PACS: Pemission Abuse Checking System for Android Applictions based on Review Mining

Jingzheng Wu, Mutian Yang, Tianyue Luo
General Department
Institute of Software, Chinese Academy of Sciences
Beijing, China
jingzheng08@iscas.ac.cn

***Abstract*—The openness and freedom of Android system improve the proliferation of Android applications. According to recent statistics, more than 2.6 million various applications are released in Google Play Store. Unfortunately, due to the limitation of developers' knowledge and the lack of strict development specifications, the quality of the apps can not be guaranteed. This may lead to potential security problems, especially for the over requirements of the apps' permissions, which is called Permission Abuse Problem. Although some previous studies have already analyzed the permission system, investigated the effectiveness of permission model and attempted to resolve the problem, it still needs an effective and practical concentrated method to detect the permission abuse problem. In this paper, we present PACS (Permission Abuse Checking System) based on data and frequent itemsets mining technique to bring an improvement by using the apps' reviews and descriptions. PACS firstly classifies the apps into different categories by mining the apps' meta-data, e.g., the reviews, descriptions, etc. Then, it obtains the maximum frequent itemsets and constructs the permission feature database. Finally, we evaluate PACS on detecting unknown applications of the abused permission. The experiment results show that 726 out of 935 applications, which account for about 77.6%, are suffering from the Permission Abuse Problem. By comparing with the previous tools, PACS has better performances.**

Keywords—Android; abused permission detection; frequent itemsets mining; classify; review mining

I. INTRODUCTION

Mobile services are playing a increasingly prominent role in our daily lives, e.g., communication, information, entertainment, business, education, etc., based on the feature rich applications (app for short). A mobile app is a software application designed to run on mobile devices such as smartphones and tablet computers. According to Statista[1], Google's Play Store incorporated more than 2.6 million applications in December 2016. All of these apps are created by organizations or individual developers, scanned by Google Bouncer, distributed to Google Play Store and downloaded by the end users.

Unfortunately, due to the developers' abilities and lack of strict development specifications, the quality of the apps can not be guaranteed, which may lead to potential security problems. A report finds that 92% of Android's top 500 popular applications are vulnerable to some extent of security or privacy risk[2]. For example, Content Provider Leakage Vulnerability is a typical flaw that caused by the unprotected content provider component[3]. This type of vulnerabilities bring the risk of leaking confidential data stored in un-carefully developed applications to other applications (including malwares) running on the same device [4]. As another example, denial-of-service (DoS) vulnerability is so serious that affects roughly 95% of the Android devices.

In this paper, we mainly focus on the Application Permission Abuse Problem based on review and description mining. As we know permission is the main security mechanism in Android operating system, which is used to enforce access control to the system APIs and applications[5]. All applications have their own statements for permissions in their configuration file "*AndroidManifest.xml*". When an application is installed, the Android system alerts the user to grant permissions the application requests and to choose whether to install the application continually. This is a relatively flexible security policy, which allows the users to know their personal privacy information, such as SMS, contacts, photos, location information, IMEI, camera, micphone, phone states, etc., will be accessed by the installed app. Whenever an app requests more permissions than it actively needs, there is an application Permission Abuse Problem which may introduce security threat. In our research, we found about 77.6% of the apps are suffering from this security problem.

Permission Abuse Problems. Indeed, given the way that today's apps are developed, which are often aiming at extensively user experience improvements and personalized recommendations, it is conceivable that the permission are requested as many as possible, constituting potential security threats to personal privacy. In this situation, the number of privacy leakage is increasing. For example, Soundcomber is a sophisticated malware with limited permissions that is able to leak sensitive high-value data based on its audible surroundings[6]. The key idea of Soundcomber is that an app can borrow the other's permissions that it does not have to by-pass the permission mechanism and leak private information. In this way, the app can avoid the detection of anti-virus software and leak the information no matter that it lacks the proper permissions. On the other hand, permission abuse also isolates the programming principle of the least privilege, intentionally or unintentionally, which seriously affected the

978-1-5090-5569-2/17/$31.00 ©2017 IEEE

code and quality specifications. By checking out the most commonly requested permissions and how they're abused, hacker can exploit them and gain the personal privacy.

