ASSISTING ANDROID APPLICATION DEVELOPERS WITH AUTOMATIC RECOMMENDATION OF PERMISSIONS

A Thesis by

Md Yasser Karim

Bachelor of Science, Bangladesh University of Engineering and Technology, 2009

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

December 2015

ASSISTING ANDROID APPLICATION DEVELOPERS WITH AUTOMATIC RECOMMENDATION OF PERMISSIONS

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science with a major in Computer Science.

_____

Huzefa Kagdi, Committee Chair

_____

Kaushik Sinha, Committee Member

_____

Ehsan Salari, Committee Member

DEDICATION

To my parents, my brothers,
and my dear friends.

ACKNOWLEDGEMENTS

First of all I would like to express my gratitude to my supervisor and thesis committee chair, Dr. Huzefa Kagdi, whose expertise, understanding, and patience, added considerably to my graduate experience. His vast knowledge and research skill in the area of software maintenance and evolution, and his expertise in research publications are really admirable. Without his supervision and constant help this thesis would not be possible. I really feel lucky to get a friendly adviser like him. I would also like to thank Dr. Massimiliano Di Penta from University of Sannio, Italy. He provided ample guidance and suggestion related to this thesis work. His extensive knowledge in this research area enlightened me in the right direction. I would also like to thank other thesis committee members who put their valuable time to review my thesis and make it a successful one. My sincere gratitude goes to the faculty members that I came across during my entire graduate program at Wichita State University. Last but not least, the extraordinary support and prayers I received from my parents, family, and friends have been the most important part of my journey throughout my graduate studies. I would like to extend my special gratitude to them for supporting me in every step of the way.

ABSTRACT

The Android platform enjoys the largest market share in mobile applications (84% at the end of 2014). Applications for Android are written mainly in Java and commonly referred to as 'apps'. The explicit permission mechanism imposes a few additional requirements on app developers. App developers not only need the knowledge of which APIs to use to implement the features of an app, but also of any required permissions. On one hand, a missed permission (i.e., a false negative) could result in malfunctioning of the feature (e.g., app crashes). On the other hand, unnecessary permissions (i.e., false positives) could introduce false dependencies, user confusion, marketplace rejection, or even expose risks (e.g., malware threats). Unfortunately, API from/to permission traceability is not necessarily explicitly documented.

The thesis presents an approach, named *ApMiner*, which relies on association rule discovery to identify co-occurrence patterns of Android APIs and permissions. Based on the usage of APIs and permissions in other apps published in a marketplace, the approach is able to learn and help a developer of a new app to recommend the permissions to be added, given the APIs being used. *ApMiner* has been empirically evaluated on 600 apps from F-Droid, a marketplace for free and open source apps. We compared *ApMiner* with the state-of-the-art approaches *Androguard* and *PScout*, which rely on traditional static and dynamic analyses to recommend permissions. Results show that *ApMiner* has statistically significant and substantial precision gains (about 1.5 to 2 times in reducing false positives) over the compared approaches, while keeping a similar or slightly better level of recall (i.e., a measure of false negatives). Overall, our findings suggest that a mining based approach could offer much improved effectiveness in automatically recommending permissions in developing (new) Android apps.

TABLE OF CONTENTS

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

Introduction

The Android platform enjoys the largest market share in mobile applications (84% at the end of 2014, source IDC[1]). Applications for Android are written mainly in Java and commonly referred to as `apps'. Due to the large proliferation of Android apps and the limited verification performed by publishing stores (e.g., the iOS platform has a stricter, manual verification for apps published on the *Apple* store), Android employs a unique permission mechanism that is aimed at preventing an app from accessing resources without the consent of the device's owner. No app, by default, has privileges to perform any operations that would adversely impact other applications, the operating system, or the user.

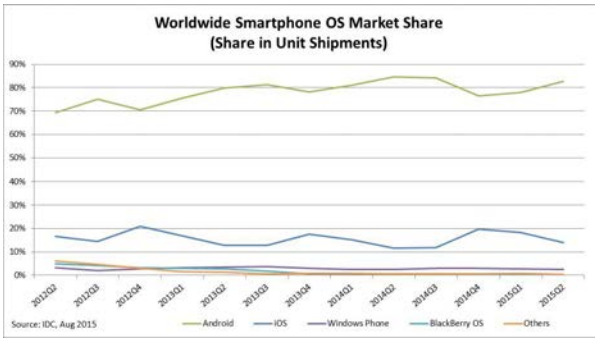

Figure 1.1: Market Share of Smartphone OS

The explicit permission mechanism imposes a few additional requirements on app developers. App developers not only need the knowledge of which APIs to use to implement the features of an app, but also of any required permissions. Every app they develop must have the specification file `AndroidManifest.xml`. This manifest file contains the essential information about the app, including API compatibility versions, components (e.g., *Activities* and *Services*), and permissions to access resources. In order to install an app on a device, the user must explicitly grant all the permissions, which are specified by the developer in the manifest file. There are 151 system-level

[1] http://www.idc.com/prodserv/smartphone-os-market-share.jsp

permissions available and over 4,000 classes at the API level 21 in Android. These permissions are required to use protected APIs for phone hardware resources (e.g., storages and WiFi) or privacy sensitive information available on the phone (e.g., contacts and location). On average, an app on the *F-Droid* platform has five permissions.

It is the developer's responsibility, which should be carefully exercised, to request only the minimal set of necessary permissions, according to Android's recommended best practices[2]. The official Android documentation for API classes and permissions is an excellent source to learn how to properly use permissions. It describes them very well individually; however, it was deemed incomplete [27, 32], and the links between API classes and permissions are not always explicitly documented. The proper specification of permissions is not merely yet another idiosyncrasy of yet another domain-specific platform. They are crucial to privacy and security aspects of users. There is empirical evidence of permission misuse even for the most popular permissions [31]. Apps with too many permissions may exhibit suspicious malware characteristics [33]. On one hand, a missed permission (i.e., a false negative) could result in malfunctioning of the feature (e.g., app crashes). On the other hand, unnecessary permissions (i.e., false positives) could introduce false dependencies, user confusion, or even expose risks (e.g., malware threats).

Unfortunately, popular Android IDEs of the calibre of *Android Studio* and *Eclipse* do not offer much help in terms of recommending the relevant API classes to a permission, or vice versa (i.e., no ``autocomplete'' from API class to permission in the manifest file). There have been recent efforts on automatically tracing API elements from/to specific permissions. One noticeable outcome of these efforts is Stowaway [27], which extracts the APIs being used in an app through static analysis. It relies on a permission map, built through dynamic analysis of the Android OS/stack. Unfortunately, the currently available Stowaway's permission map is outdated[3]. To the best of our knowledge, the most accurate and up-to-date tool is *PScout* [32], a tool that maps permissions to

[2] `https://developer.android.com/training/articles/security-tips.html#RequestingPermissions`

[3] http://www.android-permissions.org

APIs based on the static analysis of the Android OS. *Androguard*[4] embeds *PScout*'s methodology and enables API to permission recommendations for a given app. In summary, these approaches are based on *program analysis* of Android framework and/or OS.

Thesis contribution. We present an approach, named *ApMiner*, which recommends permissions to be used for given Android APIs[5] (classes) by combining static analysis with mining of other apps' permission usage. To this aim, *ApMiner* firstly analyzes other existing apps, by extracting their API classes and the permissions used together from their Android packages (e.g., `.apk` binary files). Then, by using association rule discovery [1], it learns API-to-permission rules. Finally, by analyzing the APIs being used by the app under analysis, it recommends likely permissions for it.

We have evaluated *ApMiner* on 600 Android apps from the *F-Droid* store, and compared it with state-of-the-art permission recommenders, i.e., *Androguard* and *PScout*. Results of the empirical evaluation indicate that, in terms of precision *ApMiner* achieves significantly better results than the alternative tools, at comparable---and slightly higher---levels of recall. Besides the quantitative evidence highlighting the *ApMiner*'s performance, we also report qualitative evidence, illustrating how *ApMiner* can recommend permissions for cases in which the Android documentation is incomplete. Although there are existing approaches that identify whether the app's behavior in terms of permissions requested deviates from what was declared [16], to the best of our knowledge, ours is the first approach that recommends API-to-permission relationships based on their *usage mining* in other apps.

