Characterization of Android Applications with Root Exploit by Using Static Feature Analysis

Huikang Hao[1(✉)], Zhoujun Li[1,2], Yueying He[3], and Jinxin Ma[4]

[1] School of Computer Science and Engineering, Beihang University, Beijing 100191, China
{huikang329,lizj}@buaa.edu.cn
[2] Beijing Key Laboratory of Network Technology, Beihang University, Beijing 100081, China
[3] National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China
[4] China Information Technology Security Evaluation Center, Beijing 100085, China

Abstract. Recently, more and more rootkit tools are provided by some well-known vendors in the mainstream Android markets. Many people are willing to root their phones to uninstall pre-installed applications, flash third-party ROMs and so on. As it is reported, a significant proportion of Android phones are rooted at least one time. However, applications with root exploit bring critical security threat to users. When the phone is rooted, the permission system, which enforces access control to those privacy-related resources in Android phones, could be bypassed. Thus, the phone will be an easy point for malware to launch attacks. What's more, even the phone is unrooted, permission escalation attacks also can be carried out. Remarkably, an amount of sophisticated Android malware embeds root exploit payloads. Hence, root exploit always suggests high security risk. It is a pressing concern for researchers to characterize and detect applications with root exploit. In this paper, a novel method to extract key features of apps with root exploit is proposed. Contrary to existing works, contrasting the static features between applications with and without root exploit comprehensively are considered at the first time. We complete and evaluate the methodology on two clean apps and two malware dataset, comprising 52, 1859, 463 and 797 applications respectively. Our empirical results suggest the peculiar features can be obtained, which can capture the key differences between applications with and without root exploit to characterize Android root exploit applications.

Keywords: Android application · Root exploit · Static features · Apriori-based feature comparison · Feature combination · Characterization

G. Wang et al. (Eds.): ICA3PP 2015 Workshops, LNCS 9532, pp. 153–165, 2015.
DOI: 10.1007/978-3-319-27161-3_14

1 Introduction

Nowadays, mobile devices are reaching into almost every corner in our life rapidly. Due to the advanced smartphone operating system such as Android, IOS, we can enjoy feature-rich smartphones in which our security-sensitive information is stored, including contacts, photos and credentials. Among the mainstream mobile operating systems, Android dominates the mobile device market. As it is reported, more than 75 % shipments of smartphones run Android system in Q1 2015 [1]. The popularity of Android unsurprisingly draws malware authors' attentions, which results in the surge of Android malware. What's worse, an amount of sophisticated Android malware starts to launch attacks with root exploit. The utilization of root exploit technique makes that Android malware is more dangerous and difficult to detect.

One of the Android's most important security mechanisms against malware is permission control. Critical system resources are protected by permission mechanism so that any applications must explicitly declare what permissions they need to realize expected functions in the AndroidMeanifeast.xml file. When a user installs an app, a prompt will be raised to list all the critical permissions with its potential risky behaviors. The risk warning brings a binary choice for user to grant or reject these permissions that the application requires. Permission system ensures only the applications granted certain permissions can access corresponding resources. Otherwise, the resource access request would be rejected. To a large extent, we can say that the permissions that an app granted represent the app's capability to access system resources. As a result, permission system provides basic protections for the system resources of Android phones.

Android is a Linux-based platform. In the system design of Linux, users and groups are used to control access to the system's files, directories, and peripherals. The superuser (root) has complete access to the system resource and its configuration. An access to system resource with root privilege is almost unrestricted. Unprivileged users can use the *su* and *sudo* programs for controlled privilege escalation [2]. Nowadays, a variety of applications that provide one-click-root function emerge in every mainstream Android market. Many Android users utilize root exploit to customize their phones. Once a phone is rooted, its owner can remove the disliked pre-installed apps, customize personalized system, backup the phones, and flash third party ROMs. According to the report of NetQin [3], 23 % Android phones are rooted at least one time in China mainland by the first half of 2012. But it is important to note that root privilege brings serious security threats to users. First of all, the unrestricted resource access capability of root privilege makes it feasible that an application can access sensitive database files and hardware interfaces without corresponding permissions granted beforehand. That is to say, root exploit can disable the permission system. Faced with malware with root privilege, one of the Android security mechanisms, namely permission system, does not play any role. As a result, a lot of sophisticated malware samples launch attacks by using root exploit. As Yajin Zhou et al. revealed in [4], around one third (36.7 %) of the collected samples leverage root exploits to fully compromise the Android security. Moreover,