Previous studies. Previous work has shown that few people read the Android install-time permission requests and even fewer comprehend them. In fact, even developers are not fully knowledgeable about permissions, and are given a lot of freedom when posting an application to the Google Play Store. Researchers examined the overuse of permissions by applications, and attempted to identify malicious applications through their permission requests or through natural language processing of application descriptions. For example, Pandita etc. focus on permissions for a given application and examine whether the application description provides any indication for why the application needs a permission[7]. They present WHYPER, a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description. Wijesekera etc. instrumented the Android platform to collect data regarding how often and under what circumstances smartphone applications access protected resources regulated by permissions[8]. They found that at least 80% of the participants would have preferred to prevent at least one permission request, and overall, they stated a desire to block over a third of all requests. Researchers have also developed static and dynamic analysis tools to analyze Android permission specifications. However, the Permission Abuse Problems still broadly exist and need to check out.

Permission Abuse Checking System. In this paper, we are motivated by the vision of bridging the semantic gap between the permissions an app requests and what it actually needs. Therefore, in this paper, we present PACS (Permission Abuse Checking System) based on data mining of the frequent item set of app reviews. Firstly, PACS uses machine learning and data mining techniques to classify the apps into different categories by considering app description and users' reviews as new detection conditions. Then, it uses Apriori algorithm to mine frequent patterns of the application within the same category, get the maximum frequent itemsets and construct the permission feature database. Finally, PACS is used to detect the unknown application of the abuse of authority, and the experiment results show that 726 applications are abusing permission among 935 applications.

Contributions. The contributions of the paper are summarized as follows:

- ***New techniques.*** We developed a novel approach, namely PACS, that detects the permission problem for the apps. An innovative classification scheme that is based on mining the apps' descriptions and reviews is designed and used to classify the apps into different categories. An Apriori algorithm that is applied to mining the maximum frequent permission itemsets is also designed and used to label the permissions that a certain category holds. By combining the above two component, PACS detect permissions abuse problems for the new unknown apps effectively.
- ***New findings***. Our study leads to surprising discoveries about the pervasiveness of Android applications. We implemented the PACS and tested on 935 apps obtained from the official app store, finding that almost 77.2% of the apps in the wild are suffering from permission abuse problems.

The remainder of this paper proceeds as follows. Section II presents the background related to this filed. Section III presents the overview of the PACS. Section IV presents the details of PACS along with the techniques used in this work. Section V presents the implementation and evaluation of PACS. Section VI discusses the limitations. Section VII presents the related works. Finally, Section VIII concludes our work.

II. BACKGROUND

A. Android Application

Android is a mobile operating system developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets[9]. Applications (app for short) extends the functionality of the mobile devices, written using the Android software development kit (SDK) and the Java programming language that has complete access to the Android APIs. Android has a growing selection of third party apps, which can be acquired by downloading and installing the APK (Android application package) files, or by downloading them using an app store which allows the users to install, update, and remove apps from their devices. For example, Google Play Store is the primary app store installed on Android devices and offers the apps.

An Android app is built by the essential blocks called app components, which is an entry point, through which the system or a user can enter the app. There are four different types of app components:

Activities. An activity is the entry point for interacting with the users. For example, an email app might have one activity that shows a list of new emails, an activity to compose an email, and another activity for reading emails. Although the activities work together to form a complicated user experience in the email app, each one is independent to the others.

Services. A service is a general purpose entry point for keeping an app running in the background without user interface, used to perform long running operations or to perform work for remote processes. For example, a service might play music in the background while the user is in a different app, or it might fetch data from the network without blocking user interaction with an activity.

Content providers. A content provider manages a shared set of app data that stored in the file system, SQLite database, the web, or on any other persistent storage location the app can access. Through the content provider, other apps can query or modify the data if allowed.

Broadcast receivers. A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system wide broadcast announcements. Because broadcast receivers are another well defined entry into the app, the system can deliver broadcasts even to apps that are not currently running.

B. *Permission of Application*

Permission is the main security mechanism in Android system, which is multifaceted, including API permissions, file system permissions, and IPC permissions[10]. Generally, there is an intertwining of each of these, and the high-level permissions map back to lower-level OS capabilities. In this paper, we mainly focus on the API permission that is used to enforce access control to the system APIs and apps. To determine the app user's authority and supplemental groups, Android processes high level permissions specified in an app package's "*AndroidManifest.xml*" file. App's permissions are extracted from the manifest file on installing by the PackageManager and stored in "*/data/system/packages.xml*". These entries are then used to grant the appropriate authorizes at the instantiation of the app's process.

