PhD Dissertation

International Doctoral School in

Information and Communication Technologies (ICT)
University of Trento, Italy

Security Testing of Permission Re-delegation Vulnerabilities in Android Applications

Biniam Fisseha Demissie

SUBMITTED TO THE DEPARTMENT OF
INFORMATION ENGINEERING AND COMPUTER SCIENCE (DISI)
IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Advisor: Dr. Mariano Ceccato, Fondazione Bruno Kessler, Trento, Italy

January 2019

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Mariano Ceccato for the continuous support during my PhD study, for his patience and motivation. His guidance helped me during the time of the research and the writing of this thesis.

I would also like to thank Fondazione Bruno Kessler (FBK), the organization that employed me during my PhD research and thesis writing.

I would like to thank all the members of my PhD defense committee, Dr. Silvio Ranise, Dr. Riccardo Scandariato and Dr. Alessandra Gorla, who are also the external reviewers, for their kind feedback and availability for my defense.

I also thank my colleagues in the Software Engineering group in FBK for the several feedbacks and suggestions I received regarding my work during our weekly seminars and informal discussions and for the off-work fun moments.

Last but not least, I would always be indebted to my family and friends for their best wishes and unconditional support.

Biniam Fisseha Demissie
Trento, Italy
April 2019

Abstract

Smartphones play an important role in our daily lives. Once used only for communication purposes are now also used for several day-to-day activities ranging from social media and entertainment to privacy sensitive operations such as data storage, fitness tracking, mobile banking and sending/receiving business e-mails. This is achieved thanks to the several smartphone applications (apps) that are available. One of the most popular smartphone operating systems is Android. As of now, there are more than 3 million apps for Android. The Android platform facilitates reuse of apps' functionalities by allowing an app to request a task from another app installed on the same device through inter-process communication mechanism. This possibility is probably one of the reasons for the popularity of Android where an app can reuse a feature available in other apps. However, this integration also poses security risks to the privacy of the end-users if it is not implemented properly. Permission re-delegation vulnerability is a kind of privilege escalation that happens when unprivileged apps exploit this integration feature to make privileged apps perform a privileged action on their behalf.

Static analysis techniques as well as run-time protections have been proposed to detect permission re-delegation vulnerabilities. However, as acknowledged by their authors, most of these approaches are affected by many false positives and, hence, fall short of precision because, they do not discriminate between intentional task requests and actual permission re-delegation vulnerabilities.

In this thesis, we propose automatic techniques to classify potential permission re-delegation vulnerabilities detected by static analysis in real world Android apps as intentional task requests or actual vulnerabilities and to automatically generate test cases that show how the vulnerabilities can be exploited. This could be helpful for developers to easily analyze their apps and fix vulnerabilities before releasing their apps.

The proposed approaches have been experimentally validated with thousands of real world apps and have been seen to perform better than state-of-the-art tools and techniques in terms of precision.

Keywords — Security testing, test case generation, security oracle, smartphone applications, Android apps, static analysis, dynamic analysis

Table of contents

List of figures

List of tables

CHAPTER 1

Introduction

Android is a popular operating system (with more than 85% market share [36]) that is used by devices ranging from smart phones and tablets to wearables, smart TVs and automobiles. The proliferation of Android devices has created a good business opportunity for novice developers to create and distribute applications (herein apps) easily using the centralized Android market, called Play Store, targeting millions of devices. The Android Play Store is now the biggest app store in the world, and as of March 2018, it has more than 3.6 million apps available for download [84]. The existence of wide range of apps plays an important role in the popularity of Android. Devices once used for leisure such as taking photos, getting social network updates and listening to music are now used for more sensitive activities, such as checking and replying to business e-mails while traveling, doing banking and shopping activities to mention but a few.

The Java-based development model attracts many app developers with different level of experience. Hence, it is common to find several apps in the market for similar purposes by different developers. For example, a simple search for *torch light* in Google Play Store returns more than 200 different but similar apps. Developers have fierce competition and when a new potential app idea arises, they need to rush and develop the app before similar apps become available on Play Store by competitors. Since the first apps that appear on the market are usually more accepted by the users, being the first to publish an app might help the developer gain market share.

Due to the high time-to-market pressure, developers usually spend more time on the app's user experience and appealing user interface and they might overlook the need for quality and security assessment of their apps [30]. Other quality aspects, possibly including security issues, might be delayed for future updates.

If apps have security defects, they might be target of attacks in order to gain unauthorized access and leak privacy sensitive user data and information.

1.1 Motivation and Problem Statement

The Android framework is design in such a way that developers can easily reuse features that are already made available either by the Android system or by other developers. For example, apps can use the default camera app to take photo because it makes this feature available to other apps.

When apps make their features and functionalities available to other apps, they are opening doors into their app. Though feature reuse is the backbone of Android, insecure implementation could make an app vulnerable to attacks and pose serious security risk to the end-user. One of the most common vulnerability in this regard is a kind of privilege escalation vulnerability called permission re-delegation [33].

However, it is hard to distinguish between legitimate integration through reuse and permission re-delegation vulnerabilities. Therefore, automatic detection of this vulnerability leads to enormous amount of false positives, because legitimate integration would also be detected as "potential" vulnerability.

Google provides *App Security Improvement Program* [38], a service for app developers to improve the security of their apps. Apart from tips and recommendations for building more secure apps, the program also provides app analysis in order to identify potential security enhancements in the apps before publishing them to Play Store.

Several approaches are also proposed by the research community ([51], [13], [93], [17], [58],[24], [55], [97],[59],[98],[26]) to either statically detect the existence of this vulnerability or provide a run-time protection.

These approaches, however, acknowledge that they suffer from many false positives and require a developer or a security analyst to manually analyze several reports, rendering these techniques expensive to apply in real scenarios. This is due to the fact that it is difficult for an automated static or dynamic analyzer to understand whether permission re-delegation occurred intentionally or by programming mistake. Thus, a more precise approach would improve analysis and detection of permission re-delegation vulnerabilities.

The key research challenge in this regard is to automatically detect permission re-delegation vulnerabilities and generate proof-of-concept test cases that document them.

1.2 Thesis Statement

In this thesis, we investigate techniques to *precisely* detect permission re-delegation vulnerabilities in Android apps. The aim is to produce tools that, given an app, analyze and report if it is vulnerable or not. When an app is vulnerable, the tools automatically generate test cases that show how the vulnerability can be exploited to attack the app. In such a way, we advance the state-of-the-art approaches (1) by classifying security reports into safe cases of apps integration or permission re-delegation vulnerability. reasoning about static or dynamic analysis reports as being benign, vulnerable, or malicious, and (2) providing execution scenarios that can be used as proof-of-concept attack.

Such tools could be helpful for app developers such that they can verify whether their apps contain permission re-delegation vulnerabilities before distributing. Market owners (e.g., Google Play) could also benefit from the tools to vet apps for security defects before publishing them (e.g., as part of *App Security Improvement Program*). In the context of BYOD (bring your own device), where employees use their own device to connect to a secure corporate network, a corporate security analyst might benefit from these tools to analyze if apps in the user's device have security defects that might compromise the confidentiality of corporate data stored in the end-user personal devices.

However, for such tools to be effective, they have to be precise and fast with few false positives. Moreover, they have to be scalable such that they can be used to analyze real world apps.

1.3 Research Contribution

Here, we briefly discuss the main research contributions of this thesis.

- **Testing Permission Re-delegation Vulnerabilities using Taint Analysis**

  We extend the permission re-delegated threat model to clearly distinguish between intended benign app integration and unintended vulnerable cases. We call this

extension the Android Wicked Delegation (AWiDe). We propose and compare two automated approaches to detect this vulnerability. In the first approach, we rely on static taint analysis to detect whether values used in privileged actions depend on potentially unintended(malicious) data. The second approach relies on dynamic taint analysis with automatically generated input values.