for an once rooted phone, unrooting the phone will not prevent Android system from suffering security threat. Permission escalation attacks also can be carried out [5]. Hence, an application with root exploit always suggests high security risk to Android phone. To deal with this security threat, it is a pressing concern for users and researchers to characterize and detect applications with root exploit.

In this paper, a novel method to extract critical features of apps with root exploit is proposed, and these extracted features can be used to characterize root exploit effectively. We collect a relatively complete dataset of clean apps with root exploit payload and contrast the static features with other benign apps comprehensively. It is noted that the contrast is carried out in two groups: a clean app group and a malicious app group. Each group has one dataset with root exploit and one dataset without root exploit payloads. In each group, we calculate the difference values of feature items between two datasets. Note that, 11 static features which compromises *permission, API call, 4 components,.so file. native code, dynamic code, reflection code* and *obfuscation code* are involved in our method. After comparing feature individually, we filter the items whose difference value less than 0 for each feature. Thus the left items form the candidate itemset for each feature. Then, for the generation of feature combinations, we join all the candidate itemset as an joint itemset. Apriori-based algorithm is introduced on the itemset to generate the feature combinations. Eventually, based on the feature combinations generated from two groups, we contrast the two sets and authenticate mutually. Hence, the shared and consistent feature combinations in the two groups are selected as the output of our method. The contributions of our work can be summarized as follows:

- A novel method to extract the key and peculiar features of apps with root exploit is proposed and implemented in our paper. These features can be used to characterize Android applications with root exploit.
- To the best of our knowledge, it is the first attempt to explore the key features of root exploit by contrasting the static features between applications with root exploit and other apps.
- We collect a relatively complete dataset of clean apps with root exploit, covering 7 mainstream Android markets. It may be useful for further research.

The rest of the paper is organised as follows: In Sect. 2, we analysis the root exploit process and its security threat to the phone as background. Section 3 describes the experimental datasets that we collect for our research. We present our method for extracting key features of apps with root exploit in Sect. 4. The implementation and the obtained result are then reported in Sect. 5 along with a discussion of our findings. Related works are shown in Sect. 6 and finally we concluded the paper in Sect. 7.

2 Background

In this section, we study the processes and security threat of root exploit. The basic processes of root exploit are consistent generally, which determines the

essential characteristics of root exploit. In addition, the root exploit disable permission system, which brings multiple risks for the phones.

2.1 Processes of Root Exploit

Since the heterogeneity of android os, a variety of rootkit tools are provided for users to root their phones. The processes of root exploit can be summarized as the following three steps: firstly, the applications called rootkit tools, exploit the Linux vulnerabilities to temporarily obtain root privilege. Up to now, the Linux vulnerabilities that have been exploited include *Gingerbreak, RageAgainstTheCage, ZergRush, Exploid, ASHMEM, Mempodroid, Levitator, Wunderbar, Zimperlich* and so on. Then, these applications place or replace a customized "su" binary file into */system/bin* or */system/xbin* directory. Finally, the root exploit payload sets the "su" file the *s* attribute that every user and role in the system can access it. Thus, the privilege escalation is completed and the system resource of system can be accessed with root privilege consistently.

2.2 Security Threat of Root Exploit

According to the Android core security design, the data and code execution of each application are isolated from each other by sandbox. The data and resource accesses are restricted strictly by the permission system. Because each Android application operates in a process sandbox, applications must explicitly share resources and data. The applications realize this by explicitly declaring permissions what they need for expected functions not provided by the basic sandbox. Thus, permission system, which is an access control mechanism, acts as the key system security mechanism in Android.