Permissions use behaviors which aim to capture apps' internal sensitive behaviors on utilizing system resources protected by some permissions. Firstly, an app needs to invoke some Android APIs to request system resources, which is called resource request stage. If the requested resources are protected by some permissions, Android's permission enforcement system will check whether this app has been granted the corresponding permissions at install time. After the resource request stage, system resources may be delivered to the app synchronously or asynchronously, depending on the API used to request resources. Finally, when the requested resources have been delivered to the app, they may be processed by application-specific logic, which reflects the internal behaviors of utilizing sensitive resources.

According to the granularity of the Android permission model, there is an opportunity for developers to request more permissions for their app than it really needs. This behavior may be due in part to inconsistencies in permission enforcement and documentation. Although the developer reference docs describe most of the permission requirements for the given classes and methods, they are neither 100 percent complete nor 100 percent accurate. Researchers have attempted to identify some of these inconsistencies in various ways. For example, in 2012, researchers Reiter etc. attempted to map out the permission requirements for the Android API available in Android Open Source Project, and proposed some interesting conclusions about these gaps. Among some of the findings in this mapping effort, they discovered inconsistencies between documentation and implementation for some methods in the WiFiManager class[11].

In this paper, we also focused on the application Permission Abuse Problem, developed a novel approach PACS to detect this problem, and found about 77.2% of the apps are suffering from the Permission Abuse Problems.

C. *Review of Application*

Users can rate the app on Google Play with a star rating and review. Users can only rate an app once, but they can update their rating or review at any time. Users will leave good reviews for the app for its good functions, beautiful interface, special features, user experiences, etc., where we can identify the category of the app belongs to. A large number of positive play store reviews is also a very important factor in app promoting, which makes excellent designs and code the app perfectly as possible as the developer can.

On the other hand, the app description is part of meta-data fields, which the distributor has to fill in as an app owner when launching it in app stores. This meta data provides users with structured information about the app. An app description thereby is a textual description of app content and functionality that appears at the app store page.

In this paper, we designed an innovative classification scheme that is based on mining the apps' descriptions and reviews, classified the apps into different categories, and used to detect the Permission Abuse Problems.

III. OVERVIEW

The goal of this work is to discover the Permission Abuse Problems in Android apps. The key idea of PRADA is that, *apps located in the same category should have the common permissions, the redundant permissions may be abused.* Naturally, when an app is already classified into a category, the permissions could be checked to determine whether they are abused. Motivated by this, in our approach, we created PACS, as shown in Fig 1, to detect the permission information of unknown apps and output the detection results. PACS is consists of two basic modules, including the training and testing module.

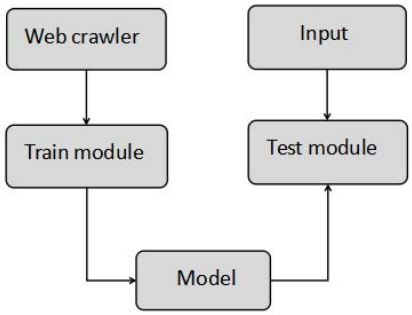

Fig. 1. Architecture of PACS

In the first step, the training module includes a crawler module, a frequent itemsets mining module and data classification module. The crawler gets all the needed data source, including the apps' packages, the apps' descriptions and the apps' reviews from the Google Play Store. The frequent itemsets mining module obtains the permission information for the apps by analyzing APK packages. The Apriori algorithm processes the same category of all apps' permissions iteratively, and calculates the maximum frequent permission set. By processing the descriptions and reviews of the apps, the classifier module constructs a text classifier, which is trained by the testing apps, and an optimized model is obtained after adjusting parameters iteratively.

In the second step, the testing module contains a classification module and a permission detection module. The classification module is used to classify the apps and labels the apps with the proper category based on the descriptions and the

reviews of unknown apps. The permission detection module processes the maximum frequent set of the corresponding category of the unknown apps by combing the classification results. Meanwhile, the permission detection module decompiles the APK file and obtains the permission information from the "*AndroidManifest.xml*" file, compares the maximum frequent itemsets and determines whether there is a Permission Abuse Problem.

IV. METHODOLOGY

A. Crawler module

The crawler module collects the apps' packages, the descriptions and the reviews from the Google Play Store, and stores them into database.