These approaches have been empirically validated on more than 300 apps. We also compare the results of static and dynamic taint analysis approaches. Results show that popular apps are affected by AWiDe vulnerabilities. We also demonstrate the relevance of this problem by elaborating proof-of-concept attacks based on the vulnerabilities detected by our approaches.

This work has led to a research paper published to the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft'16) [26].

- ***PREV*: Testing Permission Re-delegation Vulnerabilities using NLP and Genetic Algorithm**

  We present *PREV*, a publicly-available implementation of a fully automated framework for detecting permission re-delegation vulnerabilities in Android apps, based on static analysis, natural language processing, machine learning, and genetic algorithm. The approach relies on behaviors learnt from similar apps in order classify whether a vulnerability report is true positive or not.

  This work is then validated by a large-scale empirical assessment, in which 11,796 apps were analyzed for learning the permission re-delegation models, and 1,258 real world apps were analyzed to detect permission re-delegation vulnerabilities. A comprehensive comparison with different static analysis based approaches is then performed in terms of precision and recall.

  This work is currently under revision in an international peer-reviewed top journal.

- **Detecting Second Order Permission Re-delegation Vulnerabilities**

  We describe and qualify a peculiar permission re-delegation vulnerability that we call *Second Order Permission Re-delegation*. We then, propose an approach based on static analysis to detect this vulnerability and to synthesize prof-of-concept attack scenarios that document how this vulnerability can be exploited. We, then, empirically assess the overall approach on more than 10k real world apps, showing

first of all that this vulnerability is prominent among apps and, second, that our approach is effective in detecting and testing it.

This work is currently under submission to an international peer-reviewed conference.

- **Detecting Anomalous Information Flows in Android Apps**

  We present Anflo, a publicly available implementation of an automated, fast approach for detecting anomalous flows of sensitive information in Android apps through a seamless combination of static analysis, natural language processing, model inference, and classification techniques.

  We then present an extensive empirical evaluation of our approach based on 596 real world subject apps, which assesses the accuracy and run-time performance of anomalous information flow detection. We also analyze malware using our approach to see if anomalous information flow detection can be applied in order to identify malicious flows.

  This work has led to a research paper published to the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft'18) [25].

1.4 Structure of the thesis

This dissertation is organized as follows:

Chapter 2 gives general background in Android apps and security mechanisms the Android framework enforces in order to protect end user data and privacy. Furthermore, it briefly highlights the communication mechanism Android provides for app integration which is the basis of this work.

Chapter 3 establishes the threat model we adopt in this thesis. It provides example scenarios demonstrating normal and malicious app-to-app communications and motivates the work presented in this thesis.

Chapter 4 presents our static and dynamic taint analysis based automated approaches to detect permission re-delegated vulnerability. It presents the design, implementation and evaluation and comparison of results both approaches.

Chapter 5 presents the approaches based on static analysis, natural language processing, machine learning, and genetic algorithm. Design and implementation of the approach is presented followed by the results from a large scale empirical evaluation and comparison with other static analysis based approaches.

Chapter 6 defines Second Order Permission Re-delegation and establishes why current sate-of-the-art multi-app analysis techniques are not effective in detecting this vulnerability. It then presents the design and implementation of the proposed static analysis based detection technique by evaluation of the approach.

Chapter 7 presents our automated approach for detecting anomalous flows of sensitive information in Android apps through a seamless combination of static analysis, natural language processing, model inference, and classification techniques. It also presents the design, implementation and evaluation of the approach.

Chapter 8 compares this thesis with different related work by highlighting the similarities and differences.

Chapter 9 draws the conclusion of this thesis and discusses the possible future directions.

CHAPTER 2

Background

In this chapter, we provide background required to grasp the technical content of the next chapters. First, we discuss the Android framework and the security mechanism it adopts. We, then, discuss about Android apps, their building blocks and their interactions.

2.1 The Android Platform

Android is the market leader operating system for smartphones. At the core of the Android platform lies a Linux kernel that provides several native features such as security and low-level memory management. On top of the Linux kernel lies the hardware abstraction that provides interfaces to device hardware capabilities to API framework. The Android framework provides a rich set of features through APIs written in Java.

2.2 Android Apps

End-users can extend the features available on the device by installing new apps. Android apps are mostly written in Java. Starting from October 2017, however, developers have also the option to write Android apps using Kotlin, a statically typed language that runs on the Java Virtual Machine (JVM). Additionally, developers have the possibility to combine these languages with C/C++ through the Java Native

Interface(JNI). The Android development kit compiles the codes along with other resources that the developer include, such as multimedia files, in to a package called APK.

Once developers build their app, they can use the Android centralized app store, called Android Play Store, to make their apps available to millions of devices. The Android Play Store is now the biggest app store in the world, and has 3.6 million apps available for download in March 2018 [84].

***System apps*:** By default, Android comes with pre-installed system apps such as the Phone app or the Browser (e.g., Chrome) app. These system apps are read-only and the user cannot remove or change them. Since system apps are trusted and considered secure, they are usually provided with several additional privileges than user apps.

***User apps*:** End-user-installed apps are usually downloaded and installed from Google Play. These apps can be removed at any time the user pleases. Apps installed by the user cannot directly access each other's resources as they are placed in security sandbox where each app can only access its own resources. These apps are granted less privileges compared to system apps and, moreover, have to request the end-user for a any needed permission.

2.2.1 Apps Components

Android apps are composed of four types of components. Each type of component provides distinct functionality and has distinct lifecyle. Moreover, each component could serve as an entry point to the app based on interactions with the end-user or another app, or event notifications by the system that the app is listening to (e.g., 'battery low' event).

Activity: Activities are the app components that the end-user sees and interacts with visually. Apps can have multiple activities each representing one single screen the end-user interacts with. For example, a photo gallery can have one activity that shows the list of photos in the device and another activity that shows one full photo when the user selects it. Activities can be launched by the user by clicking the icon from the list of apps in the device or by interacting with another apps (See Section 2.4). Apps usually have one main activity that serves as entry point when it is launched by the end-user when clicking on its icon.

Broadcast Receiver: Broadcast receiver is another entry point to an app, through which the system and other apps can deliver broadcasts announcements to the app. The main purpose of a broadcast receiver is to let the app register and listen to system-wide broadcast announcements, for example a broadcast announcing the battery level is getting low, without the end-user interaction. That is, broadcast receivers can listen to broadcasts even when the app is not running. Some broadcasts can only be sent/announced by the system (e.g., the battery level getting low), while some other broadcasts can only be received by apps that have the required privilege (e.g., new SMS message has been received by the device).

Service: The other possible entry point to an app is a service. A service is an app component that performs long-running operation in the background (i.e., without a user interface). A very good example of a service is a music player where the end-user can use other apps while the music app is playing music in the background. An activity, for example, can start a service and let it run or *bind* itself to the service so that it can interact with it.

Content Provider: A content provider allow apps to manage shared persistent data. Through the content provider, other apps can query or modify the data in a secure and efficient manner if the content provider allows it. An example of a content provider is the Android's Contacts Provider that manages the user's contact information. Apps can use this provider to add, edit or remove contacts from the phone's Contacts Directory.

In this dissertation, all app components except Content Providers are subject to vulnerability analysis. Content Providers are not considered because their operation is only related to data and are not affected by the threat model we address.