Thesis structure. Section 2 provides a description of the Android permission mechanism. Section 3 describes the proposed approach. The empirical study and its results are described in Section 4. Section 5 discusses the threats to validity. After a discussion of related work (Section 6), Section 7 concludes the paper and outlines directions for future work.

[4]https://github.com/androguard/androguard

[5]in the discussion, our reference to API means API classes/interfaces unless stated otherwise

CHAPTER 2

Background

Android is a complete package of software for mobile devices. It consists of four components in a hierarchy. At the bottom, there is a Linux kernel, which provides core system services such as security, memory management, process management, network connectivity, and driver models. The next layer is a combination of two components, the Dalvik virtual machine (DVM) for executing Dalvik Executable format (.dex) files and a set of core libraries for system, graphics, database, and web browsing. The third component is an application framework providing a set of services and systems fully supporting rich application development. Above all the components, there is a set of basic applications such as a phone application included in Android SDK.

2.1 Dex code

Dalvik Virtual Machine is a register-based machine. A frame consists of a specific number of registers, a pointer to a .dex file, and data for execution. It employs a sand box architecture in which an application has its own process running on an instance of DVM. The advantage of the architecture is that a failure of an application does not impact other applications or the system. An Android application is written in Java and translated into Dalvik executable (dex) format. During translation to dex, separated and indexed constant pools and methods are derived from original Java classes. The pools store strings, types, fields, and methods. At runtime, the constant pools can be referenced by instructions as needed. Arguments to a method correspond to registers in the method invocation frame. Some assignment and array operation instructions perform for more than one type. For example, a 32-bit move instruction works for both integer and floats. Instructions are variable length.

2.2 Application components

There are four different types of application components used to implement an Android application. Each component has distinct purpose and conditions.

Activities: An activity controls a single window connecting to a user interface.

Services: A service runs in the background to support long-running operations. It does not provide a user interface but can be started by another component.

Broadcast receivers: A broadcast receiver is a component that receives and responds to messages from other applications.

Content provider: A content provider provides a way of accessing data repositories on the phone for both retrieving and manipulation. To use a content provider, it has to be associated with a designated API. A basic content provider class is bundled in the Android SDK and provides services for built-in data repository such as text messages, audio, video, images, contact information, and so on. Application developers can also define custom content providers; these often connect to private data repositories such as a table in SQLite database. This paper focuses on reverse engineering dex code in order to discover how applications access databases, either direct SQLite queries or via content providers.

2.3 The Manifest File

Every application must have an AndroidManifest.xml file (with precisely that name) in its root directory. The manifest file presents essential information about the app to the Android system, information the system must have before it can run any of the app's code[9]. Among other things, the manifest does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application , the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement

each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.

- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

2.4 Android Permission Scheme

Android is designed such that each app has its own sandbox (e.g., process, virtual machine, and data storage). It is based on the principle of least privilege: access beyond certain elementary (default) resources is controlled through explicit mechanisms, such as permissions. For instance, if an app needs access to hardware functions (e.g., cameras, GPS, and Network) and/or sensitive data (e.g., contact information), it must request the appropriate permissions. The user must grant the permissions collectively an app needs, so that it can be installed on a device (e.g., when an Android binary package, an *.apk* file, is installed). In summary, Android has an explicit model of application security, in which sensitive hardware and data access is always routed through a layered Android software stack (Applications to APIs to Android OS to Linux Kernel). Fig. 2.1

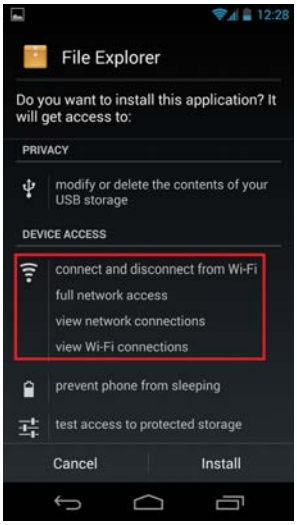

Figure 2.1: An app asking the user to grant the shown permissions for its successful installation.

shows an app called *File Explorer* from *F-Droid*[1] when a user begins its installation, i.e., an end user perspective. Among the several permissions the *File Explorer* app requests is *Wi-Fi* (highlighted within a red box)[2]. Clicking on the *Install* button, the user grants all the requested permissions on their device.

From the development perspective, Android requires that all these permissions must be specified in the manifest file of the *File Explorer* app. The *AndroidManifest.xml* file must reside in the app's root directory. Currently, Android permits only static specification of permissions (granted once during installation) and they cannot be changed dynamically at runtime. It is the app developer's responsibility to specify the appropriate permissions in this manifest file. Fig. 2.2 shows the developer's specified permissions in the *AndroidManifest.xml* of the *File Explorer* app.

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Figure 2.2: An excerpt of the permissions in the app's AndroidManifest.xml.

Android provides a predefined set of system-level permissions. Developers must ascertain

[1]https://f-droid.org
[2]http://source.android.com/devices/tech/security/index.html

that the features they implement using Android API elements have the corresponding appropriate permissions. The API documentation and current IDE support are not always much of a help in determining the specific permissions needed for specific API elements (and consequently for the app's features) and vice versa. Fig. 2.3 shows an example of usage of API element *WifiManager* (a Java class) in the *File Explorer* app. The code elements highlighted in red justify the specification and request for the *ACCESS_WIFI_STATE* permission (highlighted in the red box in Fig. 2.2).

```
import android.net.wifi.WifiManager;
import android.net.wifi.WifiManager.WifiLock;
...
public class FTPServerService extends Service implements Runnable {
    public static InetAddress getWifiIp() {
        ...
        WifiManager wifiMgr
        = (WifiManager) myContext.getSystemService(Context.WIFI_SERVICE);
        if (isWifiEnabled()) {
            int ipAsInt = wifiMgr.getConnectionInfo().getIpAddress();
```

Figure 2.3: Android WifiManager class needing the ACCESS WIFI STATE permission.

Uncovering and documenting the developer knowledge of the traceability between API elements and permissions could potentially help other developers in building their apps. The usage knowledge in the *File Explorer* app can be captured in the form of the rule *WifiManager* $\Longrightarrow$ *ACCESS_WIFI_STATE*. For example, the developers of the *FBReader* (ebook reader) and *MPDroid* (music streaming) apps could have used it, as this usage was also found in them.

CHAPTER 3

Approach

Our *ApMiner* approach is based on the premise that if APIs and permissions frequently co-occur in existing (other) Android apps deployed on a marketplace (e.g., *Google Play* or *F-Droid*), then they can be recommended to developers of a new app. We use the frequent itemset mining technique [1] to generate association rules to codify the API-permission usage. As such, they are considered as a common practice. We first explain how frequent itemset mining can be used to identify API-permission rules. Then, we provide details about the various components of *ApMiner*.

3.1 Identify API-Permission Rules by Frequent Itemset Mining

Frequent itemset mining [1] is a data mining technique widely used for affinity analysis (e.g., market basket analysis) to discover co-occurrence relationships among entities in events (i.e., transactions). Given a set of n transactions, $T = \{T_1, T_2, ..., T_n\}$ and a user specified minimum support σ_{min}, this technique produces a set of m frequent itemsets, $E = \{E_1, E_2, ..., E_m\}$. Each T_i in T is a set of entities. Each E_i in E is a set of entities and it is found in σ number of transactions, where $\sigma(ec) \geq \sigma_{min}$, i.e., it is frequent. In our context, the set of transactions T corresponds to the set of existing apps fed for training. The entities correspond to the APIs and permissions used in the apps in T. That is, a transaction is a set of APIs and permissions used in an app. Therefore, our frequent itemsets are the sets of APIs and permissions found to co-occur in T.

3.2 Extracting and Mapping Permissions and APIs

The prerequisite for applying frequent itemset mining is the formation of the transactions, T, which are composed of entities. These transactions form our training set. In our context, each transaction corresponds to an existing app, and entities refer to its APIs and required permissions. The permissions are specified in the manifest file, i.e., `AndroidManifest.xml`, which is located at the root level of the app. The APIs are found in the Java source code files, via the use of the *import*

statement. The extraction of `AndroidManifest.xml` file and APIs is straightforward for open source apps, such as those hosted on the *F-Droid* forge. The permissions can be easily obtained from the off-the-shelf XML processing tools for parsing and querying the `AndroidManifest.xml` file. More specifically, the permissions are extracted from the <*uses-permission*> tags (see Fig. 2.2). The APIs can be extracted with lightweight static analysis implemented using *srcML* [8]. For closed source apps, typically only the `.apk` package of an app is available. To this aim we use *Androguard*[1], which is effective in reverse engineering the app package to the needed source code and manifest files even in the presence of obfuscation [11]. In our approach, we rely on the `.apk` (binary) package file.