Android is a privilege-separated operating system, in which each application runs with a distinct system identity (Linux user ID and group ID) [6]. The Android system gives each app a distinct user ID(UID) at installation time and the ID cannot be changed all the time unless the app is removed. Generally, the UIDs of these user applications given by system are bigger than 10000. Thus, each app in Android system is isolated in its process space and regards as an unique Linux user. Unfortunately, when the phone is rooted, the UID of an app could be changed to 0, i.e. the root UID, which is able to match access control rule of all system resource almost.

The enforcement of permission system concentrates on two modules: Android system services and Linux kernel [7]. The two modules implement permission check on different levels, but they are all based on user and group mechanism in Linux access control.

Android System Services. The resources such as contacts and locations are protected by Android system services. For example, the location information is managed by *Location Manager Service*. When an app attempts to acquire these resources protected by system services, the service checks related

permissions by invoking the general permission check interfaces(such as *checkPermission,checkCallingPermission,checkUriPermission etc.*) of *ActivityManagerService*. Then the check is redirected to *PackageManagerService*, which keeps a table that records the granted permissions for each application. The *PackageManagerService* returns the check result according to the records and the system services which protect resource judges to accept or deny the access of resource. The parameters of these APIs generally comprise PID(Process ID) and UID(User ID), which suggests the permissions are verified finally by matching user ID.

Linux Kernel. The permissions related to file system and hardware interface such as camera, bluetooth and network etc. are enforced in Linux kernel level. In Android, each kernel-protected system resource is tagged with corresponding kernel-enforced permission to protect. Then, the permission is assigned with a unique GID(Group ID) as indentifier. if an app requests the related permissions in the manifest, the app,i.e. the UID will become a member of the user group that is permitted to access the resources. Any app is checked to verify whether it has the corresponding GID before accessing the protected resources.

When the phone is rooted, an app can run with root UID 0, which can pass all the permission checks. By analysing the implementation of permission system, we demonstrate that root exploit disable the permission system fundamentally. Hence, root exploit brings critical threat to the Android security.

3 Experimental Dataset

In order to contrast the static features effectively, quite a number of applications embedded root exploit payloads and apps without root exploit are needed to collect respectively. In our research, we collect 4 data sets in total, i.e. a clean app set with root exploit payload, a clean data set including benign apps without root exploit and two malware datasets whose root exploit payloads are at least one and none individually. For convenience of description, we call above 4 data set as *clean root set, clean set, malicious root set* and *malicious set* successively. Clearly, the 4 data sets can be divided into two compared groups: a clean set group and a malicious set group.

The *clean root set* consists 52 benign apps with root exploit. We collect these apps from official Google Play and 6 third party Android markets, which cover the mainstream Android markets nowadays. When collecting this dataset, we notice that many apps are found to occur repeatedly and one developer may release new version of identical app with slight variation. To prevent identical apps from having a large impact on the result, we consolidate duplicate apps into single instance in the dataset. In *clean set*, there are 1859 benign applications that we collect from official Googel Play. These apps in *clean set* are widely distributed around all the 44 categories of Google Play. Note that all above two datasets are verified manually. We have sent original datasets with 60 and 1863 apps to VirusTotal [8] and collect the analysis result. According to the analysis

result, we verify that whether the checked app embeds root exploit payload. Furthermore, the applications flagged as risky by at least 10 anti-virus products are removed and some applications about which VirusTotal has no information are also rejected. Eventually we collect 52 verified clean apps for *clean root set* and 1859 verified clean apps for our *clean set*. It is worth noting that 8 malware samples are found to masquerade as benign ones which claims to root users' phone in four third party markets(Yingyongbao, Anzhi, Mumayi, Anzhuo).