Apps define their components and other essential information about the app in a special XML file called `AndroidManifest.xml`. Figure 2.1 shows a sample Android manifest file for an example app `com.demoapp` with the definition of all the four kinds of components. Line 14 defines an activity component with the name `com.demoapp.MainActivity`. Lines 28, 33 and 38 define a Broadcast Receiver, a Service and Content Provider, respectively.

2.3 Android Security

Android relies on the security features provided by the underlying Linux kernel — apps run as a separate process with their own set of resources and preventing them from

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.demoapp."
  android:versionCode="1"
  android:versionName="1.0" >
  <uses-sdk
    android:minSdkVersion="19"
    android:targetSdkVersion="21" />
  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity android:name= "com.demoapp.MainActivity"
              android:label="@string/app_name"
              android:permission="com.demoapp.MY_CUSTOM_PERMISSION">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name= "android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name= "android.intent.category.DEFAULT" />
        <data android:scheme="https" android:host="*" />
      </intent-filter>
    </activity>

    <receiver android:enabled="true"
              android:exported="true"
              android:name="com.demoapp.DemoReceiver">
        . . .
    </receiver>
    <service android:enabled="true"
              android:exported="true"
              android:name="com.demoapp.DemoService">
        . . .
    </service>
    <provider android:name="com.demoapp.DemoDB"
              android:authorities="com.demoapp.demodb">
        . . .
    </provider>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CALL_PHONE" />
  </application>
</manifest>
</xs:schema>
```

Fig. 2.1 An example Android Manifest file

interfering with each other. This process is called *Sandboxing*. Moreover, Android enforces fine-grained permissions that apps have to request in order to access sensitive resources (e.g., GPS sensor) or interact with critical components of apps (e.g., Phone app).

2.3.1 Sandboxing

Apps are automatically assigned a unique Linux user ID (UID or app ID) and group ID (GID) at installation time. Each app is given a private data directory that only that app has read/write access to and world-accessible resources on the device. This process creates kernel-level sandboxes of apps such that they do not interfere with the execution and resource of other apps. Apps that belong to the same developer can share the same UID if they are signed with the same key. These apps, however, can access each other's resources.

2.3.2 Permission

A sandboxed app that cannot access any resource would not be interesting as it would be limited. In order for apps to provide a richer functionality, Android provides a fine-grained permission to allow or disallow resource access at the app layer. Apps have to explicitly request for the permission to access a given resource, and the end-user has to grant them. Apps request permission for a given resource (e.g., Internet) by specifying in the manifest file as shown on Figure 2.1, line 42. On Android versions prior to 6, the user did not have the option to selectively grant permission. That is, if an app is installed, it means that all the permissions requested during installation time are granted to the app. Later versions, however, provide the possibility to grant and revoke permissions selectively and apps have to check and request the permissions at run-time. An example of an app requesting access to photos, media and file on the device is shown in Figure 2.2. End-users also have the possibility to remember their choice such that the system automatically performs the previous choice (either deny or allow) until the end-user changes it.

Apps can also define custom permission in order to protect their components from other apps. An app's component can specify in the manifest file the permissions needed by other apps in order to communicate with the given component as shown in Figure 2.1 line 16. This specifies that an app that wants to communicate with the activity `com.demoapp.MainActivity` needs to hold the custom permission `com.demoapp.MY_CUSTOM_PERMISSION`.

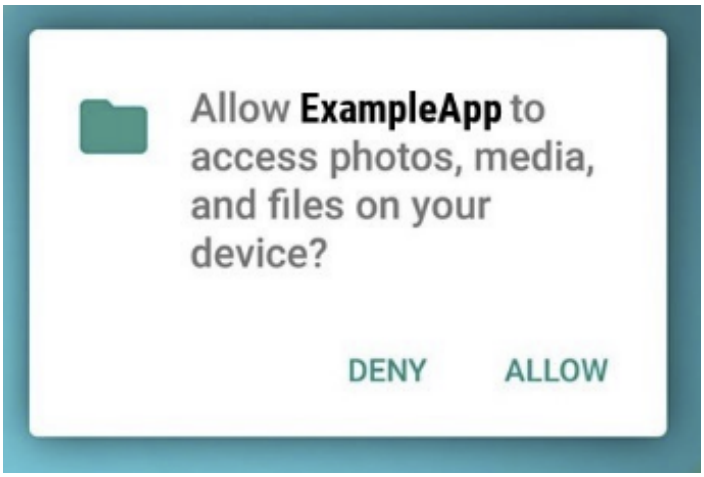


Fig. 2.2 Example of runtime permission request

2.4 Android Inter-Component Communication (ICC)

Inter-component communication (ICC) is the backbone of the Android framework that allows different individual components to manage tasks and resources independently but cooperatively. Android provides several mechanisms for the different components of Android apps to communicate with each other. Though the traditional Linux inter-process communication (IPC) techniques such as network sockets and shared files are possible, developers are recommended to use the Android system IPC functionalities (such as Intents or Binders) as they provide better security such as checking or enforcing caller permissions [40].

***Intent based ICC*:** An intent is a special message a component can send in order to request an operation to be performed by other components. These other components could be Activities, BroadcastReceivers, or Services of the same app or other apps that are currently installed on the device. For instance, when a component of a messaging app wants to take a photo, it could request a component of a camera app to take the picture and get back the image file by sending an Intent that specifies this operation.

Intent based ICC can be either *(1) implicit* where the callee app specifies its requests but the destination component that handles the request is determined by the framework itself (or the device user in case of multiple components), or *(2) explicit* where the callee specifies which component it wants to communicate with.

An Intent contains the details of the operation to be performed. The *component* specifies the name of the component that this intent will be sent to (for *explicit* intents), the the *Action* and the *Category* specify the operation to be performed, the *Data* contains a reference to a resource needed to complete the task (e.g., the document to

display), while *Data* specifies the MIME type of the data and *Extra* contains a bundle of additional data as key-value pairs.

Apps can subscribe for receiving Intents for those tasks that they are available to accomplish on behalf of other components (called capabilities). To this aim, they declare an `intent-filter` in their manifest. The `intent-filter` in Figure 2.1 on line 17 specifies that this component is the main entry-point to the app. This is because the `action` field specifies the action `MAIN` (line 18) which has the meaning this activity is the entry-point of the app when its icon is clicked in the Home screen's Launcher. On the other hand, the `category` has the value `LAUNCHER` (line 19) which specifies this activity should appear in the Home screen's Launcher (where the end-user can see the icon of the app).