Firstly, the crwaler obtains the initialized URL queue, which is the URL link set of all categories in *Coolapk* official website. Then, it traverses the queue and analyze each corresponding URL link to extract the conditional URL. Thirdly, the crawling process put the new URL to the queue until getting the stop condition. For the specific application web page, it downloads the APK package of the apps, and extracts the name, descriptions, score, reviews, profile and other contents of the apps. The whole processing of the crawler module is shown in figure 2.

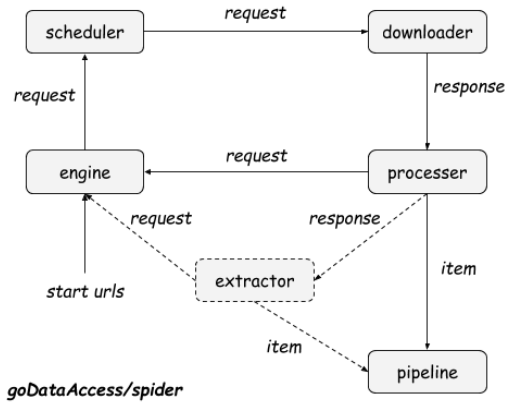

Fig. 2. Flow diagram of crawler module

APK Package Crawler Module extracted 9349 apps from the Google Android Store and divides them into 10 categories based on different function. The apps' categories are shown in Table I.

TABLE I. APPLICATION CATEGORIES

Id	Classification	No. of apps.
01	System tools	1300
02	Desktop plug-in	320
03	Theme landscaping	1553
04	Social chat	521
05	Information reading	765
06	Communication network	755
07	Audio & video entertainment	1340
08	Photography	663
09	Life service	876
10	Utility tools	1256

Descriptions and Reviews. The crawler obtains the complete data source of the apps, records the category ID for each app review and stores them into the database according to the category name and category ID mapping relationship.

B. APK decompile

APK (Android application package) is the format of Android application for distribution and installation, and mainly includes five parts:

- The META-INF directory, used to store the signature information to verify whether the APK is complete.
- The res directory, used to store the resource files.
- AndroidManifest.xml, used to describe of the application name, permissions, etc.
- Classes.dex, Java source code, used to Decompile APK package.
- Resources.arse, a compiled binary resource file.

```
|-- AndroidManifest.xml
|-- META-INF
|   |-- CERT.RSA
|   |-- CERT.SF
|   `-- MANIFEST.MF
|-- classes.dex
|-- res
|   |-- drawable
|   |   `-- icon.png
|   `-- layout
|       `-- main.xml
`-- resources.arsc
```

Fig. 3. APK compositon

For an APK, to get the "*AndroidManifest.xml*" file, it needs to decompile before analyzing. The commonly used reverse tool is APKTool, which can reverse an APK file into Smali source code and XML files inside.

By analyzing the "*AndroidManifest.xml*" file, we find that it includes the following parts:

- <Uses-permission>, used to request authorization.
- <Application>, used to express each app's component and the properties
- <Receiver>, used to receive the changed data etc.
- <Service>, a component which can be run at any time on the background.

PACS mainly analyzes the parameters in the <uses-permission> part, and extracts the permission information from it. Table II lists the names of some extracted permissions and their explanations.

TABLE II. PART PERMISSIONS AND EXPLANATIONS（PREFIX "ANDROID.PERMISSION" IS OMITTED.）

Name	Function
ACCESS_NETWORK_STATE	Get network status
ACCESS_WIFI_STATE	Get WiFi status
INTERNET	Access networks
BLUETOOTH	Use Bluetooth
READ_CONTACTS	Read contact
CALL_PHONE	Dial telephone

C. *Frequent item set mining*

Frequent pattern refers to mining frequent items in the data set. Frequent pattern Apriori algorithm is an original algorithm for mining frequent item sets of Boolean association rules proposed by AgrawalR etc. We mine the permission association of the dataset based on the frequent pattern Apriori algorithm, and respectively build the permissions association data set to detect unknown apps' permissions.

We give an example to illustrate the specific algorithm process. Supposing that there are four apps under each category, and each app lists the permissions data, as shown in Table III. Support=P (AB) refers to the probability of event A and event B to occur at the same time. In order to simplify the calculation, we hypothesize that the minimum support degree is 50%. For each iteration, the algorithm will generate a number of candidate sets. If the relative support of the item set I meets the predefined minimum support threshold, then I is a frequent item set.