For our malware datasets, we used Zhou and Jiang's [4] collection of 1260 malicious applications, which comprises of 49 malware families. From the authors' analysis, among 1260 samples in the collection, 463 of them carry at least one root exploit payload. Here we collect these applications as *malicious root set*. The remainder 797 applications are well studied with no root exploit payload. Hence, we regard the 797 apps as our *malicious set*.

4 Method

To deal with the security threat, it is an urgent need to characterize and detect applications with root exploit. In order to extract key features of root exploit effectively, we propose a method to contrast the static features between apps with and without root exploit payload systematically. It should be clear that a complete static feature set is involved in our method, which compromises *permission, API call, 4 components,.so file. native code, dynamic code, reflection code* and *obfuscation code*. All above 11 individual features and feature combinations are considered to capture the characteristic of root exploit. Note that the clean set group and malicious set group are compared respectively by using the same procedures. The framework of our method is illustrated in Fig. 1 and the major processes of our method is outlined as follows.

Individual Feature Comparison and Filtering. In this process, 11 static features extracted from the 4 datasets are compared according to the two groups division. The purpose of this process is to obtain candidate features for further generation of feature combinations. For the feature comparison, we implement an analyzer based on Androguard [9] to extract and analyze all the 11 features. Androguard [9] is an open static analysis framework, which provides uncompress, decompilation and analysis of Android .apk file. All the feature items and its frequency can be obtained by our analyzer for further analysis.

Given Dx is one of the input datasets which contains n applications. Then for a certain application in Dx, let Si represent a feature of the application. Furthermore, we define $Si = \{A,B,C,\ldots\}$ as the set of possible items for a certain feature. Each item can be considered as a corresponding feature value as a feature. For example, permission CAMERA is an item of the feature "permission". In this process, we carry on feature comparison and filtering individually. For a feature Si, given one feature item A, we measure the importance of A by utilizing the difference of frequency. We calculate the differences in the same group by $diff(A) = freq(A)_{root} - freq(A)$. Here *freq(A)root* refers to the frequency

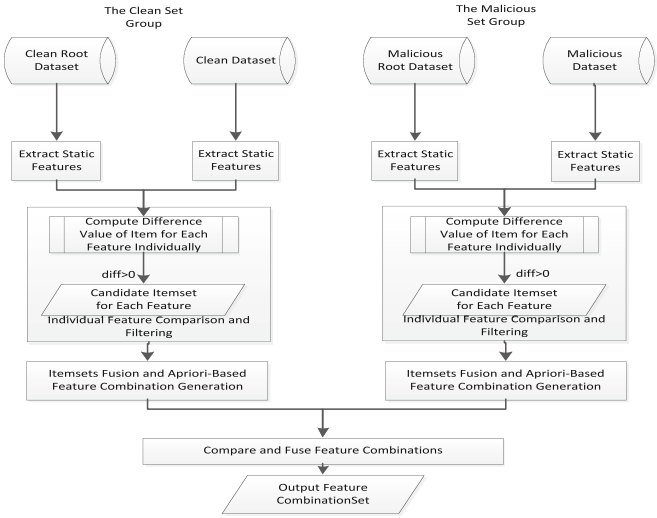

Fig. 1. The framework of our method

of item A in the root dataset and *freq(A)* is the frequency of A in the dataset without root exploit.

If *diff(A)* $> \delta$, we add item A to the candidate itemset Fi for Si. Otherwise, the item A will be discarded. Here, δ is a user-specified threshold value. In our research, we set the threshold δ 0 for capturing more feature items. After filtering the feature items, each feature has its corresponding itemset.

Apriori-Based Feature Combination Generation. In the former process, we obtain 11 candidate itemsets for 11 features individually. However, the behavior of applications is usually reflected in specific patterns and combinations of the extracted features. To this end, we define the joint set F that comprises all the 11 candidate itemsets for all the extracted features

$$F := F_1 \cup F_2 \cup \ldots \cup F_{10} \cup F_{11}$$

Apriori algorithm [10] is an algorithm for frequent item set mining and association rule learning. Here we amend this algorithm to generate the feature combination patterns. *Support* is usually used as the measurement for the effectiveness of the item pattern. To put it simple, let $a,b \subseteq F$, then the frequency of the apps that contains feature items a and b simultaneously in the total dataset is just the support. Here, we define the difference value of the two sets' support in one group as the *support* of the joint candidate itemset. In the practise of the algorithm, we set *support*$>= \epsilon$ as a rule to pruning the candidate item set, where ϵ is a specific threshold. We start the feature combination generation by enumerating the single item in F who matches pruning rule. Based on these single items, we add new items from F one by one to test their *support*.

