET_ACCOUNTS	D-GET_ACCOUNTS	S- SMS	S-LOCATION
WRITE_EXTERNAL_STORAGE	D-WRITE_EXTERNAL_STORAGE	D-READ_PHONE_STATE	S-READ_PHONE_STATE
READ_EXTERNAL_STORAGE	D-READ_EXTERNAL_STORAGE	A-READ_PHONE_STATE	B-READ_PHONE_STATE
ACCESS_NETWORK_STATE	D-ACCESS_NETWORK_STATE	D- WRITE_CALENDAR	WRITE_CALENDAR

A: activity, B:broadcast receiver, D: unused or dynamical loading, I: intent.

Table 10: Detection rates of FDP.

	TP rate	FP rate	*F*-measure	ROC area
J48	0.945	0.061	0.945	0.939
KNN	0.937	0.078	0.936	0.963
SVM	0.929	0.088	0.928	0.920
NB	0.920	0.095	0.919	0.953

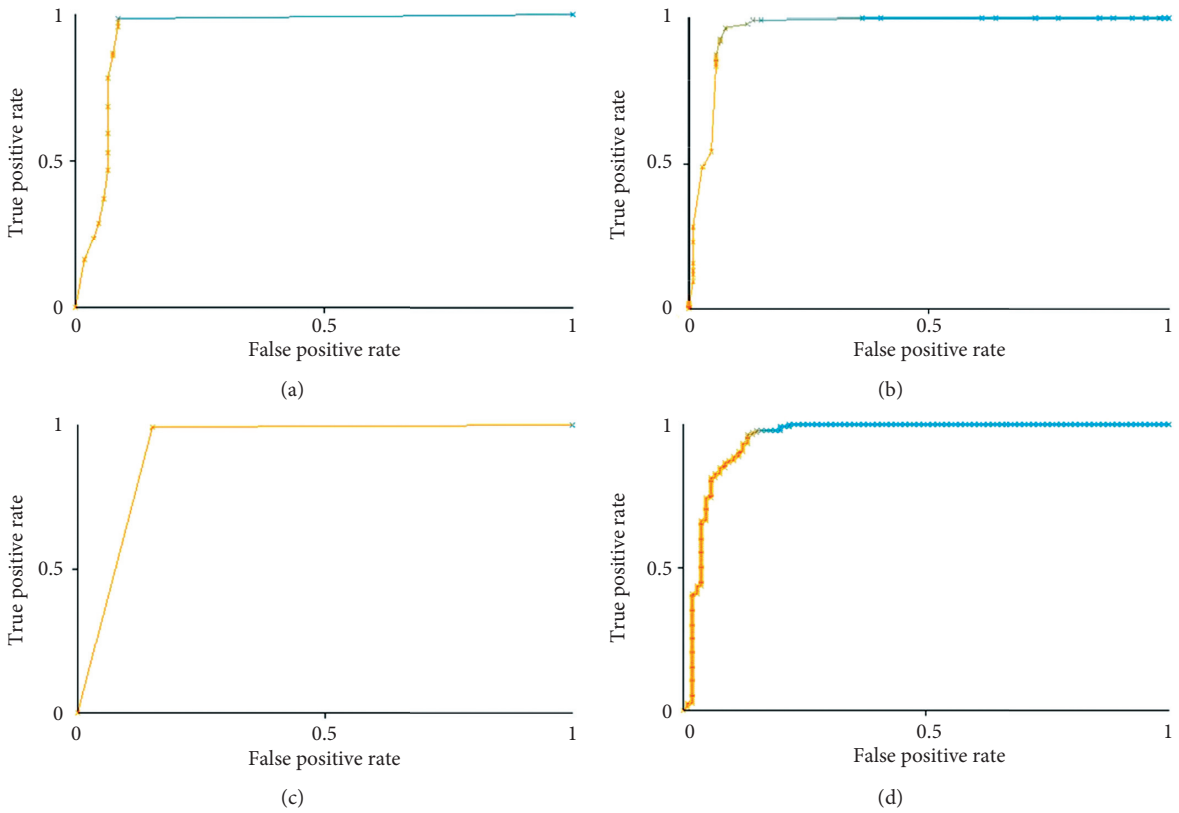

Figure 10: Detection performance as ROC area: (a) J48, (b) KNN, (c) SVM, and (d) NB.

Table 11: The top fourteen features of FDP according to information gain.