TABLE III. PART APPLICATION DATABASE UNDER ONE CATEGORY

TID	Permission set
APK.01	CALL_PHONE,READ_CONTACTS, INTERNET, WAKE_LOCK
APK.02	VIBRATE, READ_CONTACTS, WAKE_LOCK
APK.03	INTERNET, WAKE_LOCK, CALL_PHONE
APK.04	INTERNET,WAKE_LOCK, READ_CONTACTS,CALL_PHONE

The algorithm in Table IV describes all the iterative steps. In the first iteration, there are 4 sets with more than 50% relative support, which are the frequent itemsets in 1- sets. In the second iteration, there are 6 frequent 2-itemsets with their support degree more than minimum support, which are composed of two 2-itemsets of frequent item sets of 1- items. The third iteration has three 3-itemsets with their support more than min_sup, and all are selected as the frequent item set. The fourth iteration only has one group of itemsets that satisfy the minimum support threshold, which is frequent 4-itemsets, and there are no frequent itemsets Y, making (CALL_PHONE, READ_CONTA-CTS, INTERNET, WAKE_LOCK) in Y, so this group is the maximum frequent itemsets for the certain permissions.

TABLE IV. PART APPLICATION DATABASE UNDER ONE CATEGORY

Iteration	Itemsets	Counting	Sup./%	Fre.Set
1	CALL_PHONE	3	75	**Y**
	READ_CONTACTS	3	75	**Y**
	VIBRATE	1	25	**N**
	INTERNET	3	75	**Y**
	WAKE_LOCK	4	100	**Y**
2	CALL_PHONE, READ_CONTACTS	2	50	**Y**
	CALL_PHONE, INTERNET	1	25	**N**
	CALL_PHONE, WAKE_LOCK	3	75	**Y**
	READ_CONTACTS, INTERNET	2	50	**Y**
	READ_CONTACTS, WAKE_LOCK	2	50	**Y**
	INTERNET, WAKE_LOCK	3	75	**Y**
3	CALL_PHONE, READ_CONTACTS, INTERNET	2	50	**Y**
	CALL_PHONE, READ_CONTACTS, WAKE_LOCK	2	50	**Y**
4	CALL_PHONE, READ_CONTACTS, INTERNET, WAKE_LOCK	2	50	**Y**

It needs to ensure the reliability of the training set when using the Apriori algorithm to iterate the pattern of permissions' frequent itemsets. Because the training APK may affect the accuracy of the frequent itemsets for a certain extent, we should refine the apps based on the apps' scores. The score on *Coolapk* website is ranging from 1 to 5, we choose the APK with score of 4 or above according to the training set and the narrow the testing set size.

D. *Application of classification module*

The apps in the different categories usually have much difference than the those in the same category. Take the category of "photography" as an example, the permissions applied by the apps usually include reading from external storage, the camera permission, recording, vibration, network status etc. On the other side, the permissions of accessing to the network, phone, calling, receiving text messages, reading the message content etc. are mostly applied by the apps in the " network news" category. Therefore, when determining the Permission Abuse Problems, classifying the unknown apps and checking permissions labeled by the category is a reliable and effective method.

Support vector machine (SVM for short) is a supervised learning model, which is usually used for pattern recognition, classification and regression analyzing. Vapnik etc. present an optimum design criterion of linear classifier after years of theory learning by extending mapping of the low dimension to high dimensional space from linearly separable form to inseparable form and even extendeing to the use of nonlinear function. In addition to performing linear classification, SVM can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces. This classifier is called support vector machine.

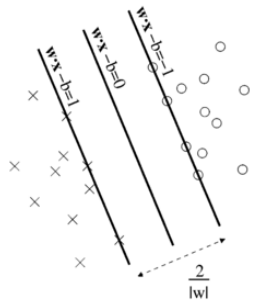

Fig. 4. Separating hyperplane

V. System Implementation and Verification

A. System implementation

We used Python language to build PACS. The crawler, especially the HTML analysis module, is built based on the BeautifulSoup package. We used the open source project AAPT to reverse the apps and extract permissions information for the APK decompiler module. Finally, the system interface of PACS is implemented based on Tkinter Python development.

B. Experimental Analysis

The data were taken from the *Coolapk* website, which contains 9349 apps of the APK package, the apps' details, scores, descriptions and reviews data. The 9349 applications are divided into 10 categories. We selected 1/10 data from each category as testing data to verify the effectiveness of the system. A total number of 935 experimental samples were randomly selected, input into the PACS, and processed to detect the permissions abuse problems. The expreiment results show that a total number of 726 applications are suffering from the Permission Abuse Problems, accounting for about 77.6% of all the apps.