The `intent-filter` on line 21, however, specifies this activity is also capable of handling/showing HTTPS URLs on behalf of other apps. This is achieved by specifying the `VIEW` (line 22) action and the `https` scheme (line 24).

By default, a component is private, i.e., it is only accessible by other components from within the same app. The declaration of `intent-filter` in a component makes the component public. A public component can be accessed by any app unless it is protected with a custom permission (See Section 2.3). However, components can override this by explicitly setting the `exported` attribute to `true` to make the component public or to `false` to make the component private. The `exported` attribute takes precedence over `intent-filter`. The Broadcast Receiver in Figure 2.1 is public as it sets the `exported` attribute to `true` on line 34.

CHAPTER 3

Threat Model and Motivation

In this chapter, we present the general threat adopted in this dissertation.

In order to provide interesting functionalities, apps require to have privileges so that they access sensitive resources, such as the camera or contacts. Additionally, to complement each other with features, apps interact with each other via inter-component communication (ICC) message called Intent. Below, we present nominal case of ICC followed by an attack scenario.

3.1 Nominal Inter-Component Communication

Figure 3.1 shows two apps — a messaging app and a camera app. When the messaging app requires to take a photo (camera feature), it does not need to re-implement this feature as it is already implemented (and exported) by another app (the Camera app) that comes with Android. The Android framework provides the mechanism for the messaging app to send a request to the available camera apps. In this way, an app can reuse a functionality exported by other apps without re-implementing it. For example, in Figure 3.1, taking photo on behalf of other apps is a feature or functionality exported by the camera app. The camera app defines public interfaces where it accepts messages requesting this task. The messaging app invokes this public interface with the required data (e.g., path where the image should be stored). The camera app takes the photo and stores in the requested path. This scenario of inter-component communication between the messaging app and the camera app is an example of integration of Android

apps to complement each other with features. However, this feature might be exploited to mount attacks if it is not implemented properly.

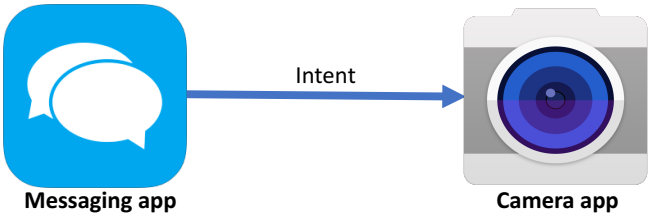


Fig. 3.1 Nominal Inter-Component Communication

3.2 Permission Re-delegation Vulnerability

Figure 3.2 shows an ICC scenario with permission re-delegation vulnerability. App B is granted the permission to make a phone call. In addition, app B exposes some functionalities (public interfaces) to other apps. On the other hand, an attacker app A is not granted any permission. End-users may install A assuming it is harmless as it is not granted any permission, and thus, no access to sensitive data and resources.

An attempt to perform a phone call by app A would be blocked by the Android framework as app A misses the required permission (`CALL_PHONE`). App B instead can make a phone call as it holds this permission. Thus, app A invokes app B's public interfaces in unexpected way, such that app B performs a phone call (a privileged task) on behalf of app A. The system will allow the phone call because app B initiates it and holds the required permission.

Apps that are granted special privilege should not, however, leak sensitive data or delegate their privileges to other apps that do not have the same permissions. If an app exposes its granted permissions to other apps, we say that the app is potentially vulnerable to permission re-delegation. Permission re-delegation [33] (also called confused deputy) vulnerability is a kind of privilege escalation vulnerability that may occur when a privileged app is hijacked and it performs a privileged task (e.g., making a phone call) on behalf of a less privileged (possibly malicious) app. Apps that are granted special privileges should not contain permission re-delegation vulnerabilities; otherwise they could be target of attacks posing security risks to the end-user. Less privileged apps could exploit such vulnerabilities by crafting malicious Intent messages intended to make a vulnerable app misuse its permissions to leak sensitive data (e.g.,

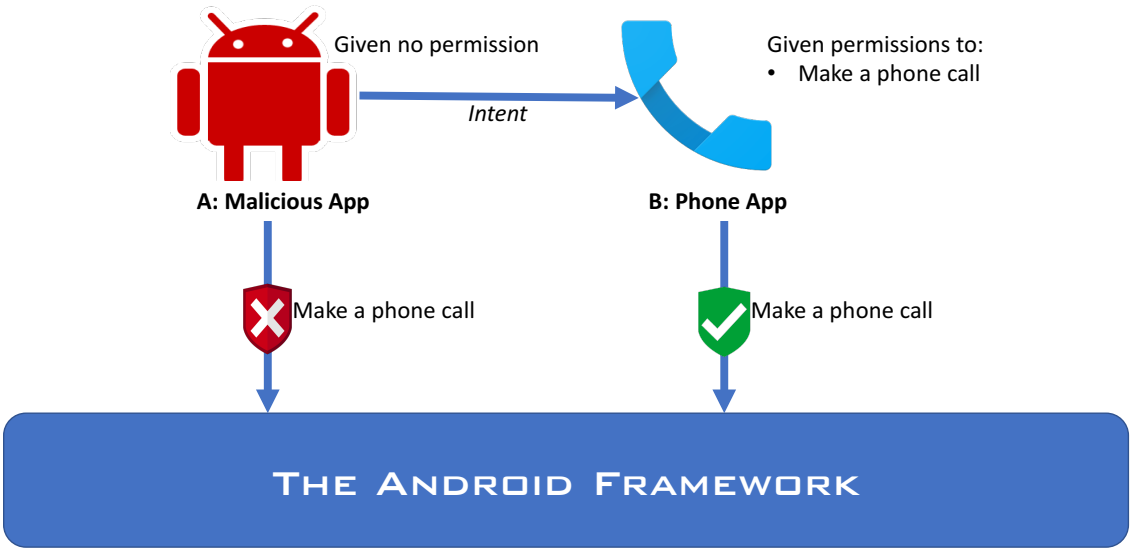


Fig. 3.2 Permission Re-delegation

GPS position or contacts), modify sensitive information (such as contacts or app private data) or perform costly operations (calls or SMS to premium numbers).

Due to the high time-to-market pressure or lack of security expertise, developers might overlook the need for security assessment of their apps [30]. This vulnerability, however, affects both user apps and system apps. System apps are expected to be of high quality as they are part of Android. However, a quick search in CVE databases[1] provides several results related to permission re-delegation (permission-bypass or privilege-escalation). For example, *PowerNotificationWarnings* (CVE-2015-3854), *SMSDispatcher* (CVE-2015-3858 and CVE-2016-3888) and *com.android.phone* (CVE-2013-6272), are few examples of affected apps and components that come pre-installed in the Android device.

The Android Play Store is now the biggest app store in the world, and has more than 3.6 million apps available for download as of March 2018 [84]. Considering the fact that this class of vulnerability is prevalent even among high quality system apps, the need for a better and precise automated security assessment of apps motivates our research in this dissertation.

An effective security assessment technique should enable us identify the *unexpected* or *surprising* ways an app could be exploited to mount permission re-delegation attacks.

[1] https://nvd.nist.gov/vuln/search

In the following chapters, we will propose novel techniques to automatically detect and test permission re-delegation vulnerabilities in Android apps.

CHAPTER 4

Testing Permission Re-delegation Using Taint Analysis

The Android Play Store hosts more than 3.6 million apps as of March 2018 [84]. It is common to find several apps in the market having similar purposes. For example, a simple search for *torch light* returns more than 200 apps. Therefore, when a new potential app idea arises, an app developer needs to rush and develop the app before similar apps become available on the market by competitors. Since the first apps that appear on the market usually get more accepted by the users, posting an app early can help the developer gain market share.