It is important to note that in Android the relationship between a specific permission and a particular API is not symmetric. That is, it is not necessarily the case that a use of a permission always requires the use of a specific set of APIs and vice versa. For example, permission X can be used with the API Y, Z, or both. Similarly, not every Android API needs a specific set of permissions. For example, the API X uses permission Y, Z, both Y and Z, or none of them. In an app, the explicit mapping between specific permissions and APIs is not available.

3.3 Forming Transactions or Training Set

To build a training set or transactions for association rule discovery, we use three different levels of mapping. In the first level of mapping, referred as *BaseMiner*, we consider the entire set of permissions, $P = \{p_1, p_2, ..., p_n\}$, related to the entire set of APIs $A = \{a_1, a_2, ..., a_m\}$ used for an app. In such a scheme, all the API class and permission entities in an app are assembled together into a single transaction. Forming transactions at this level does not require any traceability between API classes and permissions. We extract all the permissions from an app's manifest file and put them in the app's transaction. Then, we use *srcML* to parse the app source code. This allows us to identify Android API usages. Also, we have a preprocessed list of all the API class names used in the Android software stack and Java standard libraries. Therefore, we keep only

[1] `https://code.google.com/p/androguard/`

the names that match to those in the preprocessed Android name list in the app's transaction. For example, we include the API element names *WifiManager*, *Service*, and *Runnable*, and exclude the name *FTPServerSevice* because it is a user-defined, extended class. For example, in the *Open WiFi Cleaner* app, there are 47 API classes and 5 permissions. Therefore, a single transaction is formed for this app. It consists of a total of 52 permissions and API classes.

In the second level of mapping, referred as *FilteredMiner*, we refine the sets A and P. One such refinement is to only include permissions and APIs that are traceable to each other, and discard those that cannot be. This refinement is motivated by the fact that apps could use permissions that they do not necessarily need (in the worst case, exposing a security risk). On the other hand, there could be several API classes that do not need any permissions to be used. Knowing the permission from/to API class traceability enables us to discard unnecessary API classes and permissions from our mining. The resulting permission and API sets P' and A' will be subsets of P and A respectively. The mapping is performed by two mapping functions that are $surjective$ ($onto$) from the set P' to the set A' and from the set A' to the set P'. Each permission in P' maps to at least one API in A', and vice versa. Finally, we know that the permissions in the set P' are traceable to the API classes in set A' collectively as a group. In the *Open WiFi Cleaner*, 4 permissions are traceable to 10 APIs. Again, a single transaction for this app is formed with the *FilteredMiner* mapping. It consists of a total of 14 entities (permissions and API classes).

Finally, in the third level of mapping, referred as *FilteredMiner*$'$, we consider the relationship between a specific subset of permissions P' to a specific subset of APIs A'. There is many-to-many relationship between permissions and APIs. One API may need multiple permissions and one permission may be used in multiple APIs. A mapping function is needed that partitions the sets P' and A' into equivalence groups, such that a subset P'_i in the set P' maps to a subset A'_i in the set A'. The subsets P'_i and A'_i form an equivalence group. In the *Open WiFi Cleaner*, we can precisely map each of the 4 permissions to at least one of the 10 traceable API classes. Therefore,

we form 4 transactions with the *FilteredMiner′* mapping for this app. In this case, each transaction consists of a single permission and its corresponding API classes.

One important problem to be solved is how to filter and partition the transactions for the *FilteredMiner* and *FilteredMiner′* transactions. Our filtering mechanism is inspired from other approaches proposing filtering of traceability links (recovered using information retrieval methods) by relying on structural information [26]. Fortunately, *Androguard*, i.e., traditional static analysis, can be employed to obtain the two filtered mappings, i.e., *FilteredMiner* and *FilteredMiner′* mappings. Actually, *Androguard* uses *PScout*'s API to permission mapping. These mappings are derived from the static analysis of Android OS/framework. These mappings are needed for each Android OS version/level, as Android is a frequently evolving system. *PScout* uses traditional static analysis, i.e., call graph, of the Android framework/OS to deliver the mapping between API classes and permissions. It should be noted that apps found in a marketplace and in use on a variety of devices may not necessarily be for only a specific Android OS or a specific range of them. It is not uncommon for apps to be written for different compiled, minimum, and targeted Android OS/API levels. Therefore, a permission mapping from a single OS level (version) may not be sufficient. *Androguard* embeds *PScout*'s methodology; however, it considers permission mappings of several versions of Android OS/framework[2] from *PScout*.

Androguard can be used to discover the traceability between API classes in Java source code and permissions in the manifest file of each app. This tool is written in Python. It is used mainly for analyzing android `.apk` and *.dex* files. It has the functionality to trace the usage of permission in the source code. At first step, it imports the *.dex* file from the *.apk* file of an app. It then parses the `.dex` file to find where the permissions are used. It extracts API classes and their member functions and methods in an app through analysis of the Android app's binary package. Finally, it outputs which of these APIs are using the mentioned permissions. It uses the static analysis of *PScout* for

[2]`https://github.com/androguard/androguard/tree/master/androguard/core/api_specific_resources/api_permission_mappings`

this purpose. It maintains a knowledge base of the API to permission mappings of various Android OS levels.

For each app in the training set of transactions T, *Androguard* traces each permission to the APIs using it. That is, each app in the training of *ApMiner*'s with the *FilteredMiner* or *FilteredMiner′* configuration uses static analysis before performing association rule mining. The *BaseMiner* configuration does not require any static analysis. In summary, the transactions can be formed from using the *BaseMiner*, *FilteredMiner*, or *FilteredMiner′* mapping. Using either *BaseMiner* or *FilteredMiner* mapping results in $|T|$ number of transactions. However, the number of entities could be fewer with *FilteredMiner* than those with *BaseMiner* mapping (i.e., not all permissions and APIs are traceable to each other). Using the *FilteredMiner′* mapping could result in greater than $|T|$ number of transactions because each transaction could be split into multiple smaller transactions due to the precise mapping related to equivalence groups. The *BaseMiner* mapping exposes the risk of learning coincidental frequent itemsets and association rules that may not be representative of the actual usage or dependency of specific permissions and API classes together or even improper usage of them. The *FilteredMiner* and *FilteredMiner′* mappings reduce this risk, albeit at the cost of a slightly higher preprocessing cost in terms of additional static analysis.

As a result of the above analysis, we obtain transactions that are different from those used in other applications of frequent itemset mining (e.g., to identify file co-changes [35, 39]). They do not contain homogeneous entities. Our transactions contain a set of permissions extracted from the XML manifest file and a set of APIs names extracted from the Java source code. Since our goal is to provide recommendations of permissions given a set of APIs, we need a mechanism to identify the type of entity, i.e., whether it is an API class or permission, because we need only $\mathit{APIClass} \implies \mathit{Permission}$ rules. Therefore, we color-code our entities in each app. Each API is tagged with the prefix A and each permission is tagged with the prefix P. This color-coding is only for the mining process. The prefixes are trimmed when recommendations are made.

One key point to note is that we cannot always limit our analysis to classes belonging to the

Android SDK. For example, *pyLoad* is an app, which is a remote download manager. It requires two permissions: $ACCESS\_NETWORK\_STATE$ and $INTERNET$. The permissions can be traced to the API classes $android.net$.
$ConnectivityManager$ and $java.net.Socket$. Only one class is from the Android package. The other is from the Java networking package.

3.4 Recommendations with Association Rules

Once the transactions are formed, frequent pattern mining is performed, which results in frequent itemsets E of APIs and permissions. From these itemsets, we form association rules. For any given set of APIs A for which a developer needs help with permission recommendations, we use the association rules to recommend the required permissions. We select association rules in which A occurs as the antecedent (lhs) and another set of permissions P forms the consequent (rhs). Simply put, an association rule gives the conditional probability of the rhs also occurring when the lhs occurs, measured by its *confidence* value. The confidence of a rule is the ratio of the support of $lhs \cup rhs$ to the support of lhs. An association rule is of the form $lhs \Rightarrow rhs$. Multiple rules are possible for the same lhs entity (and also the rhs entity). When multiple rules are found for a given entity, they are first ranked by their confidence values and then by their support values; both in descending order (higher the value, stronger the rule). We allow a user specified cut-off point to pick the top n rules. Thus, the final recommendation of permissions *APIClass* $\Longrightarrow$ *Permission* is the list of all consequents in the selected n rules, ranked by confidence and support values.