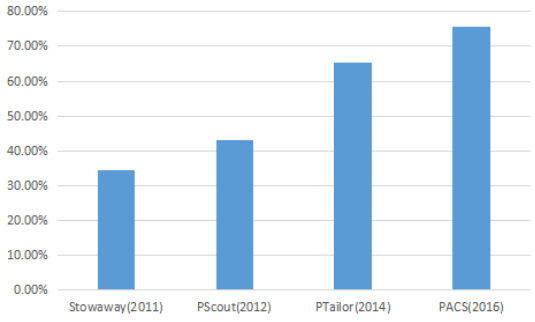

Fig. 5. Comparison of the testing results of the permissions abuse

In 2011, Felt AP etc. used the Stowaway tool to analyze 940 apps, and detected about 34.4% of them are permission abuse. In 2012, Au etc. used PScout to analyze 1260 applications, and found that 543 applications in the presence of permission abuse, accounting about 43.1%. In 2014, Xiao-long Bai etc. used PTailor to tailor the collection of 1246 applications, and found that there are 811 applications in the phenomenon of permissions abuse, accounting for about 65.1%. Bartel etc. got the conclusion that the proportion of the application permissions abuse will continue to increase with the time going on by analyzing the different versions of the application. In Fig. 5 we compare PACS with the previous work, and found that the permissions abuse problems are still increasing, which is consistent with Bartel's conclusions.

C. Experimental verification

To make sure of our conclusions, we take the following experimental verifications: we randomly select the apps which are labeled with the permissions abuse after detecting, then reverse them into Java code, search API and permissions' mapping relations, and analyze the code and the requirements to confirm whether the apps are of the permissions abuse ones.

We select 5 apps from the results of PACS, and list them in Table V as follows.

TABLE V. Part Testing Result of APK

Apk name	No. of permissions abuse
dopool.player.apk	5
cn.ersansan.kichikumoji.apk	4
cn.am321.android.am321.apk	6
cn.etouch.ecalendar.apk	20
com.aikan.apk	6

Take dopool.player.apk as an example, to verify whether it a real one, we inspect and analyze it manually.

Decompile APK to generate java source code. The Apk file is an application of the package, which can be decompiled to get java source code. First, use the APKTool to decompile the package. After decompiling, class.dex files, which are java file compiled through the DX tool package are obtained in the decompiled folder. Then, an open source tool, called dex2jar which is used to decompile class.dex and generate class-dex2jar.jar, is introduced. Then, we can get the Java source code from the jar file by using jd-gui tool. The final source code results are shown in Fig. 6.

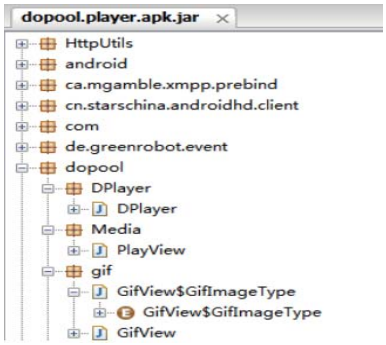

Fig. 6. The source code for the decompiled Dopool

PACS system for abuse detection. We run PACS to detect the selected apps, and the testing results are shown in Table VI.

TABLE VI. Results of Dopool in PACS (Prefix "Android.permission" is omitted)

Result	List of the Permissions Abuse

Abuse	SEND_SMS
	RECEIVE_USER_PRESENT
	READ_CALENDAR
	CALL_PHONE
	WRITE_CALENDAR

Analysis of PACS's results. Firstly, we give the relations between the permissions abused and the corresponding APIs in the list. Some of the relations of Dopool are listed in Table VII.

TABLE VII. PART OF MAPPING BETWEEN PERMISSIONS AND API

Name	Android API
SEND_SMS	sendTextMessage
	sendMultipartTextMessage
	sendDataMessage
CALL_PHONE	enforceCallPermission
	endCall
	call
	dialRecipient

From Table VII, we can see that SEND_SMS permission is related to three APIs, which are "sendTextMessage", "sendMultipartTextMessage" and "sendDataMessage". When searching these APIs in the Dopool source code, there is no calls to them. We got the same results when it comes to CALL_PHONE permission. Therefore, these two permissions can be determined as the abused permissions.

