Overprivileged Permission Detection for Android Applications

Sha Wu, Jiajia Liu*
School of Cyber Engineering
Xidian University
Xi'an, China
*Email: liujiajia@xidian.edu.cn

***Abstract*—Android applications (Apps) have penetrated almost every aspect of our lives, bring users great convenience as well as security concerns. Even though Android system adopts permission mechanism to restrict Apps from accessing important resources of a smartphone, such as telephony, camera and GPS location, users face still significant risk of privacy leakage due to the overprivileged permissions. The overprivileged permission means the extra permission declared by the App but has nothing to do with its function. Unfortunately, there doesn't exist any tool for ordinary users to detect the overprivileged permission of an App, hence most users grant any permission declared by the App, intensifying the risk of private information leakage. Although some previous studies tried to solve the problem of permission overprivilege, their methods are not applicable nowadays because of the progress of App protection technology and the update of Android system. Towards this end, we develop a user-friendly tool based on frequent item set mining for the detection of overprivileged permissions of Android Apps, which is named Droidtector. Droidtector can operate in online or offline mode and users can choose any mode according to their situation. Finally, we run Droidtector on 1000 Apps crawled from Google Play and find that 479 of them are overprivileged, accounting for about 48% of all the sample Apps.**

***Index Terms*—Android application, overprivileged permission, frequent item set mining, detection tool**

I. Introduction

With the expansion of the coverage of smartphone in daily life, a smartphone almost has become a personal computer bringing great convenience to user. According to the latest research from Gartner, Google's Android further extended its lead over Apple's iOS in the second quarter of 2018, occupying 88 percent market share [1]. The rapid development of Android system naturally leads to a great boom of Android applications. The open-source feature of Android system have made it a popular platform for third-party applications. Since the smartphone contains user's private information, Android system exploits permission mechanism to regulate how applications access certain resources. However, security concerns still arise because applications usually declare extra permissions sensitive to user's privacy while having nothing to do with their functions. In [2], authors find that the permissions declared by the majority of mobile applications are not in accordance with their actual requirements. The problem that an App declares extra permission is called permission overprivilege by us. The reason for the phenomenon of permission overprivilege is the absence of strict development specifications and effective supervision. Besides, users have few knowledge of App's permissions and there is not any detection tool in the market, thus most users grant any permission declared by the App. All the above facts make users suffer from a high risk of privacy leakage.

There have been some pioneering works studying the Apps' permissions. In [3], authors built a tool named Stowaway to detect overprivileged permission of an App. Stowaway determines the set of API (Application Programming Interface) calls that an application uses and then maps those API calls to permissions. They applied Stowaway to a set of 940 applications and found that about one-third of them were overprivileged. In [4], a tool named PScout was built to extract the permission specification from the Android source code using static analysis and take the first steps to answer some key questions about Android's permission system. The authors used PScout to analyze the source code of Android 2.2 to Android 4.0, and obtained a more complete mapping between API calls and permissions. Using PScout static analysis tools, they found that 543 out of 1260 applications declared at least one extra permission that was not in the list produced by PScout. Bartel etc. presented an advanced class-hierarchy and field-sensitive set of analyses to extract the mapping between API methods and permissions [5]. By running their tool Spark-Android on the 679 applications downloaded from the official Android market, they found 124 declared one or more permissions which they did not use. Chester etc. created a detection tool named M-perm which combines static and dynamic analysis in a computationally efficient manner compared to previous tools [6]. It's a pity that the authors tested only 50 Apps, so there is no convincing results.

With the enhancement of App protection technology and the update of Android system, there are several limitations in previous studies. First, static analysis methods used by the previous works are hardly realizable since code encryption technology develop rapidly. Nowadays, most application developers adopt code encryption to prevent the App from being decompiled. Code encryption technology confuses the source code, which makes people fail to decompile it or obtain obfuscated code hard to understand. Second, the previous works are based on specific Android versions and will be useless if the Android version changed. Google continuously updates Android system, which also has an influence on the App development. In this case, the previous tools are invalid. Finally, all the previous works did not provide a publicly-available and user-friendly tool for ordinary users.