Due to the high time-to-market pressure, developers usually spend more time on the app's user experience and appealing user interface and they might overlook the need for quality and security assessment of their app [30]. Other quality aspects (possibly including security issues) might be delayed for future updates. Therefore, an effective, fast and automated tool to reveal defects in apps' code would be highly beneficial.

Though the Android operating system enforces barriers to isolate apps that are developed by different authors and prevents them from interfering with each other, poorly developed apps might still pose security risks to the end-user privacy. In fact, apps might unintentionally expose their access to sensitive resources, such as phone contacts, or privileged actions, such as making phone calls. As described in Chapter 3, this problem is known as *permission re-delegation* [33]. However, delegation can be legitimate when it is available because of the developer's intention, or a vulnerability

when added by mistake and is exploited in a way that was different from the developer's intentions.

In this chapter, we extend the permission re-delegated threat model to clearly distinguish between the intended and unintended (vulnerability) cases. We call this extension the Android Wicked Delegation (AWiDe for short). The vulnerability consists in a vulnerable app executing privileged tasks on behalf of an attacker app (similar to the re-delegation case) but with the additional preconditions that (i) the privileged task is executed with data controlled by the attacker, and (ii) without performing adequate validation of such data.

We propose and compare two automated detection techniques for AWiDe vulnerabilities. In the first approach, we instantiate static taint analysis to detect whether values used in privileged actions depend on potentially malicious data. The second approach relies on dynamic taint analysis with automatically generated input values. Dynamic taint analysis tracks data dependencies during execution and detects when data from the attacking apps is used in privileged actions.

These approaches have been empirically validated on more than three hundred apps. Results show that popular apps are affected by AWiDe vulnerabilities. We also demonstrate the relevance of this problem by elaborating actual attacks based on the vulnerabilities detected by our approaches. The corresponding app developers have confirmed one of these vulnerabilities.

Contribution: the proposed approach in this chapter explores a different approach compared to state-of-the-art techniques such as [33, 59, 98] in detecting permission re-delegation vulnerabilities by relying on failed input (intent) validation. In this way, our approach aims to minimize the false positive reports produced by state-of-the-art analysis techniques. Moreover, different from the state-of-the-art, our approach combines both static and dynamic analysis to detect permission re-delegation vulnerabilities and compares and contrasts the result of each analysis technique.

4.1 Threat Model

4.1.1 Background

The Android framework offers two features that are relevant to our approach (i) allowing the integration and collaboration of apps from different authors but still (ii) guaranteeing a certain level of separation to enforce security and confidentiality. Separation among

apps is achieved by enforcing *sand-boxing* and by adopting a permission based system to regulate the access to sensitive resources (See Section 2.3).

An app can delegate a specific task to another app without actually knowing which apps are available in the current device to accomplish that task. For example, different end users might have different apps installed that are able to take pictures, but the requester app does not need to know which one to delegate a priori. For the requester, it is enough that the delegated app can take pictures.

Android has been developed in such a way that a requester app has just to specify what should be done (and/or with what data), and the framework identifies an app that is able to accomplish the task (if there is any). Apps use Intents for inter-component communication (ICC). Intents contain the description of the operation that the requester app needs to be performed. Apps specify in their XML manifest files what services they expose to other apps, with the so-called *intent-filters*. The framework relies on the manifest file to decide what app to delegate.

Figure 4.1 shows a fragment of the manifest file of an Android app that provides a service to expand URLs shortened by using *goo.gl*. URL shortening is a service to substantially reduce long and complex URLs to few characters but still pointing to the original locations. URL shortening is useful in social networks (e.g., Twitter), in messaging apps or to reduce typing when manually entering URLs. In the Android context, however, shortened URLs might break the seamless integration of apps. For example, assume an e-mail that contains a URL pointing to *maps.google.com* is received. Clicking on this link opens the Maps app and displays the location on the map. However, if this URL is shortened, the framework does not know to which app to delegate and therefore, opens the browser even if there is a dedicated app for the URL. URL expander apps help fix this problem by *intercepting* the Intent message, expand the URL and then delegate to the appropriate app. The official market contains apps that offer this feature (e.g., *URL Expandroid*[1] and *Short URL Evaluator*[2]).

From Figure 4.1 we can see that the app defines an intent-filter to accept *goo.gl* shortened URLs to expand to the original full URL. According to the filter definitions, Intents for this app must specify the *VIEW* action and the *DEFAULT* category. The data part of the Intent specifies the short URL to expand with scheme *https* and host *goo.gl*. The system inspects contents of an *implicit* Intent and if they match the intent-filter , then the app is delegated to perform the action. If there are more than

[1] net.studiofly.android.yuzu

[2] com.github.nicolassmith.urlevaluator

one app that can perform the same task, then the user is prompted to chose from a list of apps to complete the action with.

A second app (e.g., an e-mail app) can send an implicit Intent with action *VIEW*, category *DEFAULT* and data *https://goo.gl/IhH0Ix*. The Intent matches the intent-filter in Figure 4.1 and therefore it would be delivered to the app (assuming no other app has registered the same intent-filter). This app will contact the appropriate URL shortening service to retrieve the original URL then send an implicit Intent with the expanded URL so that the appropriate app picks it.