Permission	Information gain
S-READ_PHONE_STATE	0.55678
READ_EXTERNAL_STORAGE	0.41997
GET_ACCOUNTS	0.41997
GET_TASKS	0.41997
RECORD_AUDIO	0.41997
B-READ_PHONE_STATE	0.28129
CALL_PHONE	0.28129
CHANGE_WIFI_STATE	0.28129
VIBRATE	0.28129
READ_CONTACTS	0.28129
WRITE_SETTING	0.25643
S-READ_SMS	0.17095
PROCESS_OUTGOING_CALLS	0.17095
READ_CALENDAR	0.17095

Table 12: Malware families of the testing set.

Id	Family	Quantity
A	Asroot	6
B	FakePlayer	31
C	DroidCoupon	4
D	DroidDeluxe	1
E	Tapsnake	4
F	Gappusin	21
G	JsmsHider	23
H	RogueLemon	9
I	Plankton	24
J	Zsone	15
K	YZHC	8
L	JiFake	20

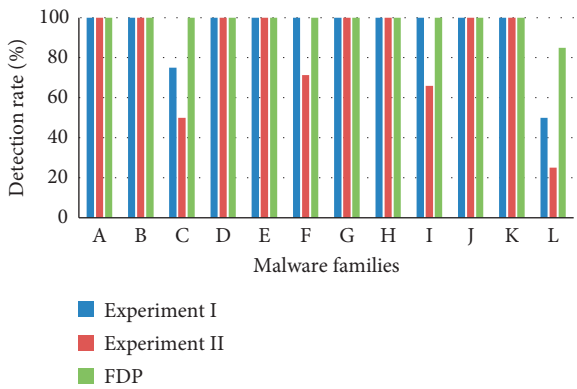

Figure 11: Detection of unknown malicious families.

applications from DroidCoupon family only declare READ_PHONE_STATE and WRITE_EXTERNAL_STORAGE; all the malicious applications from the Plankton family only declare READ_PHONE_STATE. It demonstrates that FDP can identify more malicious applications owing to the fine-grained permission features.

5.7. Comparison with Related Approaches. PApriori uses frequent pattern mining to get the maximum frequent permissions of 49 malware families and constructs the permission characteristics library to detect malicious applications. For 1260 malware applications from MalGenome, PApriori detects 87% of the malware samples. However, when malicious applications declare less dangerous permissions as benign applications, PApriori fails. PApriori lists malware families from MalGenome undetected by itself, such as Asroot, FakePlayer, DroidCoupon, DroidDeluxe, Tapsnake, and Plankton. And that is why PApriori has obvious limitations in detecting benign applications. FDP can detect all the above malicious families because of the fine-grained selected dangerous permissions, which represents the difference between benign applications and malicious applications.

DREBIN detects malware with the accuracy of 94%, but it cannot detect malicious applications from the Gappusin family because there are too few malicious features to identify the sample. We employ applications from Gappusin family as unknown malicious applications and the TP Rate gets 100%. Meanwhile, there is the problem of unbalanced data in the data sets of DREBIN in which it contains 123,453 benign applications and 5,560 malicious applications. If DREBIN chooses the balanced data sets, the accuracy may improve.

DroidEnsemble employs string features such as permissions and intent and structural features to detect malware applications. The size of the dataset and the composing proportion of benign applications and malicious applications are similar to FDP, and the accuracy reaches 98.4%. It is worth noting that structural features may perform poorly in detecting unknown malicious families because it relies heavily on previous identified malicious families. Therefore, it has the same problem as PApriori and DREBIN that the detection cannot take effect if there are not obvious differences between benign applications and malicious applications for the declared permissions.

5.8. Run-Time Performance. In order to evaluate the applicability of practical implementation of our method, we calculate the time consumption using 50 malicious applications and 50 benign applications, and the size of the applications is from 12 KB to 61 MB. On average, FDP can analyze one application within 15.205 s, which demonstrates that our method is efficient in Android malware detection, though there are many fine-grained information to extract.

6. Related Work

There are a lot of research studies related to FDP, which exploits static analysis to extract features such as permissions, API calls, and intent for detection.

According to the differences between malicious apps and benign apps, DroidRanger [36] summarized the rule of permission characteristics to detect unknown malicious apps. Enck et al. developed Kirin [5], which created nine security rules for security. Flet et al. proposed stowaway [37] to detect whether the app declares excessive permission based on the analysis of the mapping relationship between permissions and APIs. DREBIN [7] employed a static method to gather as many features of an application as

possible and use SVM to analyze them. Feizollah et al. [38] proposed the method to detect the malicious applications based on intents and permissions. DroidEnsemble [39] employed string features and structural features for identifying Android malicious applications and optimized the results with ensemble methods.