In view of the limitations of previous studies, we develop a tool named Droidtector for users to detect overprivileged permission of an application. The tool have two work modes: online mode and offline mode, and user can choose any mode according to the situation. The major differences between our work and previous studies are summarized as follows:

- The work of this paper is based on frequent item set mining. Running FP-growth algorithm on the permission files of many Apps under a certain category, we can get the permission set related to the function of the App of the category.
- Our research doesn't need the analysis of the source code of Apps or Android system. What we need is just the App's *AndroidManifest.xml* file, of which the code is not confused by encryption technology. Any App's *AndroidManifest.xml* file can be dumped expediently.
- The methods and tool in this paper have nothing to do with Android version, hence the tool is applicable no matter which Android version the App is compatible with.

The rest of this paper is organized as follows: in Section II, we introduce permission mechanism and dangerous permissions of Android. In Section III, we present the methodologies of our work. The modules design and the effectiveness verification of the tool are thoroughly described in Section IV. The test results on 1000 Apps and the analysis of the results are showed in Section V. Finally, we conclude the whole paper in Section VI.

II. Overview

A. Permission Mechanism

The most notable feature of Android is open source, which enables that any people can release the App developed by himself. Open source brings people advantages along with disadvantages. Due to the absence of effective control in the process of developing and releasing Apps, smartphone users usually face the risk of privacy leakage. In order to solve this security problem, Android executes permission mechanism [7]. Android security architecture consists of the following parts:

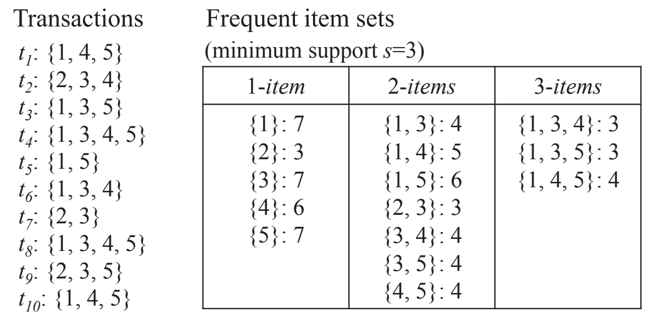
1) Application components: An Android application contains four components: activity, service, content provider and broadcast receiver. Activity is responsible for the real-time interaction between user and device. Service operates in the background without user interface. Content provider supports data sharing between different applications. Broadcast receiver is applied in the interaction between different components.

TABLE I
Dangerous Permissions

Number	Permission Name (prefix is omitted)
1	READ_CALENDAR
2	WRITE_CALENDAR
3	CAMERA
4	READ_CONTACTS
5	WRITE_CONTACTS
6	GET_ACCOUNTS
7	ACCESS_FINE_LOCATION
8	ACCESS_COARSE_LOCATION
9	RECORD_AUDIO
10	READ_PHONE_STATE
11	READ_PHONE_NUMBERS
12	CALL_PHONE
13	ANSWER_PHONE_CALLS
14	READ_CALL_LOG
15	WRITE_CALL_LOG
16	ADD_VOICEMAIL
17	USE_SIP
18	PROCESS_OUTGOING_CALLS
19	BODY_SENSORS
20	SEND_SMS
21	RECEIVE_SMS
22	READ_SMS
23	RECEIVE_WAP_PUSH
24	RECEIVE_MMS
25	READ_EXTERNAL_STORAGE
26	WRITE_EXTERNAL_STORAGE

2) AndroidManifest.xml file: AndroidManifest.xml file is the entry of an application, including the following authentication information: version, package name, components, permissions and other basic information. When an App is installed, some permissions registered in the *AndroidManifest.xml* file need user's authorization. An application must register the permission it wants in *AndroidManifest.xml* file, or the relevant functions cannot operate normally. In *AndroidManifest.xml* file, <uses-permission>, <permission> and <permission-group> are the tags related to permission. Tag <permission> is used to register the permission defined by developers while <uses-permission> tag declares system permission. Our research focuses on the permissions under <uses-permission> tag.