```
<activity android:name= ".ExpandUrl"
            android:label="@string/app_name" >
   <intent-filter >
      <action android:name="android.intent.action.VIEW" />
      <category android:name= "android.intent.category.DEFAULT" />
      <data android:scheme="https" android:host="goo.gl" />
   </intent-filter >
</activity>
<uses-permission android:name="android.permission.INTERNET" />
```

Fig. 4.1 An example Android Manifest file

4.1.2 Motivating Example

We say an app is vulnerable to permission re-delegation attack when it *unintentionally* performs a privileged task on behalf of another app that does not have the required permission. Figure 4.2 shows an example of such attack where the vulnerability is due to inadequate validation of an Intent message that is used in a privileged operation.

The scenario includes two apps: a benign victim *URL Expander* app U and an *attacker* app A. Let's assume that U defines the manifest file in Figure 4.1. U is granted the special permission *INTERNET* to query the URL shortening service. An intent-filter is defined to offer the URL expansion service. U extracts the URL to expand from an incoming Intent message. It then queries the URL shortening service and expands the URL. U is meant to query only trusted URL shortening services that are listed in the intent-filter (e.g., *goo.gl*).

However, U fails to correctly validate Intents. When a different URL that is not listed in the intent-filter of U is explicitly sent to it, instead of rejecting the Intent, U attempts to expand the URL by directly connecting to it. A malicious app A that intends to, for example, leak sensitive information can use this app as a proxy to pass sensitive data to the attacker controlled URL.

Even if the attacking app A is not fully trusted, it was installed by the final user because it requested no permission, thus it was assumed harmless. Moreover, sandboxing is expected to preserve security by guaranteeing separation among apps. This assumption fails to hold when U contains security vulnerabilities that will let a malicious app bypass the security mechanism.

In our attack context, A sends an explicit Intent to U with the data field set to *https://evil.com/log?deviceid=1234*, a host controlled by the attacker. However, U's validation is defective and does not properly check the host name to block untrusted domains, but queries it directly. The malicious host returns a URL that encodes a malicious executable binary as expanded version. A can do multiple queries in order to collect several parts of a malicious binary. A can register for the URL that is returned by U and capture the expanded URL(s) so that it can dynamically load the combined and decoded malicious binary.

As a result of inadequate Intent validation, U performs a privileged action on behalf of the attacker A using data controlled by A. Eventually, A is able to download malware on demand (a most recent version and, possibly, device specific) without INTERNET permission. Moreoever, if A has access to network, this attack allows it to download a malware stealthily as network usage would be attributed to U.

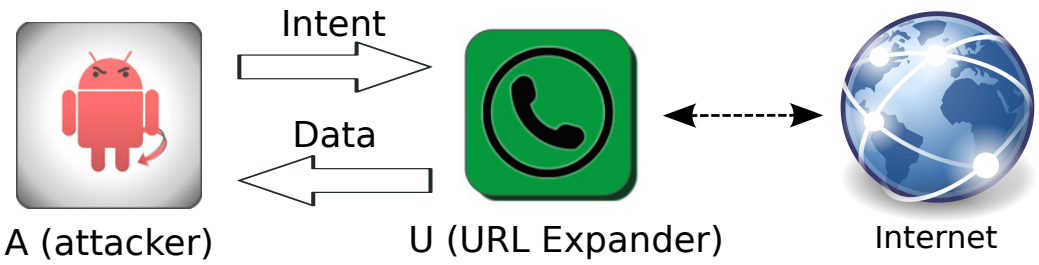


Fig. 4.2 Example of attack scenario.

To summarize, this attack happens when an app registers an intent-filter to let the Android framework know about the app's capability of a certain service but fails to validate when an explicit Intent is sent to it before performing the privileged task.

4.1.3 Vulnerability Preconditions

Based on the previous attack scenario, we identify the preconditions that a vulnerability should meet in order to be exposed to attacks based on inadequate message validation.

Sensitive actions can be performed only by apps that are granted the permission to access the corresponding resource. An attacker app that misses the permission to access sensitive resources might resort to other vulnerable apps that hold the needed

access right. The goal of the attacker is to make the vulnerable app execute privileged actions on its behalf. Thus, the first precondition of this vulnerability is:

> ***Precondition PR_1: Privileged API call.*** *While performing the action requested by the Intent, the vulnerable app calls a privileged API.*

Using the example of Figure 4.1, this corresponds to an app that, after receiving an Intent from a requester attacking app, accesses the Internet invoking the API *HttpURLConnection.connect()* that requires the special permission INTERNET. This is a case of permission re-delegation, as described by Felt et al. [33], because the vulnerable app performs a privileged task on behalf of a second app that misses the required permission.

However, as acknowledged by Felt et al., permission re-delegation includes also cases of legitimate delegation. Our threat model goes beyond that and requires additional preconditions to distinguish between legitimate delegation and attacks. A second precondition is usage of Intent data in the privileged operation. Data used in the privileged operation should come from the Intent, which is possibly controlled by the attacker. Thus, the following precondition should be also satisfied:

> ***Precondition PR_2: Attacker data.*** *A privilege API is called using data coming from the Intent sent by the attacker app.*

In the running example, the string with the query to the URL shortening service contains the shortened URL to be expanded comes from the Intent. However, this can be a legitimate case of delegation if a *goo.gl* shortened URL is received. A real attack scenario would use a malicious data that a properly implemented app would normally discard.

Despite intent-filter s have not been defined explicitly for security, nonetheless intent-filters reflect the intention of the developer on the format of incoming data. In case data is received that clearly violates the intent-filter , it violates the developer's intention and hence, is most likely a case of unintentional delegation.

The Android framework inspects intent-filter s before delivering *implicit* Intents to identify the appropriate destination app(s). Thus, the destination app is confident to receive only valid messages, i.e., messages that are compatible with the protocol defined in the intent-filter. However, *explicit* Intents contain the name of the target component, therefore, Android delivers the Intent directly without verifying its content.

While in implicit Intents validation is performed by the Android framework, in explicit Intents, validation of Intent is up to the receiving app which may implement incomplete or partial validation.

The final precondition needed for delegation to be malicious is missing Intent data validation. A vulnerable app accepts and processes an invalid Intent that does not satisfy the conditions defined in the corresponding intent-filter .

> ***Precondition PR_3: Incomplete data validation.*** *The validation of Intent data implemented in the app code is defective, because it accepts a message that violates the app intent-filter.*

An Intent with the *shortened* URL *https://evil.com/log?deviceid=1234* sent to this app clearly violates the intent-filter of Figure 4.1, because the host *evil.com* is a server controlled by the attacker, instead of the expected trusted *goo.gl* server.

A defect that satisfies all of these three preconditions represents a security issue: an invalid Intent is processed as if it was valid and Intent data is used to call privileged API. In this way, an attacker app manages to control the execution of a privileged action without having the corresponding permission. We call this an *Android Wicked Delegation*, (AWiDe).

The objective of the rest of this chapter is to elaborate and assess two automated approaches to identify when an app contains AWiDe vulnerabilities. Our analysis consists in tracing (either statically or dynamically) malicious data from Intent messages, and in detecting when a malicious data is used in sensitive operations, namely privileged API calls.

4.2 Static Analysis for Vulnerability Detection

The first approach we propose is based on static taint analysis. We instantiate static flow analysis to trace the dependencies on Intent values through the app control flow. When (potentially malicious) Intent values are used in privileged API calls, the tool reports a candidate AWiDe vulnerability.

4.2.1 Taint Analysis

Taint analysis is intended to track the taint status of values throughout the app control flow. A problem is reported whenever a tainted value is used in a security sensitive statement, called the *sink*. Taint status is propagated on assignments to the variable on the left hand side, when an expression on the right hand side uses a tainted value. Tainted variables become untainted upon sanitization by means of special functions or when they are assigned untainted values, such as a constant value or an expression that contains no tainted value.