This joining operation is repeated continuously to increase the number of items until the *support* of the item set less than pre-defined ϵ. In order to list all the possible feature combination, we take the pruning item set considerable and use frequent item set to generate new candidate item further. Finally, a set of feature combinations are formed for each comparing group. All the feature combinations are sorted according to its *support* for further analysis.

Feature Combination Contrast and Integration. All above two processes are conducted on the two group: the clean group and malicious group. After the previous procedures, two sets of feature combinations are obtained. As the final step, we contrast the two sets and authenticate mutually. Based on the result of comparison, we leave the shared feature combinations of the two groups as the outputs of our method.

5 Implementation and Experimental Result

5.1 Implementation

By using our approach, we complete a demo based on Androguard [9] in Python. As stated in pervious sections, 11 features in total is selected in our work as feature to describe applications behavior characteristic. Among them, *permission* and *API calls* are inspected by counting the feature names and their frequency. Here we focus *required permissions* and all the *API* decompiled from .apk file are considered. For the 4 app components, namely, *activity, service,content provider, broadcast receiver*, its numbers in each app are collected in a feature set. What's more,.so file are investigated with file name and file number. *native code, dynamic code, reflection code,obfuscation code* refer to whether applications have native code, dynamic code, reflection code and obfuscation code. We test apps with these features and compute their frequency.

For our implementation, Androguard-based module is used to extract static features. We amend Androguard [9] so that it can uncompress and decompile the .apk files to output all checked items of above 11 features. All these features are written in text file for further analysis. Then we complete analysis modules according to our algorithm in Python. Two scripts are coded to contrast individual features and generate feature combinations respectively. Note that, when we introduce our Apriori-Based feature combination generation algorithm, we set the threshold of *support* ϵ as 0.5, which is mainly to capture more effective feature combinations to distinguish root exploit.

5.2 Individual Feature Comparison

Our experiments are conducted on two dataset groups, which covers four datasets that are described exhaustively in Sect. 3. Then we show the results of individual feature comparison for the two groups separately.

Table 1. A list of top 15 required permissions ordered by difference value for the clean app group and malicious app group.

The clean group comparison		The malicious group comparison	
Permission	Difference value	Permission	Difference value
READ_PHONE_STATE	0.5769	CHANGE_WIFI_STATE	0.6777
GET_TASKS	0.5385	ACCESS_WIFI_STATE	0.4048
RECEIVE_BOOT_COMPLETED	0.5192	WRITE_EXTERNAL_STORAGE	0.3613
ACCESS_WIFI_STATE	0.5192	READ_EXTERNAL_STORAGE	0.3117
SYSTEM_ALERT_WINDOW	0.4423	ACCESS_LOCATION_EXTRA_COMMANDS	0.3094
WRITE_SETTINGS	0.4230	GET_TASKS	0.1801
WRITE_EXTERNAL_STORAGE	0.4230	ACCESS_NETWORK_STATE	0.1340
INSTALL_SHORTCUT	0.4038	INSTALL_PACKAGES	0.1264
KILL_BACKGROUND_PROCESSES	0.3846	ACCESS_FINE_LOCATION	0.0891
MOUNT_UNMOUNT_FILESYSTEMS	0.3846	RECEIVE_BOOT_COMPLETED	0.0617
ACCESS_MTK_MMHW	0.3461	CHANGE_CONFIGURATION	0.0501
CHANGE_WIFI_STATE	0.3269	READ_PHONE_STATE	0.0267
RESTART_PACKAGES	0.3269	ADD_SYSTEM_SERVICE	0.0204
READ_LOGS	0.3077	BROADCAST_STICKY	0.0200
GET_PACKAGE_SIZE	0.2885	MODIFY_AUDIO_SETTINGS	0.0195