3.5 Recommendation Examples from *ApMiner*

We demonstrate the applicability of *ApMiner* on an example app, namely *ADB Manager*[3], from *F-Droid*. It has 6 permissions and uses 67 API classes. The analysis scenario is the following: given the set of its 67 API classes, how well could *ApMiner* predict the 6 permissions (ACCESS_WIFI_STATE, RECEIVE_BOOT_COMPLETED, ACCESS_NETWORK_STATE, INTERNET, WAKE_LOCK, ACCESS_SUPERUSER)?

[3] `https://f-droid.org/repository/browse/?fdfilter=adb+manager`

To this aim, We trained *ApMiner* from other *F-Droid* apps (i.e., excluding *ADB Manager*). The transactions formed after each mapping are mined to discover the association rules. After generating the association rules, we ranked them according to their confidence values. The association rules are typically limited to only those that meet a minimum threshold value for confidence. In our case, we considered all rules with a value of at least 40%. Based on this criteria, *ApMiner* with the *BaseMiner* configuration predicted 11 permissions (ACCESS_WIFI_STATE, RECEIVE_BOOT_COMPLETED, READ_PHONE_STATE, READ_CONTACTS, ACCESS_NETWORK_STATE, INTERNET, GET_ACCOUNTS, WAKE_LOCK, ACCESS_SUPERUSER, VIBRATE, WRITE_EXTERNAL_STORAGE). In this example app, *ApMiner* was able to predict all permissions that it listed. Thus, its recall is 100%. In predicting these 6 permissions correctly, however, it recommend 11 permissions, i.e., 5 of them were false positives. Thus, its precision is 54%.

With the *FilteredMiner* configuration, *ApMiner* recommended 6 permissions (ACCESS_WIFI_STATE, READ_CONTACT, INTERNET, WAKE_LOCK, VIBRATE, WRITE_EXTERNAL_STORAGE). Out of 6 permissions, 3 were correct. Thus, its precision and recall are 50% each.

With the *FilteredMiner'* configuration, *ApMiner* recommended 7 permissions (ACCESS_WIFI_STATE, READ_CONTACT, ACCESS_NETWORK_STATE, INTERNET, WAKE_LOCK, VIBRATE, WRITE_EXTERNAL_STORAGE). Out of 6 permissions, 4 were correct. Thus, its precision and recall are 57% and 67% respectively.

We also used the direct API to permission mappings provided by *PScout* and *Androguard* to predict permissions of this example app. *PScout* predicted 9 permissions (ACCESS_WIFI_STATE, READ_CONTACT, INTERNET, READ_LOGS, VIBRATE, WRITE_EXTERNAL_STORAGE, READ_EXTERNAL_STORAGE, ACCESS_FINE_LOCATION, READ_PHONE_STATE). Out of 9 permissions, 2 were correct. Thus, its precision and recall are 22% and 33% respectively. *Androguard* recommended total 11 permissions (ACCESS_WIFI_STATE, READ_CONTACT, INTERNET, WAKE_LOCK, VIBRATE, WRITE_EXTERNAL_STORAGE, READ_EXTERNAL_

STORAGE, ACCESS_FINE_LOCATION, READ_PHONE_STATE, READ_CONTACT, GET _ACCOUNTS). Out of these 11 permissions, 3 were correct. Thus, its precision and recall are 27% and 50% respectively. On summary, for the *ADB Manager* app, *ApMiner* outperformed *Androguard* and *PScout* in terms of precision and recall.

CHAPTER 4

Empirical Study

The *goal* of this study is to evaluate the permission recommendations provided by *ApMiner*. The *purpose* is to assess what extent *ApMiner* is able to recommend correct and complete API-permission relationships in apps according to what is found in previously built, tested, and deployed mobile apps. The quality focus is the performance of the *ApMiner* prediction in terms of precision, recall, and F-score when considering the three different granularity levels of association rule minings, i.e., from the *BaseMiner*, *FilteredMiner*, and *FilteredMiner'* mappings of permissions and API classes. Furthermore, we compare *ApMiner* with two state-of-the-art approaches *Androguard* and *PScout*, which do not perform any mining of apps. They analyze Android OS and trace from API to permission calls with static analysis. The *perspective* is of an app developer who needs assistance with APIs and permissions to use. The *units of analysis* of our study are 600 open source Android apps from *F-Droid*.

4.1 Study research questions and design

The study aims at addressing the following research questions for the task of recommending permissions given the API classes used in an Android app:

RQ1: *To what extent is ApMiner able to precisely and completely provide API-to-permission (i.e., APIClass $\Longrightarrow$ Permission) recommendations?* This research question is related to the explanatory purpose of *ApMiner* ---i.e., positivist prospective----of our study, as to what is the correlation of the *ApMiner* recommendations with the APIs and permissions in the real apps. We answer this question considering the three different mapping levels *ApMiner* uses (*BaseMiner*, *FilteredMiner*, and *FilteredMiner'*), and compare their performances.

RQ2: *How does the mining-based approach of ApMiner compare with the traditional static analysis approaches of Androguard and PScout in APIClass $\Longrightarrow$ Permission recommendations?* This research question is related to the comparative purpose of our study.

4.2 Objects of the study: Android Apps and Test Benchmark

The *F-Droid* store features free and open source Android apps. The apps can be browsed and installed from the F-Droid website or directly from the *F-Droid* client app. The *F-Droid* Website also offers the source code of all applications for download. *F-Droid* does not require users to register and flags applications that contain ``anti-features'' such as advertising, user tracking or dependence on non-free software. For our study, we randomly selected 600 apps from *F-Droid* . Each app on the *F-Droid* Website has a link to the versioning system storing the software development history. Most of them use Git, SVN or Mercurial. For the purpose of our empirical evaluation, we need to know the actual permissions required by each app, along with the app's source code to identify the used APIs. We identify the actual permission usages from the `AndroidManifest.xml` file in each app. We considered these permissions used in each app to be the ground truth (test benchmark) in our evaluation.

4.3 Association Rule Generation in *ApMiner*

In our evaluation, we trained our *ApMiner* approach at three levels of transactions corresponding to the *BaseMiner*, *FilteredMiner*, and *FilteredMiner'* mappings of permissions and API classes. One type of association rules, $APIClass \implies Permission$, were generated at each transaction level. For itemset mining, we used the *arules*[1] package of R[2]. When applying association rule discovery, it is necessary to specify the minimum support for frequent item sets and minimum confidence for association rules. We considered 1% as threshold for minimum support. As for the confidence, we calibrated it considering values between 40% and 80% (in increments of 10%), as described in Section 4.6. We need all of the API elements used for the *BaseMiner* mapping. Therefore, we processed the source code and did static analysis with *srcML* (see Section 3).

[1] `https://cran.r-project.org/web/packages/arules/index.html`
[2] `https://www.r-project.org/`

4.4 Baselines for Comparison: *Androguard* and *PScout*

We consider both *PScout* and *Androguard* approaches in our comparison. It would allow us to observe the effects of single vs. multiple Android OS/framework mappings. Specifically, we used the ``All API calls mappings" for Android 4.1.1[3] for *PScout*, which was the latest version available. For each test app, we use the respective mappings of *PScout* and *Androguard* to form their *APIClass* $\Longrightarrow$ *Permission* recommendations. That is, we take each of the API classes and then query their mappings to find the corresponding permission(s). The final *APIClass* $\Longrightarrow$ *Permission* recommendations is the union of all such query results.