```
1  Intent intent = getIntent();
2
3  if (intent != null) {
4   Uri phoneNo = intent.getData();
5   String message = intent.getStringExtra("msg");
6
7   message = "welcome!";
8
9   SmsManager smsManager = SmsManager.getDefault();
10  smsManager.sendTextMessage(phoneNo, null, message, null, null);
11 }
12 return;
```

Fig. 4.3 An example code demonstrating Intent data being used in sensitive sink.

Figure 4.3 shows a snippet of an example Android app's component code that uses Intent data (highlighted red on line 1) in a sensitive sink (highlighted red on line 10). On line 1, Intent data is retrieved using `getIntent()`. Since the incoming input is potentially malicious, the variable `intent` on this line is tagged *tainted*. On the other hand, the API call `sendTextMessage()` is considered a sensitive sink as its invocation requires holding the `SEND_SMS` permission.

On line 4 and 5, taint tags are propagated on assignments to the variables `phoneNo` and `message`, as the expressions on the right hand side use a tainted value (`intent`). Values become untainted upon sanitization by means of special functions or when they are assigned with untainted values, such as a constant value or an expression that contains only untainted values. After execution of the statement on line 7 (highlighted blue), the variable `message` becomes untainted as it is assigned a constant string.

If this app's component is public, any app can use this component to send SMS to an arbitrary number. This potential security problem depicted on the example code in Figure 4.3 (the data-flow path involving statements 1, 4 and 10) can be identified using either static taint analysis [86] or dynamic taint analysis [22]. Static taint analysis is formulated as a flow analysis [81] problem, where the information propagated in the control flow graph is the set of variables potentially holding tainted values. Static taint analysis detects candidate vulnerabilities as paths in the control flow that make tainted values reach a sink (i.e., a vulnerable statement). However, static taint analysis is conservative, because flow information is propagated through all the paths, potentially including infeasible paths that cannot be executed.

Dynamic taint analysis, instead, updates the taint tags of program variables during execution. Often, taint tracking is implemented by either instrumenting the code or

the run-time (e.g., the VM). Instrumentation adds tainted flags and adds the code required to update flags on assignments. Alternatively, using a modified execution environment (e.g., a virtual machine), deployed to keep track of the tainted flag of program variables. Dynamic taint analysis is more precise but incomplete, because it analyses the app just with respect to the current observed execution, at the cost of some runtime overhead. A problem is reported when a value tagged as tainted is used in a vulnerable statement (the sink).

4.2.2 Static Analysis

In our analysis, FlowDroid [8] was used for static taint analysis. FlowDroid is a static analysis tool that reasons about information flow at the level of variables, by finding paths from sources to sinks. This tool was initially conceived to trace data-flow and detect privacy leaks in the form of sensitive data sources (e.g., from the contact lists) that are disclosed on sinks, for example through network messages. It also accepts options to run analysis on different precision levels.

We instantiated static taint analysis by changing the configuration of sources and sinks to meet our analysis requirements. In our threat model, we are interested in tracing how Intent data flows in the app and potentially reaches a privileged instruction, i.e., an API that requires a special permission to be executed.

Sources: According to precondition PR_2, tainted data comes from intents sent by other apps. The API call used to read the incoming intents is the method `Intent.getIntent()`. This call returns an *Intent* object that contains the message payload. All fields of this object will be considered tainted.

Sinks: According to precondition PR_1, the sinks are all the method calls to the Android framework that require a special permission. However, there is no complete documentation of what are the privileged APIs in Android. Thus we reused the result of the work by Au et al. [10], that relied on dynamic analysis to extract the API - permission mapping. Sinks consist of more than 30K method signatures spread across the entire Android library classes.

Decreasing Precision Level: FlowDroid supports analysis with different levels of precision. Therefore, we set a time limit for the analysis that the most precise level requires. We stop the analysis of an app that can not complete within 10 minutes, and we restart the analysis with a less precise configuration. We iterate this process until the analysis completes or all the precision levels are attempted. The time budget was selected by observing that FlowDroid is in general very fast and that when it does not complete within 10 minutes, it is unlikely that it will ever finish.

Filtering: The report filled by FlowDroid specifies what pair of source-sink is connected by a data flow. The tool report is parsed and its results are stored in XML files for easy analysis.

Due to over-tainting (and possibly less precise analysis option), the report sometimes contains cases where our source (i.e., method `Intent.getIntent()`) is not involved at all. This can be also caused by a known bug[3] on FlowDroid's taint wrapping which, in some conditions, propagates taint tags from a field to its class. When flow analysis involves multiple classes. For example, if the field `ClassA.field1` becomes tainted and the analysis spans on multiple classes, FlowDroid marks the whole class `ClassA` as tainted. Thus, all operations on other fields of the same classes (e.g., `ClassA.field2`) are marked as potentially tainted, introducing false positives.

Thus, we have to filter the cases reported by FlowDroid and keep only those source-sink pairs that explicitly mention `Intent.getIntent()` as source. This filtering has been implemented as an XPATH expression that was evaluated on the reports of FlowDroid output as XML file.

Intent-Filter Violation: With respect to vulnerability preconditions of Section 4.1.3, so far static analysis checked just preconditions PR_1 and PR_2, i.e., data from the Intent is used in a privileged API call. However, static analysis does not check precondition PR_3, i.e., the validity of Intent data. The flows reported as vulnerable are supposed to be checked manually. This consists in verifying if the source code of the app checks Intent data for the same conditions that are specified in the intent-filter. Moreover, to qualify the security report as a true positive, an input should be elaborated that satisfies all the vulnerabilities preconditions.

4.3 Dynamic Analysis for Vulnerability Detection

The second approach we propose relies on dynamic analysis. The execution of the app under analysis requires proper input data, i.e., intents. Intents are automatically generated that satisfy precondition PR_3 (see Section 4.1.3), by checking whether they violate the intent-filters of the app under analysis. Then, trace analysis is used to identify what executions hit a privileged API (precondition PR_1). Eventually, precondition PR_2 holds for those intents whose data is used in privileged API calls. Dynamic taint analysis is applied to check this last condition.

To perform dynamic analysis, we adopt a more complex sequence of steps, required to generate input data and to monitor the app execution. App execution is monitored

[3]https://github.com/secure-software-engineering/soot-infoflow-android/issues/43

to collect execution traces and then to perform dynamic taint analysis of selected execution traces.

4.3.1 Data Generation

Vulnerability precondition PR_3 requires input data (intents) to violate the intent-filters defined by the app under analysis. An intent-filter specifies a set of conditions and *all* the conditions should hold for an Intent to be considered valid (i.e., if the Intent was implicit, the Android framework would have delivered the Intent to the app). Consequently, to violate a filter, an Intent needs to violate just one condition. In order to maximize condition coverage, we adopt a heuristic approach to generate input data. Each generated input violates at least one condition in the intent-filter.

Before starting the generation of test data, the manifest file of the app is parsed to extract information about the Intent filters. Since there could be many filters, each of them is subject to test data generation. The first Intent to be generated is the *prototype*, a valid Intent that *does* satisfy all the conditions in the filter.

For example, for the Intent-filter in Figure 4.1, we automatically create a prototype Intent with:

- `action=`*`VIEW`*,
- `category=`*`DEFAULT`*, and
- `data=`*`https://goo.gl/IhHOIx`*

To check the validity of the prototype Intent, we resort to the validation implemented in the Android framework. We send the generated prototype as implicit Intent so that the Android framework identifies all the apps that are compatible with it. If the app under test is selected as a potential destination of the prototype Intent, we are sure that the prototype is valid. It is possible that multiple apps could handle the Intent, in which case the framework prompts the user to select an app to complete the task with. To minimize this interference, no other user apps are installed on the testing environment other than the app under test and a helper app.

A new testing Intent is generated by mutating the prototype such that it negates one of the conditions in the filter. The mutation operators are:

- **Mutate action**: an action other than the one specified in the intent-filter will be set;

Different actions are collected from Common Intents in the Android documentation[4]. A null value or a random value that is different from the prototype action is selected from set. In the previous example prototype, the action could become,

`action=`*`VIEW`* → `action=`*`NULL`*

- **Mutate category**: similar to the action field, category will be set to value different from the one specified in the intent-filter of the component;

  Similarly, the Common Intents set is consulted for the common categories. For prototype category, an example mutation is as follow,

  `category=`*`DEFAULT`* → `category=`*`BROWSABLE`*

- **Mutate data**: the different parts of Intent data are mutated separately.

  The data field of an Intent has several parts (e.g., host, scheme, , path, pathPattern). This mutation operator, however, considers the *scheme*, *host* and *path*fields. The scheme is randomly chosen from (`tel`, `sms`, `mms`, `http`, `https`, `ftp`, `file`, `content`). The host and the path are randomly generated.