In this article, FDP differs in three aspects from the previous work. First, we choose the features depending on the characteristic of malicious applications in order to better represent the malicious behavior. Second, the permission features are further subdivided into fine-grained permission features, which enable FDP to detect more malware families, especially, when malicious applications declare few permissions or when the dangerous permissions declared by malicious applications are similar with those declared by benign applications. Third, FDP uses static analysis to extract fine-grained dangerous permission and other features, and it uses the J48 as the optimal classifier to detect malicious applications with TP Rate and *F*-measure reaching 94.5% and analyze an application in a reasonable time.

7. Conclusion and Future Work

In this paper, we propose a new method based on the fine-grained permissions for detecting Android malicious applications, which gathers the features that better represent the differences between malicious applications and benign applications. The experimental results demonstrate the effectiveness of our FDP method, which shows that FDP can detect more Android malicious families than existed methods. Moreover, the FDP method is efficient enough for practical implementation of Android malware detection.

Although FDP makes a breakthrough compared with the previous work, it still cannot extract more information from dynamic loading, reflection mechanism, and encryption because of inherent limitations of static analysis. Provided that FDP considers these limitations, it must use dynamic analysis method to extract related features, which increases the detection overhead significantly. In addition, more and more android applications try to protect themselves from decompiling, which also increases the difficulty for using our method.

Therefore, in the premise of efficiency, how to extract the valuable features from dynamic analysis and overcoming the antidisassemble problem is the direction of our future work. Besides, we only use the classic machine learning algorithms in this article, we will try to optimize these machine learning algorithms to construct a better detection model in our future work. Meanwhile, the training data is one important factor to affect the machine learning algorithms to recognize the features pattern. We try to collect more malicious applications and malware families in order to solve the problem of overfitting and underfitting in the future work.

Data Availability

The data used to support the findings of this study are available from AndroMalShare [26] and MalGenome. The applications from MalGenome are available from corresponding author upon request.

Conflicts of Interest

The authors declare no conflicts of interest.

Authors' Contributions

Xu Jiang proposed ideas and implemented and experimented with the system. Baolei Mao was responsible for the elaboration of ideas and comments on research. Jun Guan and Xingli Huang were responsible for a basic simulation to validate the proposed idea.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 61672433).

References

[1] World's Most Popular Mobile Operating Systems (Android VS IOS: Market Share 2012–2018), June 2019, https://ceoworld.b-iz/2019/01/18/worlds-most-popular-mobile-operating-syste-ms-android-vs-ios-market-share-2012-2018.

[2] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang, "Privacy risk analysis and mitigation of analytics libraries in the android ecosystem," *IEEE Transactions on Mobile Computing*, p. 1, 2019.

[3] Research and Analysis Report on Online Privacy and Online Fraud in 2018, June 2019, https://m.qq.com/security_lab/news_deta-il_473.html.

[4] İ. Doğru and Ö. Kiraz, "Web-based android malicious software detection and classification system," *Applied Sciences*, vol. 8, no. 9, p. 1622, 2018.

[5] W. Enck, M. Ongtang, and P. Mcdaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 235–245, Chicago, USA, November 2009.

[6] H. Yang, Y. Zhang, Y. Hu et al., "Android malware detection method based on permission sequential pattern mining algorithm," *Journal on Communications*, vol. 34, pp. 106–115, 2013.

[7] D. Arp, M. Spreitzenbarth, M. Hubner et al., "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, pp. 23–26, San Diego, USA, February 2014.

[8] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information-flow tracking system for real time privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 1–29, 2014.

[9] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pp. 611–622, Berlin, Germany, November, 2013.

[10] S. Kumar, R. Shanker, and S. Verma, "Context aware dynamic permission model: a retrospect of privacy and security in android system," in *Proceedings of the International Conference on Intelligent Circuits and Systems (ICICS)*, pp. 324–329, Phagwara, India, April 2018.

[11] S. Rosen, Z. Qian, and Z. M. Mao, "AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users," in *Proceedings of the third ACM Conference*

on Data and Application Security and Privacy—CODASPY '13, pp. 221–232, San Antonio, USA, February 2013.

[12] O. Yildiz and I. A. Doğru, "Permission-based android malware detection system using feature selection with genetic algorithm," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 2, pp. 245–262, 2019.