VI. DISCUSSION

Although some previous studies focus on the mapping relationship between Android APIs and the permissions, the mapping is still far away from perfect. For example, the most complete API mapping relation in PScout is only finding 101 mapping relationships in Android 4.1.1 which has 180's in total. This is shown that the API coverage rate is relatively low.

Secondly, the previous studies are based on an assumption that as long as the code calls the relevant API, the permissions of the application is normal. However, by analyzing the apps' source code, we found that the current developers that are in different levels use APIs, which is not actually need to apply or should not be used, arbitrarily in their code.

In this paper, we propose a new permissions abuse checking system, called PACS, based on the permissions set and reviews set mining, avoiding using the mapping relation of API and permissions. We classify the similar apps into the same category by mining the descriptions and reviews, build the corresponding permission model to detect the permissions abuse problems.

VII. RELATED WORK

The permission mechanism in Android security framework manages the access of third-party applications to privacy and security relevant parts of APIs. Some previous studies already analyzed the permission system, investigated the effectiveness of permission model and attempted to resolve the overprivileged issue[12,13]. Similar to our work, some researchers have investgated the correlation between permissions and descriptions. However, even if permissions and descriptions do not correlate, our solution can bring an improvement by using the users' reviews.

Felt etc. proposed a scheme for the detection of permissions abuse through the mapping relationship between permission and API, and built a tool named Stowaway[14]. They analyzed the Android 2.2.1 API version, and have the result that there were 1259 APIs, which need to apply for permission before calling the API, while there were only 78 APIs whose corresponding permission have been defined. They randomly selected 40 from 940 applications running and evaluated on Stowaway, and got that 18 apps could be classified into permission abuse. They believed that the reason of abuse is the lack of mapping relationship from API to permission and pointed out that the root cause of the abuse is the existing of the errors and incomplete of Android API documents.

Au etc. built PScout, a version-independent tool to extract the permission specification and attempted to answer some key questions about Android's permission system[15]. They found that 1) extracting an accurate permission specification with light-weight call-graph analysis, 2) optimizing that analysis with domain-specific information to selectively refine parts of that call-graph with flow-sensitive analysis, and 3) using a uniform abstraction for permission checks are possible. They analyzed the source code of Android 2.2 version to 4.0 version and got a more complete mapping between the API and permissions, and they also proposed PScout static analysis tools to deal with the permission abuse, finding that there are 543 applications have abuse problems in the sample of 1260 applications.

Moonsamy etc. studied the Android permission system as the smartphone platform makes use of permissions to regulate access to system resources and users' private information[16]. They considered the implications of incorporating used permissions in permission patterns and determined their usefulness in contrasting between clean and malicious applications. They also proposed an efficient pattern mining method to generate a set of emerging contrast permission patterns for our clean and malware dataset. By analyzing a dataset of 1227 clean applications. They found out that there is a discrepancy in the official documentation that allows for application with API level 3 or level to implicitly inherit certain permissions, although they are not declared in the "*AndroidManifest.xml*" file.

Zhang etc. presents VetDroid, a dynamic analysis platform for reconstructing sensitive behaviors in Android apps from a novel permission use perspective[17]. the first approach to perform accurate permission use analysis to vet undesirable behaviors. VetDroid features a systematic framework to effectively construct permission use behaviors. It is shown to be able to clearly reconstruct malicious behaviors of real-world apps to ease malware analysis. When they apply VetDroid to 1,249 top free apps in Google Play, it can also assist in finding information leaks, analyzing fine-grained causes of information leaks, and detecting subtle vulnerabilities in regular apps.

Pandita etc. investigated the correlation between permissions and descriptions[7]. They focused on permissions for a given application and examine whether the application

description provides any indication for why the application needs a permission. They presented WHYPER, a framework using Natural Language Processing techniques to identify sentences that describe the need for a given permission in an application description. In the evaluation, WHYPER achieved an average precision of 82.8%, and an average recall of 81.5% for three permissions (address book, calendar, and record audio) that protect frequently used security and privacy sensitive resources. Their results demonstrated great promise in using NLP techniques to bridge the semantic gap of user expectations to aid the risk assessment of mobile applications.