4.4.1 Metrics

We evaluated the effectiveness by calculating three widely used metrics *Precision*, *Recall*, and *F-Score*. Let us consider an app T with the set of permissions P^t and the corresponding set of APIs A^t. For the *APIClass* $\Longrightarrow$ *Permission* recommendations, assume that *ApMiner* predicted the set of permissions P^r. Precision is the measure of the actual (correct) over the total (correct+incorrect) number of predictions. A predicted set, P^r, could have false positives. Recall is the measure of the correct predictions over the total number in the actual set, P^t. The predictions could miss actual permissions or API classes in P^r , i.e., they are false negatives. The number of correct predictions (true positives) is t_p, the number of incorrect predictions (false positives) is f_p, and the number of missed actual ones (false negatives) is f_n. Precision and recall are defined as:

$$Precision = \frac{t_p}{t_p + f_p} \cdot 100\% \qquad Recall = \frac{t_p}{t_p + f_n} \cdot 100\%$$

while the F-Score is their harmonic mean:

$$F - Score = \frac{2 \cdot (Precision \cdot Recall)}{Precision + Recall} \times 100\%$$

[3]urlhttp://pscout.csl.toronto.edu/

4.5 Evaluation Procedure

We performed a posthoc evaluation of the recommendations provided by our approach. For each test app, we already know the permissions and corresponding API classes from our established benchmark. We simulate a use case in which a developer building this test app could have invoked *ApMiner* after writing a single API class or all of them. That is we have simulate the scenario: given API classes, the developer needs help with finding the appropriate permissions. *APIClass* $\Longrightarrow$ *Permission* rules are envisioned to help with this scenario.

To evaluate the accuracy of *APIClass* $\Longrightarrow$ *Permission*, we take each API class from the test app, i.e., A^t, and ask *ApMiner* to recommend the corresponding permissions from the association rules discovered from the training set. We considered the n *APIClass* $\Longrightarrow$ *Permission* rules for each API class ordered by their confidence and support. We take the union of the consequents, i.e., recommended API classes, of these rules, after all the permissions in A^t are attempted. This union gives the final recommendation set P^r. Precision and recall are calculated for the *APIClass* $\Longrightarrow$ *Permission* recommendations from the sets P^r for each test app. We followed the 10 fold cross validation for computing the precision and recall results. We had a total of 600 apps. We divided them into 10 folds of 60 apps each. We performed 10 runs. In each run, 9 folds, i.e., 540 apps, were used in training the association rules of *ApMiner* and the remaining one was used as a test set. The 10 runs ensured that each fold was tested. Precision, recall, and F-score values of 60 test apps, i.e., the test fold, in each run were recorded. *Androguard* and *PScout* were also subjected to the same test fold of each of the 10 runs, providing recommendations based on their methodology.

To address **RQ1** and **RQ2**, we compare the different mapping levels of *ApMiner*, and then compare *ApMiner* with *Androguard* and *PScout*. We test the following null hypotheses:

H_{01}: There is no statistically significant difference between the precision, recall, and F-score values from *BaseMiner*, *FilteredMiner*, and *FilteredMiner'* mappings in *ApMiner*.

H_{02}: There is no statistically significant difference between the precision, recall, and F-score values from *ApMiner*, *Androguard*, and *PScout*.

Table 4.1: Average Precision and recall of *ApMiner* with different minimum confidence values.

	Precision			**Recall**			**F-Score**		
Min. Confidence	*BaseMiner*	*FilteredMiner*	*FilteredMiner′*	*BaseMiner*	*FilteredMiner*	*FilteredMiner′*	*BaseMiner*	*FilteredMiner*	*FilteredMiner′*
40%	29.83%	41.10%	60.37%	82.96%	49.78%	52.64%	43.84%	45.03%	56.24%
50%	36.66%	44.87%	60.01%	75.43%	47.76%	47.27%	49.34%	46.27%	52.88%
60%	56.86%	49.03%	62.53%	67.68%	46.65%	42.73%	61.38%	47.81%	50.76%
70%	66.17%	59.98%	62.32%	57.25%	38.10%	40.68%	61.38%	46.59%	49.22%
80%	75.58%	74.92%	64.92%	44.25%	33.21%	34.93%	55.82%	46.02%	45.42%

Specifically, we pairwise compare the techniques using Willcoxon Rank Sum Unpaired tests. Because we perform multiple comparisons, we adjust the obtained p-values using the Holm's correction procedure [20]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on. Also, to analyze the magnitude of the observed differences, we rely on the Cliff's delta (d) [18] effect size measure. We followed the guidelines in [18] to interpret the effect size values: negligible for $|d| < 0.14$, small for $0.14 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

4.6 Results

Before assessing how *ApMiner* performs in comparison with alternative approaches, we assess how its performance varies for different values of the confidence threshold in the association rule discovery. Results for a confidence threshold varying between 40% and 80% with stepwise increments of 10% are reported in Table 4.1. Not surprisingly, increasing the confidence means higher precision (Figure 4.1) at the cost of lower recall(Figure 4.2). Their aggregated value (F-Score) reaches the highest value for the confidence of 60% for the *BaseMiner* and *FilteredMiner* levels of filtering, while the *FilteredMiner′* level of filtering reaches the highest F-Score for the confidence of 40%. For the purpose of comparison with the other approaches, we only report results for the confidence of 40% because it allows us to achieve the best performances for the finest level of filtering (*FilteredMiner′*). That is, it provides the best F-score value; thus, a good middle ground between precision and recall.

Table 4.2 reports the precision, recall and F-Score values of *ApMiner* from the 10-fold cross

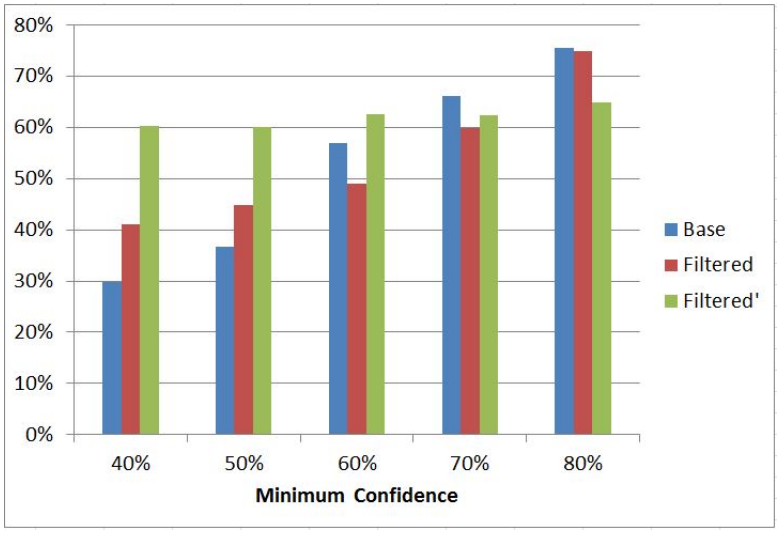

Figure 4.1: Average Precision with different minimum confidence values.

validation. We report the minimum, maximum, and average values across the 10 folds *ApMiner*. Moreover, this table reports results for the alternative approaches, i.e., *PScout* and *Androguard*. Because they are unsupervised approaches (in the sense that they do not need any training of rules), for each fold we consider results for the apps contained in the test set of *ApMiner*. The precision and recall numbers for all the mappings were calculated for the same test set of apps for each run.

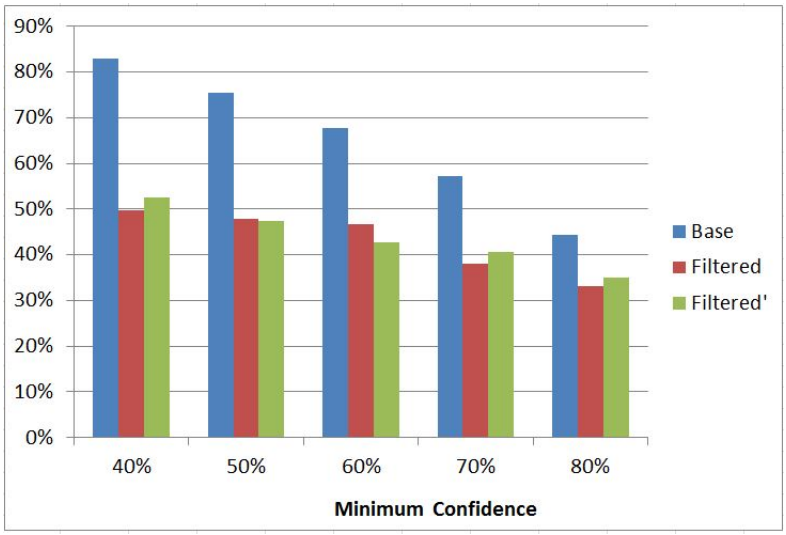

Figure 4.2: Average Recall with different minimum confidence values.