Permission. For the clean group, 112 permissions are involved in total. 45 permissions occurs in both the *clean root dataset* and *clean dataset*. 33 permissions are found only in the *clean root dataset*. The remainder 34 permissions are only required only by the applications in the *clean dataset*. For the malicious group, 64 permissions occur at all the time in both two datasets of the group. 16 and 35 permissions appear only in the *malicious root set* and *malicious set* respectively. In summary, there are 115 permissions for the malicious group. Note that, not all the permissions are Android-defined permissions. According to our filtering rules, 68 and 42 permissions are left to incorporate to candidate itemset of clean group and malicious group respectively. Table 1 lists the top 15 permissions ordered by difference value for the two groups.

According to the comparison results ordered by difference value, we observe that there are 20 shared permissions in the candidate itemsets for the two groups. Specially, among the top 15 required permissions in Table 1, 6 permissions, i.e. READ_PHONE_STATE, GET_TASKS, RECEIVE_BOOT_COMPLETED, ACCESS_WIFI_STATE, WRITE_EXTERNAL_STORAGE, CHANGE_WIFI_STATE, are common permissions.Permision READ_PHONE_STATE, GET_TASKS and RECEIVE_BOOT_COMPLETED are the top 3 for the clean group, while for the malicious group, thet are CHANGE_WIFI_STATE, ACCESS_WIFI_STATE, WRITE_EXTERNAL_STORAGE. We notice that READ_PHONE_STATE and RECEIVE_BOOT_COMPLETED are risky permissions significantly and requested widely in malicious data sets as reported in previous work [11], which results in that the two permissions don't have a particularly significant difference value as in the clean group. However, they are still important verifiable characteristic for root exploit. As literature [12] demonstrated the root exploit example, the rootkit tools collects information needed to exploit as the first step. So it is necessary to request READ_PHONE_STATE. Similarly,

the rest 5 permissions are needed to complete basic processes for root exploit. Permission system is the key security design for Android. When a phone is not rooted, the system resources are restricted by permissions. Once an application attempts to root the phone, it should require a part of basic permissions for to complete root exploit. Meanwhile, a mass of permissions related to specific behaviors can be avoided.

API Calls. API calls are fine-grained descriptions of application behaviors. The so-called *Used Permissions* can be reflected by API calls. In our method, all the API calls that can be extracted from .apk file are involved. For the clean set group, 73886 APIs are extracted and 10472 API calls are common used by two datasets. Besides, 28678 API calls are unique ones that only appear in *clean root set*. For the malicious set group, 57779 API calls are the total number and 4687 API calls are common ones, 18000 API calls are found only in *malicious root set*. After sorting these API calls in accordance with difference values, we select 29792 and 21247 API calls whose values outnumber 0 to add into candidate itemsets for clean and malicious group respectively. Combining the comparison result of the two groups, we notice that third-party packages and corresponding APIs are widely used in applications with root exploit. For the clean group, packages *com.tencent.mm.sdk,com.umeng,com.zhiqupk.root,com.feiwo et.al.* are discovered in the *clean root set* exclusively. These third-party APIs are the key difference between apps with and without root exploit. Specially, among the top 10 APIs in the differen value order, 7 APIs are from the package *com.tencent.mm.sdk*. Similarly, for the malicious group, the packages *com.google.update,com.keji,net.youmi.android* are viewed only in *malicious root set* and the packages *uk.co.lilhermit.android,com.adwo.adsdk,com.madhouse.android.ads,com.admogo.adapters,com.vpon.adon.android* are common API packages which can distinguish root exploit effectively.