Researchers have designed systems to analyze and dynamically block runtime permission requests. We also designed PACS to detect Permission Abuse Problems by mining the apps' descriptions and reviews, and found that most apps, about 77.2% of the apps, are suffering from Permission Abuse Problems. Comparing with the existing tools, PACS is much more effective and practical.

VIII. CONCLUSION

Nowadays, more than 2.6 million apps are created by the various organizations and individual developers. Without strict development specifications, the quality of the apps can not be guaranteed, especially for the apps' permission requirements, which may induce Permission Abuse Problems. This problem is increasing at an higher grow rate, and may bring serious security risks. Therefore, in this paper, we focused on the Permission Abuse Problems, and designed and implemented abuse detection system PACS based on a set of frequent itemsets mining. PACS uses machine learning and data mining techniques to classify the apps into different categories by considering app description and users' reviews as new detection conditions. It mines the frequent patterns of the application within the same category, obtains the maximum frequent itemsets and constructs the permission feature database. In the evaluation, PACS finds 726 out of 935 apps are abusing permission. Comparing with the previous tools, PACS is much more effective and practical.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable work. This work is supported by NSFC No.61303057 and Project No.2012ZX01039-004.

REFERENCES

[1] Number of available applications in the Google Play Store from December 2009 to December 2016. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[2] Hossain Shahriar, Hisham M. Haddad. Content Provider Leakage Vulnerability Detection in Android Applications. Proceedings of the 7th International Conference on Security of Information and Networks. Glasgow, Scotland, UK; ACM. 2014: 359-366.

[3] Jingzheng Wu, Zhifei Wu, Mutian Yang, Tianyue Luo, Yanjun Wu, Yongji Wang. Quality, Reliability and Security Study of Vendor Customized Android Applications. 22nd Network and Distributed System Security(NDSS'15). San Diego, CA, USA. 2015.

[4] Christoph Treude, Martin P. Robillard. Augmenting API documentation with insights from stack overflow. Proceedings of the 38th International Conference on Software Engineering. Austin, Texas; ACM. 2016: 392-403.

[5] Gradeigh D. Clark, Swapnil Sarode, Janne Lindqvist. No time at all: opportunity cost of Android permissions (invited paper). Proceedings of the 3rd Workshop on Hot Topics in Wireless. New York City, New York; ACM. 2016: 12-16.

[6] Kehuan Zhang Roman Schlegel, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS'11). San Diego, CA, USA. 2011.

[7] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, Tao Xie. WHYPER: towards automating risk assessment of mobile applications. Proceedings of the 22nd USENIX conference on Security. Washington, D.C.; USENIX Association. 2013: 527-542.

[8] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, Konstantin Beznosov. Android permissions remystified: a field study on contextual integrity. Proceedings of the 24th USENIX Conference on Security Symposium. Washington, D.C.; USENIX Association. 2015: 499-514.

[9] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, Patrick Traynor. *droid: Assessment and Evaluation of Android Application Analysis Tools. ACM Comput Surv, 2016, 49(3): 1-30.

[10] Daibin Wang, Haixia Yao, Yingjiu Li, Hai Jin, Deqing Zou, Robert H. Deng. CICC: a fine-grained, semantic-aware, and transparent approach to preventing permission leaks for Android permission managers. Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. New York, New York; ACM. 2015: 1-6.

[11] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, Tung Thanh Nguyen. Learning API usages from bytecode: a statistical approach. Proceedings of the 38th International Conference on Software Engineering. Austin, Texas; ACM. 2016: 416-427.

[12] Vincent F. Taylor, Ivan Martinovic. SecuRank: Starving Permission-Hungry Apps Using Contextual Permission Analysis. Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices. Vienna, Austria; ACM. 2016: 43-52.

[13] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. Proceedings of the 17th ACM conference on Computer and communications security. Chicago, Illinois, USA; ACM. 2010: 73-84.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, David Wagner. Android permissions demystified. Proceedings of the 18th ACM conference on Computer and communications security. Chicago, Illinois, USA; ACM. 2011: 627-638.

[15] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, David Lie. PScout: analyzing the Android permission specification. Proceedings of the 2012 ACM conference on Computer and communications security. Raleigh, North Carolina, USA; ACM. 2012: 217-228.

[16] Veelasha Moonsamy, Jia Rong, Shaowu Liu. Mining permission patterns for contrasting clean and malicious android applications. Future Generation Computer Systems, 2014, 36(2014): 122-132.

[17] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security. Berlin, Germany; ACM. 2013: 611-622.