As the mapping becomes more filtered, the precision and recall values increase and de-

crease respectively. The *FilteredMiner′* and *BaseMiner* mappings reported the best precision and recall values respectively. For the *APIClass* $\Longrightarrow$ *Permission* mapping, the average of the precision values from 10 folds for the *FilteredMiner′* mapping is 63%. The average of the recall values from 10 folds for the *BaseMiner* is around 83%. On average, the precision gains (up from about 30% to 60%) over the increased level of filtering are more pronounced than the loss in recall (down from about 80% to 50%). In summary, going from ``more but generic" (*BaseMiner*) to ``less but precise" (*FilteredMiner′*) did produce the commonly seen tradeoff between precision and recall values. These results suggest that discovering rules from *BaseMiner*'s coarse level captured a few coincidental (e.g., due to generic controller/view classes such as *Activity* and *Views* that typically do not require any permissions) than actual dependencies between permissions and API classes. That is, noisy rules bubbled up over the valuable, signal rules. The ratio of signal to noise gradually improves as we move from the *BaseMiner* to the *FilteredMiner′* mapping. On average, *BaseMiner*, *FilteredMiner*, and *FilteredMiner′* recommended 10, 5, and 4 permissions for the tested apps respectively. On average, the correct number of permissions in the test apps was 5.

Table 4.3 reports results of the statistical comparison involving *ApMiner* with the different levels of mapping, as well as of *PScout* and *Androguard*. Specifically, the table reports the p-values (adjusted using the Holm's correction procedure, and and statistically significant values shown in boldface) resulting from a pairwise comparison among techniques using the Wilcoxon Rank Sum test, as well as the Cliff's delta effect size values (effect sizes are positive when Technique 1 is better than Technique 2; negative otherwise).

The first part of the table, which is the comparison among the *ApMiner*'s different levels

Table 4.2: Precision and recall of *ApMiner* measured on 600 apps from the *F-Droid* platform.

	Precision					**Recall**					**F-Score**				
	ApMiner			*PScout*	*Androguard*	*ApMiner*			*PScout*	*Androguard*	*ApMiner*			*PScout*	*Androguard*
	BaseMiner	*FilteredMiner*	*FilteredMiner′*			*BaseMiner*	*FilteredMiner*	*FilteredMiner′*			*BaseMiner*	*FilteredMiner*	*FilteredMiner′*		
avg	29.83%	41.10%	60.37%	27.30%	26.14%	82.96%	49.78%	52.64%	44.70%	51.23%	42.15%	44.90%	55.21%	32.42%	33.65%
max	35.76%	45.95%	63.09%	35.77%	34.23%	89.77%	66.65%	66.98%	51.98%	60.56%	49.09%	48.88%	63.08%	36.46%	38.80%
min	22.35%	34.95%	52.90%	18.77%	15.33%	78.29%	45.76%	46.57%	33.45%	34.12%	35.79%	41.65%	52.37%	32.42%	24.47%

of mappings, confirms the results discussed above. Thus, the finer levels of mapping have significantly better precision than coarser ones, with a medium to large effect size. The opposite is true in terms of recall, which is the best for *BaseMiner*. A noteworthy point is that the difference in terms of recall between *FilteredMiner* and *FilteredMiner′* is negligible, and is in favor of *FilteredMiner′*, making the latter achieving the best compromise, as it can also be noticed from the F-Score results. Overall, the null hypothesis H_{01} can be rejected and we can summarize our results as follows:

RQ1 summary: Increasing the level of filtering achieves a statistically significant increase of precision (from 30% to 60%) while exhibiting also a significant decrease of recall (from about 80% to 50%). The *FilteredMiner′* level of mapping achieves the best middle ground, with average precision of 60% and average recall of 52%.

Table 4.2 also reports the precision, recall, and F-Score values for *Androguard* and *PScout*. Results of the statistical comparison are reported in the second and third parts of Table 4.3 for *Androguard* and *PScout* respectively. *Androguard* and *PScout* achieve the average precision of 26.14% and 27.30% respectively. *Androguard* achieves slightly better recall (51.23%) than *PScout* (44.70%). Not surprisingly, *Androguard* and *PScout* perform similarly in terms of precision. *Androguard* provides significantly better recall (but negligible effect size). These observations in-

Table 4.3: Results of the statistical comparison among techniques and *ApMiner* mapping levels.

Technique 1	**Technique 2**	**Precision**			**Recall**			**F-Score**		
		p-value	d	magnitude	p-value	d	magnitude	p-value	d	magnitude
BaseMiner	*FilteredMiner*	<**0.001**	-0.41	medium	<**0.001**	0.69	large	**0.01**	-0.10	negligible
FilteredMiner	*FilteredMiner′*	<**0.001**	-0.46	medium	0.27	-0.05	negligible	<**0.001**	-0.40	medium
BaseMiner	*FilteredMiner′*	<**0.001**	-0.70	large	<**0.001**	0.62	large	<**0.001**	-0.41	medium
Androguard	*BaseMiner*	<**0.001**	-0.17	small	<**0.001**	-0.54	large	<**0.001**	-0.26	small
Androguard	*FilteredMiner*	<**0.001**	-0.41	medium	0.24	0.06	negligible	<**0.001**	-0.35	medium
Androguard	*FilteredMiner′*	<**0.001**	-0.60	large	0.59	**0.02**	negligible	<**0.001**	-0.50	large
PScout	*BaseMiner*	**0.01**	-0.10	negligible	<**0.001**	-0.64	large	<**0.001**	-0.23	small
PScout	*FilteredMiner*	<**0.001**	-0.41	medium	**0.01**	-0.11	negligible	<**0.001**	-0.30	small
PScout	*FilteredMiner′*	<**0.001**	-0.68	large	<**0.001**	-0.14	negligible	<**0.001**	-0.54	large
Androguard	*PScout*	0.48	-0.02	negligible	<**0.001**	0.13	negligible	0.69	-0.01	negligible

dicate that there is potential value in including permission mappings from multiple Android OS versions in *APIClass* $\Longrightarrow$ *Permission* recommendations. On average, *Androguard* and *PScout* recommended 8 and 7 permissions for the tested apps respectively.

Concerning the comparison between *ApMiner* and the two alternative techniques, the *ApMiner*'s *BaseMiner* mapping has significantly better precision and recall than *Androguard*. The effect sizes are small for precision and large for recall. When finer levels of mapping are adopted, the difference in terms of precision increases (the effect sizes are medium for the *FilteredMiner* mapping and large for the *FilteredMiner'* mapping), whereas it becomes negligible in terms of recall. Due to the similar performances between *Androguard* and *PScout*, results for the latter are in line with those achieved for *Androguard*.

Based on the obtained result, we can reject the null hypothesis H_{02} and summarize our results as follows:

RQ2 Summary: The *BaseMiner* level of *ApMiner* achieves significantly better recall (83%) than *PScout* (45%) and *Androguard* (51%) with a large effect size, while the precision difference is small/negligible. When increasing the granularity of the mapping, the difference in terms of precision becomes large in favor of *ApMiner* (up to 60%, vs. 26%, 27%), while the recall difference goes down to negligible.

4.7 Qualitative Analysis

Encouraged by the quantitive results of *ApMiner*, we wanted to examine if the rules we mined are already found in (well done) Android documentation[4]. The *android.Manifest.permission class* contains the description of all 151 permissions available in API level 21. Each permission is described with its usage and in which condition an app developer should mention this permission in $AndroidManifest.xml$. For example, it states that the $READ\_PHONE\_STATE$ permission will allow read only access of the phone state. However, these permissions are not always linked to the specific API classes in this documentation. We found only 13 permis-

[4] `http://developer.android.com/index.html`

sions that mention the specific API classes that require them. For example, in the description of the $BIND\_NFC\_SERVICE$ permission, it is clearly mentioned that the service class *HostApduService* requires it. Our approach recommended several *Permission* $\Longrightarrow$ *APIClass* rules with confidence value of over 95% that were correct, i.e., had high precision and recall. One such rule is $READ\_CONTACT$ $\Longrightarrow$ *ContentResolver*. Apps using this class typically act as a *content provider* for the contacts stored in the device. We found at least 44 permissions that were in the accurate *Permission* $\Longrightarrow$ *APIClass* rules with incomplete or no documentation for the corresponding API classes.