4 Components. The 4 components reflect the structure of an application. We extract the number of certain component in an app as the feature item and compute its corresponding frequency among all the apps in the dataset. For the clean group, the total numbers of candidate items for *activity, content provider, service* and *broadcast receiver* are 20,5,13 and 12 respectively. On the other hand, the candidate item number of *activity, content provider, service* and *broadcast receiver* are 21,3,3 and 3 in the malicious group. The difference values of most the candidate items for the two groups doesn't exceed 0.1, while only the item "1" of *service* in the malicious group is 0.3014, which acts as the significant feature item.

.so File. The .so files are shared libraries (.so) in an Android application, which are usually under the dictionaries *lib/armeabi* and *assets*. Usually, the authors of applications dynamically load native code or author-specific code by introducing .so file. In our method, we compare the usage of .so files in two groups. For the clean set group, the type number of .so file in *clean root set* and *clean set* are 71 and 41 respectively. The candidate itemset has 71 files, in which *assets/libsecmain.x86.so,assets/libsecexe.x86.so,lib/armeabi/librgsdk.so* and the

file *lib/armeabi/libsmartutils.so* are the top 4 files whose difference values are surpass 0.10. For the malicious set group, the utilization of .so files of all the 463 apps in *malicious root set* converges on 14 types, while there are 37 types .so file in *malicious set*. As a result, 8 files are common in both two dataset and 13 files whose difference values are more than 0 are remainder as candidate items. Note that, the difference value of *lib/armeabi/libnative.so* file is 0.6160, which is much higher than other 12 files. The values of the rest ones in itemset are all less than 0.0205.

Native Code, Dynamic Code, Reflection Code and Obfuscation Code. For these 4 features, everyone has only one feature item, which refers to a certain application has native code, dynamic code, reflection code and obfuscation code or not. According to the calculation of the 4 feature item frequency, It is noticed easily that *native code* is a significant feature that differentiates applications with root exploit from others. *Dynamic code* feature is more frequent in applications without root exploit and *obfuscation code* feature has made no difference for our method. As for *reflection code* feature, it is not consistent in two groups. Finally, only *native code* feature item is left for the clean group and *native code*,*reflection code* are for malicious group.

5.3 Feature Combination Generation

After individual feature comparison, candidate itemsets for each feature are gained. Then, we unite these 11 candidate itemsets into a joint candidate set for the two groups.Thus 29982 feature items are included into the joint set of clean group and 21334 ones are in the joint set of malicious group. By introducing our Apriori-Based feature combination generation algorithm, a set of feature combinations are obtained and we display them in Table 2. All the supports of these feature combinations is over 0.5. We authenticate the result of two groups mutually and find out that the combination (READ_PHONE_STATE,ACCESS_WIFI_STATE,Native Code) and the combination (ACCESS_WIFI_STATE, CHANGE_WIFI_STATE, *lib/armeabi/libnative.so*,uk.co.lilhermit.android.core.Native,Native Code) are generally consistent. *ACCESS_WIFI_STATE* and *Native Code* are common features of two group and they can group together as a feature combination to be the output of our method. In addition, the peculiar feature items that only appears in the *clean root set* and *malicious root set* are collected as supplementary method, e.g. APIs in *com.tencent.mm.sdk.platformtools* are used only in *clean root set*, whose difference values doesn't surpass 0.5. But when an application uses these APIs, we justify the application has a higher possibility of embedding root exploit payloads.

6 Related Work

Note that some sophisticated malware samples launch attacks with root exploit, many techniques have been proposed as a part of malware detection technique. DroidRanger [13] implements dynamic execution monitoring that focuses on

Table 2. A set of feature combinations for the two groups.