3) Protection layer: Protection layer refers to the attribute of permission, which determines authorization method. Some permissions is authorized automatically by system, and some need user's grant while installing the App. Besides, there are some permissions need to verify the certificate signed with secrete key by developers. Only when the App that attempts to use a permission is signed by the same certificate as the App that defines the permission can the permission be granted [8].



Transactions

t_1: {1, 4, 5}
t_2: {2, 3, 4}
t_3: {1, 3, 5}
t_4: {1, 3, 4, 5}
t_5: {1, 5}
t_6: {1, 3, 4}
t_7: {2, 3}
t_8: {1, 3, 4, 5}
t_9: {2, 3, 5}
t_{10}: {1, 4, 5}

Frequent item sets
(minimum support s=3)

1-*item*	2-*items*	3-*items*
{1}: 7	{1, 3}: 4	{1, 3, 4}: 3
{2}: 3	{1, 4}: 5	{1, 3, 5}: 3
{3}: 7	{1, 5}: 6	{1, 4, 5}: 4
{4}: 6	{2, 3}: 3	
{5}: 7	{3, 4}: 4	
	{3, 5}: 4	
	{4, 5}: 4	

Fig. 1. There are 10 transactions in total, and each transaction can be regarded as the set of dangerous permissions declared by an App. Each digit represents a dangerous permission listed in Table I. With minimum support $s = 3$, the frequent item sets are shown in the right. All 3-items sets are called maximal frequent item set. The merged maximal frequent item set is $\{1, 3, 4, 5\}$.

TABLE II
Merged Maximal Frequent Permission Sets

Category	**Dangerous Permissions**
Sports	6, 7, 8, 10, 25, 26
Travel & Local	3, 6, 7, 8, 10, 25, 26
Medical	7, 8, 25, 26
Books & Reference	3, 6, 10, 25, 26
Maps & Navigation	3, 7, 8, 10, 25, 26
Weather	7, 8, 10, 25, 26
Photography	3, 7, 8, 9, 25, 26
Education	3, 6, 9, 10, 25, 26
News & Magazines	6, 7, 8, 10, 25, 26
Social	3, 4, 6, 7, 8, 9, 10, 25, 26
Video Players & Editors	3, 9, 10, 25, 26
Finance	3, 4, 7, 8, 10, 21, 22, 25, 26
Shopping	3, 7, 8, 10, 25, 26
Auto & Vehicles	7, 8, 10, 12, 25, 26
Communication	3, 4, 6, 7, 8, 9, 10, 21, 25, 26
Music & Audio	9, 10, 25, 26
Food & Drink	7, 8, 25, 26

4) Normative approach: Normative approach can help developers to identify whether the permission is indispensable for the App. Permission mechanism restricts the permission usage of Apps. However, this restriction is decided by developers but not users. As a consequence, the security of an App depends on the users' trust in developers.

In a word, Android permission mechanism is essentially an access control mechanism, with the aim to restrict Apps from accessing important functionality of a smartphone such as telephony, network, camera and GPS location [9]. Every permission name is an unique identifier to one kind of system resources, for example, *android.permission.INTERNET* and *android.permission.CAMERA* stand for accessing the Internet and the camera separately. An App must declare the permission it wants to obtain using permission tags in *AndroidManifest.xml* file, otherwise the operation of corresponding functionality will meet an exception.

B. Dangerous Permissions

Android permissions are divided into several protection-levels: normal, signature and dangerous [10]. Only dangerous permissions require user's authorization since dangerous permissions cover areas where the App wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other Apps. For instance, the ability to read the user's contacts is a dangerous permission. If an App declares that it needs a dangerous permission, the user has to explicitly grant the permission to the App. Until the user approves the permission, the App cannot provide functionality which depends on that permission. Due to lack of the knowledge that whether the permission declared by an App is indispensable or not, most users usually authorize all the permission requests. As a result, users' private information useless to the normal operation of the App is stolen unknowingly because of the App's overprivileged behavior.