An exhaustive examination of all the Android API classes would be highly tedious and human error-prone because of its large number of classes. We randomly sampled 30 *APIClass* $\Longrightarrow$ *Permission* rules that had confidence over 80% (and were deemed accurate in the quantitative analysis). We consulted the documentation of these API classes to see if they mentioned permissions. We found 6 cases where the required permissions were mentioned in the documentation. For example, the documentation of the *TelephonyManager* class clearly mentioned that it requires the $READ\_PHONE\_STATE$ permission. For the other 24 cases, we could not find any link between API classes and their needed permissions. For example, the *APIClass* $\Longrightarrow$ *Permission* rule *LocationManager* $\Longrightarrow$ *INTERNET* indicates that the *INTERNET* permission is frequently needed for using Google maps or other web based location services. It was the case in many of the apps that used the *LocationManager* in our evaluation. This relationship is not documented. The documentation for the *LocationManager* class lists only $ACCESS\_COARSE\_LOCATION$ and/or $ACCESS\_FINE\_LOCATION$ permissions. In fact, there was a *StackOverflow* question ``*Location manager is not working without internet*''[5], which indicates that it is perhaps important for the expected behavior. Another example of no documentation is the rule *MediaPlayer* $\Longrightarrow$ *VIBRATE*. Apps related with media, such as audio or video, often use the vibration feature; thus, they need the namesake permission.

[5] `http://stackoverflow.com/questions/20112140/location-manager-is-not-working\ \-without-internet`

Our manual examination suggests that the mined rules with our *ApMiner* approach are not only accurate and meaningful, but also a potentially valuable help to new app developers. We could not find them in the Android documentation. Thus, our rules could be used to augment or supplement Android documentation. Although the accuracy of our *ApMiner* can be considered high enough, we wanted to investigate the violated rules (i.e., those produced false positives and negatives). We examined *APIClass* $\Longrightarrow$ *Permission* rules with at least 99% confidence (i.e., very high likelihood of occurrence, which are expected to be obeyed).

Regarding the violated *APIClass* $\Longrightarrow$ *Permission* rules, $PackageMangaer \Longrightarrow CHANGE\_COMPONENT\_ENABLED\_STATE$ and $PowerManger \Longrightarrow WAKE\_LOCK$ are violated in apps *OpenWatchRecorder* and *FonBot*. In these apps, the $SET\_PREFERRED\_APPLICATION$ and $REBOOT$ permissions are used respectively, which are contrary to the recommended permissions by *ApMiner*. The $CHANGE\_COMPONENT\_ENABLED\_STATE$ permission allows an application to change whether an application component (other than its own) is enabled or not and the $SET\_PREFERRED\_APPLICATION$ permission is used for setting a priority for application packages. The $REBOOT$ permission is required to reboot the device and the $WAKE\_LOCK$ permission allows to keep the processor from sleeping or the screen from dimming. Once again, this examination calls for the need to analyze *related permissions*. We need to include *Permission* $\Longrightarrow$ *Permission* rules. The super, sub, and equivalence relationships between permissions need to be considered.

A manual examination of the violated rules in apps suggest that we need to consider $APIClass \Longrightarrow APIclass$ and *Permission* $\Longrightarrow$ *Permission* dependencies to further improve our *ApMiner* accuracy.

We also manually searched the rules to observe the difference among *ApMiner*, *Androguard* and *PScout*. We have found several rules from *ApMiner* that could not be found in the api-permission mappings of *Androguard* and *PScout*. Let us consider the app *AndroSS* to facilitate the discussion. It is for capturing screenshots and save them in the storage both internal and

external. This app requires two permissions: *VIBRATE* and *WRITE_EXTERNAL_STORAGE*. We found 56 API classes (from Android and Java) were used in its implementation. Among these API classes, we searched the list of rules recommended by *ApMiner* to find any corresponding rules. We found two rules: *android.os.Vibrator* $\Longrightarrow$ *VIBRATE* and *java.io.FileOutputStream* $\Longrightarrow$ *WRITE_EXTERNAL_STORAGE*. Therefore, *ApMiner* recommended both permissions used in *AndroSS* app. After that we searched the api-permission mappings of *Androguard* and *PScout*. From *Androguard*, we found the reference of *android.os.Vibrator* $\Longrightarrow$ *VIBRATE*. We could not find the other rule. Therefore, *Androguard* could only recommend one permission. From *PScout*, we found a reference related to api *android.os.Vibrator*, which mapped to the permission *WAKE_LOCK* and not *VIBRATE*. For *java.io.FileOutputStream*, we could not find any mapping in *PScout*. Therefore, *PScout* was not able to recommend any permissions for this app.

This example suggests that analysis beyond Android APIs needs to be considered for accurate permission recommendations, which is exactly what *ApMiner* does.

CHAPTER 5

Threats to Validity

In this section we discuss construct, internal and external threats to validity of the results of our empirical study.

Threats to *construct validity* concern the relationship between theory and observation, and mainly relate, in our study, to possible errors in our measurements. They include:

- *Mapping Errors*. We used *Androguard* for preprocessing the transactions with *FilteredMiner*' mapping. Although, it is a vetted and popular tool, it could create a residual error in our approach, which could have its own ripple effect. Overall, our *ApMiner* complements *Androguard* in providing support for both forward and reverse engineering of apps, i.e., history based learning and static analysis.

- *Evolutionary Changes*. With the release of new API levels in Android, the system-defined permissions and API classes may evolve. Our approach may need to trace these name changes via a technique such as origin analysis, and evolve our rules accordingly.

- *Granularity*: Our result are reported at the API class level. Developers may need a finer granularity information than classes for tracing permissions. For example, if a class contains a method to write on the external storage and one to read from it, and one uses only the read method, then the write permission is not needed. It is possible that our results would be different at granularity levels such as methods.

Threats to *internal validity* concern confounding factors internal to our study that might have influenced our results. They might be due to the fact that *ApMiner* captures correlation (in terms of frequent co-occurrence) between permissions and APIs. Such a frequent co-occurrence may not correspond to a cause-effect relationship between API usage and required permission,

or vice-versa. We mitigated such a threat by complementing the quantitative analysis with some qualitative investigation, done by manually analyzing a few exemplar cases.

Threats to *external validity* concern the generalisation of our findings. They may be related to:

- *Permission Coverage*. Out of the 151 permissions, 47 were used in our *F-Droid* evaluation. Therefore, the results may not be reflective of all permissions.

- *Customized permissions not considered*. An Android app is allowed to define its own customized permission (beyond the available system-defined ones). Such permissions may not be used frequently. Therefore, our approach may fail to capture rules involving them.

- *Platform and store generalization*. We used only open source apps from the *F-Droid* platform. Closed source apps, such as those distributed on *Google Play Store*, would require reverse engineering, which can be performed by existing tools such as *Androguard*. Nonetheless, we do not claim that our approach and results would hold equally well on other platforms, such as *iOS* and *Windows Phone OS*. These other platforms are more tightly controlled in terms of their app stores than Android.

CHAPTER 6

Related Work

This section discusses related work for what concerns the (mis)use of permission schemes in Android, malicious (and hidden) behavior of mobile apps, and approaches for security and privacy analysis of mobile applications.

Several authors investigated the usage of permission schemes in Android apps. In particular, Stevens *et al.* [31] conducted a study on permissions misuses in 10,000 Android apps and on questions developers ask about permissions usage on StackOverflow. They found that popular permissions are more subject to permission misuses than less popular ones, and that users post several questions on StackOverflow about such very popular permissions. In other words, while popular permissions are not necessarily better understood than others, they receive attention in the sense that people search for solutions on StackOverflow. Felt *et al.* [13], whose user study highlighted the low attention paid by end-users to permission scheme. Barrera *et al.* [3] conducted an empirical analysis on permission-based security models. Shin *et al.* [29] proposed a formal model to analyze the permission mechanism in Android framework. One extension of permission based research work is identifying security flaws [30]. Holavanalli *et al.* [19] proposed an extension in the Android permission framework by adding permissions flow. Felt *et al.*[14] discussed possible ways to address permission re-delegation and presented a new mechanism for defending against its threats. Our work does not explicitly focus on mechanisms to defend against permission misuses. Nevertheless, our approach based on association rule discovery can point out cases of infrequent, and therefore possibly suspicious, permission-API relationships. Also, work such as the one of Stevens *et al.* [31] and of Felt *et al.* [13] motivate our research. Despite the fact that API required permissions are pretty-well documented, their level of understanding still cause misuses. Hence, approaches such as our proposed *ApMiner* could provide fast recommendations to the developers even before getting answers on StackOverflow.