	Feature combinations
Clean Group	(READ_PHONE_STATE,ACCESS_WIFI_STATE,Native Code), (READ_PHONE_STATE,GET_TASK,Native Code) (RECEIVE_BOOT_COMPLETED)
Malicious Group	(ACCESS_WIFI_STATE,CHANGE_WIFI_STATE,*lib/armeabi/libnative.so*,uk.co.lilhermit.android.core.Native,Native Code), (CHANGE_WIFI_STATE,com.google.update.Dialog,com.google.update.UpdateService)

system calls used by existing Android root exploits and/or made with the root privilege. In [14], in order to detect root exploit, the authors distill each known vulnerability into a corresponding static vulnerability-specific signature to capture its essential characteristics. Similarly, Rastogi et.al. [15] proposed the method based on vulnerability conditions, which can be considered as the signature.

Many tools and frameworks which are devoted exclusively to detect and prevent root exploit are also designed. Ho et. al. [16] propose PREC, a framework which can identify system calls from high-risk components and execute those system calls within isolated threads to detect and stop root exploit.In [17], a system that enables to extract and collect events related to root exploit is proposed, which can cope with root exploit effectively.

In summary, the methods proposed by the previous works mainly focus on two points. Many techniques [13,16,17] detect root exploits based on monitoring and searching for system calls and events related to known root exploit processes dynamically. Compared to these works, our method is implemented based on static feature analysis and we focus on the reflections of root exploit on Android code level, but not the system calls or events. On the other hand, such approaches [14,15] learn the well-studied root exploit vulnerabilities and extract the preconditions of them as signature for rule-matching. We observe that these methods are based on behaviors that will be exhibited when the vulnerability is being exploit. These works draw attentions to root exploit vulnerabilities, while our method inspects to search for essential features for root exploit on the .apk file level. Unlike the previous methods, our method provides a new angle to study the root exploit characterization and detection.

7 Conclusion

Root exploit brings a variety of security threats to the phones. Applications with root exploit always suggest high risk. To characterize and detect apps with root exploit, we propose a novel method to extract peculiar features of apps with root exploit. To the best of our knowledge, this work is the first one to focus on

characterizing root exploit from the angle of static feature contrast. By applying our method, a set of key features and corresponding feature combinations are obtained to capture the key differences between applications with and without root exploit.

Acknowledgements. This work was supported in part by National High-tech R&D Program of China under grant No. 2015AA016004, NSFC under grants No. 61170189 and No. 61370126.

References

1. IDC. http://www.idc.com/getdoc.jsp?containerId=prUS25282214
2. Users and Groups. https://wiki.archlinux.org/index.php/Users_and_groups#Group_management
3. NetQin: 2012 moblie phone security report (2012). http://cn.nq.com/neirong/2012shang.pdf
4. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: S&P 2012, pp. 95–109. IEEE (2012)
5. Zhang, Z., Wang, Y., Jing, J., Wang, Q., Lei, L.: Once root always a threat: analyzing the security threats of android permission system. In: Susilo, W., Mu, Y. (eds.) ACISP 2014. LNCS, vol. 8544, pp. 354–369. Springer, Heidelberg (2014)
6. System Permission. http://developer.android.com/intl/zh-cn/guide/topics/security/permissions.html
7. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B.: Vetting undesirable behaviors in android apps with permission use analysis. In: CCS 2013, pp. 611–622 (2013)
8. VirusTotal. https://www.virustotal.com
9. Androguard. http://code.google.com/p/androguard
10. Apriori algorithm. https://en.wikipedia.org/wiki/Apriori_algorithm
11. Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X.: Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inf. Forensics Secur. **9**(11), 1869–1882 (2014)
12. Lee, H.-T., Kim, D., Park, M., Cho, S.: Protecting data on android platform against privilege escalation attack. Int. J. Comput. Math. (ahead-of-print), 1–14 (2014)
13. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: NDSS (2012)
14. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 281–294. ACM (2012)
15. Rastogi, V., Chen, Y., Enck, W.: Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 209–220. ACM (2013)
16. Ho, T.-H., Dean, D., Gu, X., Enck, W.: Prec: practical root exploit containment for android devices. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, pp. 187–198. ACM (2014)
17. Ham, Y.J., Choi, W.-B., Lee, H.-W.: Mobile root exploit detection based on system events extracted from android platform. In: SAM 2013, 1p. WorldComp (2013)