III. Methodologies

A. Overall Methodology

If an App wants to perform some specific functionalities, it must access the related permissions. Therefore, the permission list implies the App's function and behavior. Google Play divides Apps into 32 categories and those Apps belonging to the same category realize similar function, thus Apps of the same category have similar permissions. According to this, we develop a tool to detect App's overprivileged permission. Our research focuses on the dangerous permissions highly associated with user privacy, and the information of latest dangerous permissions are listed in Table I. We selected 17 categories of which the Apps are wildly used by users, and from these categories we crawl a great number of APK (Android Package) files. After extracting the dangerous permissions of these APK files, we make use of FP-growth algorithm to deal with the data and obtain the output results. Finally, the detection tool is developed on the basis of the results.

B. Maximal Frequent Item Set

Frequent item set mining is one of the best-known and most popular data mining methods, which plays an important role in data mining field [11]. In the beginning, frequent item set mining aimed at finding regularities in the purchase behavior of consumers of supermarkets. In particular, it is exploited to identify sets of products frequently bought together. Once such sets are found, the supermarket will optimize the organization of products on the shelves by putting associated products together or conveniently sell bundled products. Even though frequent item set mining is originally developed for market basket analysis, nowadays it is widely applied for almost any task requiring to discover regularities between variables in a given data set.