A different research thread relates finding stealthy and malicious behavior of Android apps. Schmidt *et al.* [28], which examined the evolution of malware in smart-phone apps. Blasing *et al.* [5] presented a system for suspicious software detection. Zhou *et al.* studied on detecting malicious apps in official and alternative markets [38], and also developed an approach to identify repackaged smartphone applications in third-party Android marketplaces [36]. Other approaches concern automated security testing, privacy monitoring, and Mahmood *et al.* [23] proposed an approach for automated security testing of Android applications. Enck *et al.* [12] proposed a system for real-time privacy monitoring on smartphones. *SCanDroid* developed by Fuchs *et al.* [15] was used for security certification of Android applications. Zhou *et al.* [37] conducted a study on characterization and evolution of Android malware. Several other tools and techniques are developed for detecting privacy leaks [22] [4]. Grace *et al.* [17] studied unsafe exposure in app advertisement. Chen *et al.* [7] used clone detection techniques to identify potential malicious app in Android market.

Several approaches tools were built by researcher to analyze the behavior of Android apps. One such example is *AsDorid* [21], which is used for identifying stealthy behavior by relying on user interface and program behavior contradiction. Nauman *et al.* [25] proposed a tool for extending the Android permission model. Chan *et al.* [6] developed a tool named *Droidchecker* for analyzing Android applications for capability leaks. Wu *et al.* [33] presented *Droidmat*, a tool for Android malware detection through manifest and API calls tracing. Xu *et al.* [34] developed a solution that bypasses the need to modify the Android OS while providing much of the security and privacy that users desire.

Gorla *et al.* [16] propose *CHABADA*, a tool that analyzes Android apps' behavior (in terms of API usages and related permissions) against the topics contained descriptions. The approach allows to highlight cases of undeclared behavior and in the study Gorla *et al.* conducted was also able to identify 56% of known app malware, without the need for performing a deep analysis of malware patterns. Another possibility to discover hidden behavior is by performing testing activi-

ties, for examples using tools such as the *EvoDroid* developed by Mahmood *et al.* [24]. We believe our work is complementary to the approaches mentioned above. First, our approach can either be used to identify suspicious behavior, but also to help developers in selecting appropriate permission. Second, while approaches *CHABADA* [16] highlight inconsistencies between description and APIs, our approach highlights potential inconsistencies between APIs and permissions.

To the best of our knowledge, the most related tools to *ApMiner* are *Stowaway*, *PScout*, and *Androguard*. As explained in the introduction (see Section 1), *Stowaway* [27] identifies API to permission mappings by performing a static analysis of the app and then by relying on an Android permission map (currently outdated[1]). The authors found that about one third of the 940 apps they analyzed using *Stowaway* are overprivileged. To date, a more accurate tool is *PScout*[2], which uses static analysis to analyze the permission mechanism of Android. The authors of *PScout* have applied the tool on four versions of Android, showing the tool scalability on over 3MLOC code bases, and also highlighting how about 22% of the non-system permissions are unnecessary. Last, but not least *Androguard*[10] is used for mainly reverse engineering of Android applications. For the purpose of permission checking, *Androguard* exploits the API to permission mappings identified by *PScout*. Since *PScout* is considered the state-of-the-art approach for Android permission analysis, we considered it as a baseline, along with *Androguard*, for the reasons explained above. Besides achieving better performances, *ApMiner* is different from the aforementioned approaches, as it exploits other apps' permission usages---analyzed through association rule discovery---other than only relying on static analysis.

[1] http://www.android-permissions.org

CHAPTER 7

Conclusion and Future Work

Permissions constitute a crucial mechanism to ensure security and privacy of Android apps. Unfortunately, such permissions are not always well documented [27, 32] leading to misuses [31]. The onus is on the developers to use them correctly in the apps they build. To support developers in the choice of permissions and to identify cases of excessive permissions, we propose an approach, named *ApMiner*, to recommend permissions for given sets of Android APIs. *ApMiner* combines static analysis---that builds candidate traces between APIs and permissions at different levels of granularity---with association rule discovery [1], from mining permission usages in other apps, which are already published at reputable marketplaces. Results of our evaluation, conducted on 600 open source apps taken from the *F-Droid* store, indicate that by increasing levels of filtering, *ApMiner* exhibits precision ranging from 30% to 60%, and recall, in turn, decreasing from 80% to 50%. Such results ensure a level of precision significantly higher (up to twice) than those achieved by alternative state-of-the-art tools such as *Androguard* and *PScout*. Overall, our approach *ApMiner* may help prevent both malfunctioning (false negatives) and maliciousness (false positives) of apps with its choice of filters.

Our future work includes evaluating *ApMiner* on other marketplaces such as *Google Play* and cross-marketplace validation. Additionally, we plan on conducting user studies and surveys with Android app developers to assess its practicality and relevance. Additionally, we plan to integrate inter-app permission analysis in our approach, i.e., an app may not request a permission, but may use a feature/permission provided by another app through the *intent* mechanism.

REFERENCES

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 3--14. IEEE, 1995.

[2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217--228. ACM, 2012.

[3] David Barrera, H Güneş Kayacik, Paul C van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73--84. ACM, 2010.

[4] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49--54. ACM, 2011.

[5] Thomas Blasing, Leonid Batyuk, A-D Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55--62. IEEE, 2010.

[6] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Droidchecker: analyzing Android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125--136. ACM, 2012.

[7] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *ICSE*, pages 175--186, 2014.

[8] Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. An xml-based lightweight C++ fact extractor. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 134--143, 2003.

[9] Anthony Desnos. App manifest. `http://developer.android.com/guide/topics/manifest/manifest-intro.html` [cited 13 November 2015].

[10] Anthony Desnos. Androguard: Reverse engineering, malware and goodware analysis of Android applications, 2013.

[11] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, 2011.

[12] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99--106, 2014.

REFERENCES (continued)

[13] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1--3:14, New York, NY, USA, 2012. ACM.

[14] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[15] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. SCanDroid: automated security certification of Android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[16] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1025--1035, 2014.

[17] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101--112. ACM, 2012.

[18] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[19] Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen, Steven Y Ko, and Lukasz Ziarek. Flow permissions for Android. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 652--657. IEEE, 2013.

[20] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 6:65--70, 1979.

[21] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *ICSE*, pages 1036--1046, 2014.

[22] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. ScanDal: Static analyzer for detecting privacy leaks in Android applications. *MoST*, 2012.

[23] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 22--28. IEEE, 2012.

[24] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599--609. ACM, 2014.

REFERENCES (continued)

[25] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328--332. ACM, 2010.

[26] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. When and how using structural information to improve IR-based traceability recovery. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 199--208, 2013.

[27] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627--638, 2011.

[28] A-D Schmidt, H-G Schmidt, Leonid Batyuk, Jan Hendrik Clausen, Seyit Ahmet Camtepe, Sahin Albayrak, and Can Yildizli. Smartphone malware evolution revisited: Android next target? In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 1--7. IEEE, 2009.

[29] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A formal model to analyze the permission authorization and enforcement in the Android framework. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on*, pages 944--951. IEEE, 2010.

[30] Wook Shin, Sanghoon Kwak, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A small but non-negligible flaw in the Android permission scheme. In *Policies for Distributed Systems and Networks (POLICY), 2010 IEEE International Symposium on*, pages 107--110. IEEE, 2010.

[31] Ryan Stevens, Jonathan Ganz, Vladimir Filkov, Premkumar T. Devanbu, and Hao Chen. Asking for (and about) permissions used by Android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 31--40, 2013.

[32] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the android permission specification. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 217--228, 2012.

[33] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and API calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62--69. IEEE, 2012.

[34] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security Symposium*, pages 539--552, 2012.

REFERENCES (continued)

[35] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574--586, 2004.

[36] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317--326. ACM, 2012.

[37] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95--109. IEEE, 2012.

[38] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS*, 2012.

[39] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429--445, June 2005.