[13] R. S. Arslan, İ. A. Doğru, N. Barişçi, and N. Barişçi, "Permission-based malware detection system for android using machine learning techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 1, pp. 43–61, 2019.

[14] W. Wang, X. Wang, F. Dawai, 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.

[15] Z. Ma, H. Ge, Y. Liu et al., "A combination method for android malware detection based on control flow graphs and machine learning algorithms," *IEEE Access*, vol. 7, pp. 21235–21245, 2019.

[16] Y. Zhao, W. Du, and H. Yin, "DroidAPIMiner: mining API-level features for robust malware detection in android," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, pp. 86–103, Sydney, Australia, September 2013.

[17] A. K. Singh, C. D. Jaidhar, and M. A. A. Kumara, "Experimental analysis of android malware detection based on combinations of permissions and API-calls," *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 4, pp. 1–10, 2019.

[18] V. Avdiienko, K. Kuznetsov, A. Gorla et al., "Mining apps for abnormal usage of sensitive data," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 426–436, Florence, Italy, May 2015.

[19] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[20] H. Rathore, S. K. Sahay, P. Chaturvedi et al., "Android malicious application classification using clustering," in *Proceedings International Conference on Intelligent Systems Design and Applications*, pp. 659–667, Vellore, India, December 2018.

[21] P. Rovelli and V. Ýmir, "PMDS: permission-based malware detection system," in *Proceedings International Conference on Information Systems Security*, pp. 338–357, Hyderabad, India, December, 2014.

[22] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious android applications," *Future Generation Computer Systems*, vol. 36, pp. 122–132, 2014.

[23] D. J. Wu, C. H. Mao, H. M. Lee et al., "DroidMat: android malware detection through manifest and API calls tracing," in *Proceedings International Conference the Seventh Asia Joint Conference on Information Security*, pp. 62–69, Tokyo, Japan, August 2012.

[24] G. Tao, Z. Zheng, Z. Guo, and M. R. Lyu, "MalPat: mining patterns of malicious and benign android apps via permission-related APIs," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 355–369, 2018.

[25] Permission Overview, June 2019, https://developer.android.com/guide/topics/permissions/overview.

[26] AndroMalShare, June 2019, http://andromalshare.org:8080/#overview.

[27] Z. Qu, S. Alam, C. Yan et al., "DyDroid: measuring dynamic code loading and its security implications in android applications," in *Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 415–426, Denver, USA, June 2017.

[28] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015.

[29] F. Wei, X. Lin, X. Ou et al., "JN-SAF: precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1137–1150, Toronto, Canada, October 2018.

[30] L. Li, T. F. Bissyandé, D. Octeau et al., "DroidRA: taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 318–329, Saarbrücken, Germany, July 2016.

[31] Y. Liu, K. Guo, X. Huang, Z. Zhou, and Y. Zhang, "Detecting android malwares with high-efficient hybrid analyzing methods," *Mobile Information Systems*, vol. 2018, Article ID 1649703, 12 pages, 2018.

[32] K. W. Y. Au, Y. F. Zhou, Z. Huang et al., "PScout: analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 217–228, Raleigh, USA, October 2012.

[33] X. Hu, Q. Xi, and Z. Wang, "Monitoring of root privilege escalation in android kernel," in *Proceedings of the Cloud Computing and Security*, pp. 491–503, Haikou, China, June 2018.

[34] Q. Yang, G. Peng, P. Gasti et al., "MEG: memory and energy efficient garbled circuit evaluation on smartphones," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 4, pp. 913–922, 2019.

[35] S. Han, Y. Yun, and Y. H. Kim, "Profiling-based task graph extraction on multiprocessor system-on-chip," in *Proceedings of the 2016 IEEE Asia Pacific Conference on Circuits and Systems*, pp. 510–513, Jeju, South Korea, October 2016.

[36] Y. Zhou, Z. Wang, W. Zhou et al., "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets," in *Proceedings of the Network & Distributed System Security Symposium*, pp. 50–52, San Diego, USA, February 2012.

[37] A. P. Felt, E. Chin, S. Hanna et al., "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 627–638, Chicago, USA, October 2011.

[38] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "AndroDialysis: analysis of android intent effectiveness in malware detection," *Computers & Security*, vol. 65, pp. 121–134, 2017.

[39] W. Wei, Z. Gao, M. Zhao et al., "DroidEnsemble: detecting android malicious applications with ensemble of string and structural static features," *IEEE Access*, vol. 6, pp. 31798–31807, 2018.