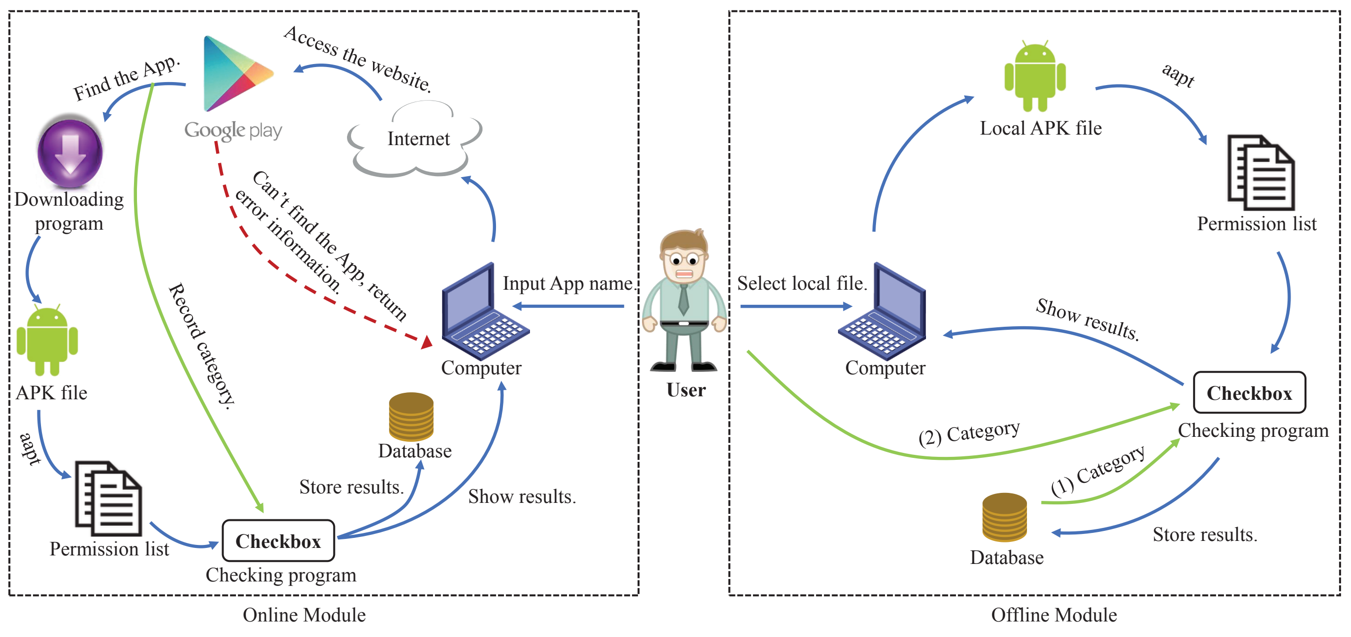


Fig. 2. The tool Droidtector has two modules: online module and offline module. In online module, the tool accesses the Google Play website to search the App with the name input by user. Error information will be returned if the App cannot be found. Otherwise the downloading program will be activated to download the APK file and the App's category will be recorded. Then the permission list of the App will be dumped by aapt, with which the checking program can operate and show user the detection results as well as store the results in database. Offline module needs no connection to the Internet, user should have got the APK file and known its category before detection. The concrete detection process is similar to that of online module, but the checking program will first look up the App's category in the database. If the database has no information about the App, the checking program will operate with the category specified by user. Otherwise, the detection will be completed with the category stored in the database.

Combining with our research, frequent item set mining is the following task: a set $A = \{a_1, ..., a_n\}$ of items stands for the set of Android dangerous permissions, a database $T = \{t_1, ..., t_m\}$ of transactions in which each transaction represents the set of dangerous permissions declared by an App of a certain category. The term *item set* refers to any subset of A, so each transaction is an item set. Given a user-specified minimum support $s \in \mathbb{N}$, an item set $I \subseteq t_k (k \in 1, ..., m)$ is called frequent if and only if the number of transactions containing I is no less than s. The purpose of frequent item set mining is to find all item sets $I \subseteq A$ that are frequent in the database T. Maximal frequent item set refers to the maximum I, and the number of maximal frequent item set may not be unique.

Fig. 1 gives a simple example of frequent item set. Here, we take dangerous permissions for example. Each number corresponds to one dangerous permission listed in Table I. Each transaction is an set of dangerous permissions declared by an App of a certain category, and there are 10 transactions/Apps in all. With a minimum support of $s = 3$, a total of 15 frequent item sets can be found, which are shown in the table on the right in Fig. 1. Of course, all 3-items sets are called maximal frequent item set. We merge all the maximal frequent item sets and we think the dangerous permission outside the merged set is overprivileged under this category.

C. *Frequent Permission-Sets Mining*

Studies of frequent item set mining are acknowledged in the data mining field because of its broad applications [12]. The most popular algorithm publicly available is called FP-growth, which was introduced by [13]. In the algorithm, a data structure called the FP-tree is used to store frequency information of the original database in a compressed form. The FP-growth algorithm needs no candidate generation and only two database scans are required. FP-growth algorithm has been used in the field of mobile application successfully. In [14] authors use FP-growth algorithm to mine the permission correlation between cognate malicious softwares and in [15] the algorithm is applied to understanding the interdependency of mobile applications usage.

To construct App datasets, we gathered a collection of 3767 Android Apps of 17 different types from Google Play. Since Google Play is the official Android market and examines the uploaded App more strictly than the third market, experiments based on the Apps from Google Play are more accurate and convincing. The App in Google Play can be scored by users and the full score is 5. To make sure the App samples are representative, we choose to download the App whose score is no less than 4 at the same time downloads is no less than 100000.

The tool named aapt (Android Asset Packaging Tool) can be used to extract the App's permissions from the

AndroidManifest.xml file. For example, the function of the command "*aapt dump permissions Facebook.apk*" is to dump the permission list of the Facebook application. We select all the dangerous permissions from the permission list of every application and store them in a file, as an input to FP-growth algorithm. By running FP-growth algorithm, we can get all the maximal frequent permission sets. After merging all the maximal frequent permission sets, the final results are shown in Table II. As previously described, every digit in the table represents one dangerous permission. The permissions in the merged set stand for all the dangerous permissions related to the functions of the Apps under a certain category.

IV. Droidtector Implementation

In this section, we will introduce the modules design and effectiveness verification of Droidtector. The tool is developed in Python and has two modules, one is the online module and another is the offline module. Online module can't execute the detection if the tool doesn't access the Internet while offline module completes the detection without accessing the Internet. By verifying the effectiveness of Droidtector, we prove the correctness of our methods.

A. Online Module

Online module needs the tool to access the Internet. User should input the name of the App he wants to check. Once the detection button is clicked, the tool will access the website of Google Play to search the App. If the App cannot be found, the tool will return error information to user or return those App names containing the string typed by user so as to remind user of correct App name. If the App exists, the tool will record its category, at the same time the downloading program will be activated. After the APK file is downloaded, aapt is used to dump its permission list. Finally, checking program will detect the permissions with category information and permission list. Detection results will be shown to user, meanwhile stored in database. The left of Fig. 2 shows the working process of online module of Droidtector.

B. Offline Module

Compared with online module, offline module doesn't need connecting to the Internet. User should have got the APK file before detection and known the category the App belongs to. The detection process is similar to that of online module. Since database has stored some detection results, during every detection the checking program will first look up the App's category in the database. If the category selected by user is different from that of database, the tool will present the detection results according to the category stored in database and remind user that there are two different categories of the App. If user wants to detect the App with another category, he should click the deletion button to clear the App's information stored in the database and then execute a new detection. Of course, the detection will be carried out with the category specified by user if the database has no information about the App. The operational process of offline module is described in the right of Fig. 2.

```
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission android:name="android.permission.DISABLE_KEYGUARD"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.INTERNAL SYSTEM WINDOW"/>
```

Fig. 3. Partial code of the *AndroidManifest.xml* file of QQmusic.apk

C. Effectiveness Verification

APK decompiling is a process of obtaining the source code from an APK file. Hackers usually decompile an APK file and modify the source code or insert malicious code to achieve their purpose, which may result in negative impact on users. To avoid the occurrence of these cases , most developers adopt code encryption to protect their APK file, which is called shell protection. With the development of shell technology, the APK decompiling is becoming more and more difficult. Sometimes, even though the APK file can be decompiled successfully, the decompiled code cannot be repackaged, which also protects the APK file.

To verify the effectiveness of Droidtector, we select three APK files which can be decompiled and repackaged. We use AndroidKiller to decompile the APK file, then the complete *AndroidManifest.xml* file can be obtained. Take QQmusic.apk for example, partial code of *AndroidManifest.xml* file is shown as Fig. 3. Then, we modify the *AndroidManifest.xml* file, delete the permissions judged overprivileged by Droidtector. The overprivileged permissions of QQmusic.apk are *android.permission.ACCESS_FINE_LOCATION*, *android.permission.ACCESS_COARSE_LOCATION* and *android.permission.RECEIVE_SMS*. Finally, we repackage the modified decompilation code and install the repackaged APK file on our Android smartphone. The App operates normally, indicating the permissions deleted are actually overprivileged.

V. Experimental Results

A total number of 1000 popular APK files were selected as test samples. The downloads of every sample App is no less than 500000. Run Droidtector to detect these APK files, the experimental results show that 479 applications face the problem of permissions overprivilege, accounting for 47.9% of all the sample Apps. The number of sample Apps and overprivileged Apps under each category are given by Table III. In 2011, Stowaway analyzed 940 Apps and found that one-third of them had overprivileged permissions. In 2012, PScout detected that 543 of 1260 Apps were in possession of overprivileged permissions, accounting for about 43.1%. In 2014, Spark-Android found

TABLE III
Numbers of Sample Apps and Overprivileged Apps

Category	Sample Apps	Overprivileged Apps
Sports	60	14
Travel & Local	59	31
Medical	58	30
Books & Reference	58	21
Maps & Navigation	58	26
Weather	60	20
Photography	60	31
Education	59	16
News & Magazines	60	16
Social	60	25
Video Players & Editors	60	36
Finance	37	17
Shopping	69	52
Auto & Vehicles	59	33
Communication	63	29
Music & Audio	60	34
Food & Drink	60	48
Total	1000	479

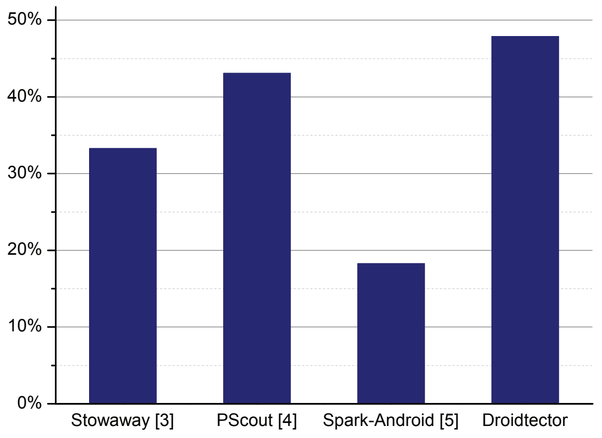


Fig. 4. Results comparison of overprivileged permission test

124 out of 679 Apps, which account for about 18.3%, declared one or more overprivileged permissions. Fig. 4 gives the testing results comparison between our work and previous studies, from which we can see that the proportion of overprivileged Apps are increasing on the whole.

The total number of dangerous permissions declared by 1000 Apps is 4825, about 4.8 dangerous permissions are declared by an App on average. A total number of 1229 dangerous permissions are overprivileged, each App averagely declares about 1.2 overprivileged permissions. This result indicates that one fourth of the dangerous permissions declared by an App is overprivileged.

VI. Conclusion

The open-source feature of Android system makes it easy that any people can release the App developed by himself. Due to the absence of strict development specifications, Android Apps usually declare extra permissions which have nothing to do with their function. We call this issue permission overprivilege. The dangerous permissions defined by Android are related to user's private information, therefore the overprivileged dangerous permission may lead to the leakage of user privacy. To address this problem, in this paper, we present a tool based on frequent item set mining to detect the overprivileged dangerous permission of an App. The tool have two kinds of work modes, one is online mode and another is offline mode. Online mode needs user to input correct App name and offline mode requires user to select a local APK file and specify its category. Finally, we test 1000 Apps crawled from Google Play and find 479 of them are overprivileged. The testing results comparison between our tool and previous research indicates that Apps' permission overprivilege becomes more serious.

References

[1] Market share alert: Preliminary, mobile phones, worldwide, 2q18. [Online]. Available: https://www.gartner.com/doc/3881811/market-share-alert-preliminary-mobile

[2] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of Android Applications' Permissions," in *IEEE International Conference on Software Security and Reliability Companion*, 2012, pp. 45–46.

[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *ACM conference on Computer and communications security*, 2011, pp. 627–638.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *ACM conference on Computer and communications security*, 2012, pp. 217–228.

[5] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

[6] P. Chester, C. Jones, M. W. Mkaouer, and D. E. Krutz, "M-Perm: A Lightweight Detector for Android Permission Gaps," in *IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 217–218.

[7] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanka, "A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework," in *IEEE Second International Conference on Social Computing*, 2010, pp. 944–951.

[8] K. W. Y. Au, Y. Zhou, Z. Huang, P. Gill, and D. Lie, "Short Paper: A Look at SmartPhone Permission Models," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011, pp. 63–68.

[9] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, and M. Conti, "Android Security: A Survey of Issues, Malware Penetration and Defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[10] Android permissions overview. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[11] C. Borgelt, "Frequent Item Set Mining," *Data Mining Knowledge Discovery*, vol. 2, no. 6, pp. 437–456, 2012.

[12] S. Pramod and O. P. Vyas, "Survey on Frequent Itemset Mining Algorithms," *International Journal of Computer Applications*, vol. 1, no. 15, pp. 86–91, 2010.

[13] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 1–12, 2000.

[14] C. Peng, R. Zhao, S. Zheng, J. Xun, and L. Yan, "Android Malware of Static Analysis Technology Based on Data Mining," *DEStech Transactions on Computer Science and Engineering*, 2016.

[15] J. Huang, F. Xu, Y. Lin, and Y. Li, "On the Understanding of Interdependency of Mobile App Usage," in *IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2017, pp. 471–475.