  In the case of the running example, a mutation can randomly change in either of the following ways,

  `data=`*`https://goo.gl/IhHOIx`* → `data=`*`ftp://goo.gl/IhHOIx`*

  `data=`*`https://goo.gl/IhHOIx`* → `data=`*`https://fpa2djfl.com/IhHOIx`*

After mutation, we need to check whether the test intents violate the intent-filter (precondition PR_3). To assess this, we rely on the Android framework. We send test intents as implicit intents (i.e., without specifying any destination). The Android framework will rely on intent-filters to decide the proper destination. We only keep those intents that were *not* delivered to the app under test, because they violate its intent-filter and are not supposed to be delivered to the app. If these intents are explicitly sent to the test app, then the app should discard them. If the app processes these intents we can say that precondition PR_3 is met.

4.3.2 Trace Analysis

Generated intents need to be sent to the app under test to verify if they satisfy precondition PR_1. Recall that PR_1 is satisfied if the app processes intents that do not

[4]https://developer.android.com/guide/components/intents-common

match with the intent-filter and the app's business logic requires access to privileged API.

By stripping away the permissions an app holds and executing it with the generated Intent would raise an exception if the app executes a privileged API. By analyzing execution traces (failures), we would be able to tell if an Intent has caused the execution of a privileged API.

Trace analysis compares execution traces of the original app ($APP_{original}$) and the modified app with stripped permission ($APP_{stripped}$).

Both $APP_{original}$ and $APP_{stripped}$ are instrumented to log execution traces and exceptions (both handled and unhandled). Instrumentation is done directly at the byte-code level by using an AspectJ[5] [52], the aspect oriented version of Java, aspect that injects new code to record all the method executions and all the exceptions thrown at runtime.

After explicit Intent is sent to both $APP_{original}$ and to $APP_{stripped}$ in distinct sessions, execution traces are collected and compared. Only the cases where the execution traces are different are interesting for us, as it means there has been an error in one of the executions. As the only difference between $APP_{original}$ and $APP_{stripped}$ is in the permissions, a variation in the trace means that the specific Intent makes app $APP_{stripped}$ trigger a privileged API call that causes an error due to an insufficient permission. In this case, the test could expose potential security vulnerability because the invalid Intent is not rejected (precondition PR_3) and forces the app to perform a privileged action (precondition PR_1).

At the end of trace analysis we know exactly what privileged APIs are executed, these are the calls that caused a permission violation difference between $APP_{original}$ and $APP_{stripped}$. The calls detected by us might also include APIs that are potentially missed by Au et al. [10] (as acknowledged by the authors, their list could be partial).

The only remaining check is to verify if Intent data has been used in these privileged calls (precondition PR_2). This process is summarized in Figire 4.4. Both $APP_{original}$ and $APP_{stripped}$ are run on Android emulator, the traces are collected and analyzed for possible inconsistencies. Depending on the cause of the inconsistency, a vulnerability report is produced.

[5]https://eclipse.org/aspectj/

Fig. 4.4 Dynamic analysis workflow

4.3.3 Dynamic Taint Analysis

An invalid Intent that makes the app execute a privileged instruction does not necessarily represent a vulnerability. In fact, the app may perform privileged actions for reasons that are not directly related to the received Intent. For example, the app may need to access the Internet to update a local cache or to check for updates. We register a potential security problem when precondition PR_2 is also satisfied, i.e., when the privileged API call uses potentially malicious data that comes from the invalid Intent.

To verify if Intent data is used in a privileged API call, we instantiate dynamic taint analysis. All data fields from the Intent are marked as tainted and taint tags are propagated on assignments during the execution. The privileged APIs that caused error during trace analysis are then checked to see whether they are called using any tainted data.

TaintDroid [29] is used for dynamic taint analysis, with some modifications to adapt the tool to our security analysis. TaintDroid is a dynamic taint analysis tool for Android, intended to reveal sensitive information disclosure during execution. In TaintDroid, data is tagged as tainted when it comes from a privacy sensitive source, such as the address book or the GPS. Sinks are represented by all those possible ways a piece of information can leave the device, such as through network transmissions or outgoing text messages. Dynamic taint tracking relies on a modified version of the Android framework that checks the taint tags of variables on assignments in byte-code, and conservatively in native calls, inter-process messaging and file system.

First, taint sources of TaintDroid differ from the ones we need for our analysis. In fact, TaintDroid was intended to monitor the flow of private data, while our analysis is intended to track Intent data. Second, while sinks in TaintDroid are points where data leaks from the device (e.g., via the network), in our analysis sinks are all the privileged API calls (that have been identified by trace analysis), including also access to local

private databases not considered originally by TaintDroid. For example, the act of writing a new contact is a privileged action (it fails if the app is not granted the proper permission), but it is not a sink for TaintDroid, because no information leaves the device. Despite these differences, the propagation of taint tags in TaintDroid perfectly fits our needs.

To adapt TaintDroid to our needs, we deploy an aspect written in AspectJ, that sets the taint tag for incoming Intent (sources) and that checks the taint tag the data on privileged API calls (sinks).

While we have a fixed source, i.e., the calls to `Intent.getIntent()`, sinks might change because they depend on the result of trace analysis. An aspect is automatically generated to intercept the privileged API calls detected by trace analysis. After this aspect is applied to the app byte-code, the app is executed with test intents on TaintDroid to check whether tainted data is used in sink calls. The automatically generated aspect reports if test intents satisfy precondition PR_2.

4.4 Empirical Validation

This section reports a study that assesses our vulnerability detection when analyzing a set of 329 apps to identify potential AWiDe security defects.

NB*: the experimental results presented in this section are based on Android versions prior to Marshmallow, which as of October 2018 account for 29% of the Android devices. On recent versions, some permissions, such as the* `INTERNET` *are granted to all apps by default and hence, the attacking app could simply declare this permission in its manifest and use its own permission (without requesting the user) instead of exploiting other apps.*

4.4.1 Research Questions

The aim of this section is to investigate the following research questions:

- $RQ_{4.1}$*:* What is the accuracy of static and dynamic analysis in detecting Android Wicked Delegation vulnerabilities?
- $RQ_{4.2}$*:* How much time does it take to static and dynamic analysis to check for Android Wicked Delegation vulnerabilities?

The first question aims at studying the detection accuracy of static and dynamic analysis. The second question is, instead, focused on the time taken to complete the analysis. Answering these research questions would suggest what is the most appropriate tool to support developers of Android apps.

4.4.2 Metrics

To answer these questions, we apply the proposed detection techniques to the same case studies. We then validate detection results by manually filtering reported vulnerabilities. During this process we record these metrics:

- *True positives:* Number of vulnerable apps correctly detected as vulnerable by a tool;
- *False positives:* Number of safe apps incorrectly detected as vulnerable by a tool (false alarms);
- *Analysis time:* How long a tool takes to analyse the apps;

While *True positives* and *False positives* are meant to quantify the accuracy of the techniques, *Analysis time* is an indicator of the speed of the two analyses.

4.4.3 Subject Apps

Our approaches work on compiled apps; therefore, availability of source code is not a requirement for the analysis. However, we decided to opt for open source projects because they offer the possibility to inspect the app source code and manually verify the correctness of vulnerability detection.

The F-Droid repository [32] represents an ideal setting for our experimentation, because (i) it includes real world and popular apps that can be found also in the official market, and (ii) apps can be downloaded with their source code, for manual validation of the security reports delivered by the tools.

The whole app catalog was downloaded in October 2014, and it consisted of a total of 1,216 apps. Sometimes, multiple versions were available for the same app. In these cases we considered the most recent version.

Apps that do not have components with intent-filter other than the entry component (commonly known as *main activity*) are not included as they usually expect Intent from the Android Launcher with action `MAIN` and category `DEFAULT`. If they contain other intent-filters, however, then they are considered. Among the 329 apps that define

intent-filters, more than 70% are also present in the official Google Play Store and some of them (in particular games) are also quite popular. We queried the official Android store to study the popularity of the apps considered in our study. Figure 4.5 shows the number of user reviews and the number of installs for the subject apps reported in the official Android store. While the number of installs ranges from 30 to 300 million, the average number of installs of the case study apps is 2.3 million. The minimum number of reviews is 0 and the maximum is 700K, while the average number of reviews is close to 13K.

The number of installs and reviews supports the claim that our study considered popular apps that can be found in the official store.

4.4.4 Results

Table 4.1 Detailed intermediate results of the static and dynamic analysis.

(a) Static analysis.

Analysed apps	With source-sink path	Filtered for "getIntent"
273	141	53

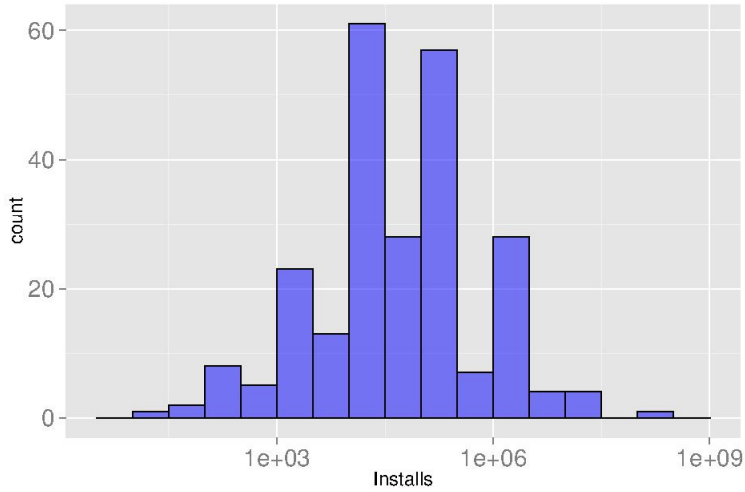
(b) Dynamic analysis.

Analysed apps	Apps calling privileged APIs	Tainted data on sink
329	84	10

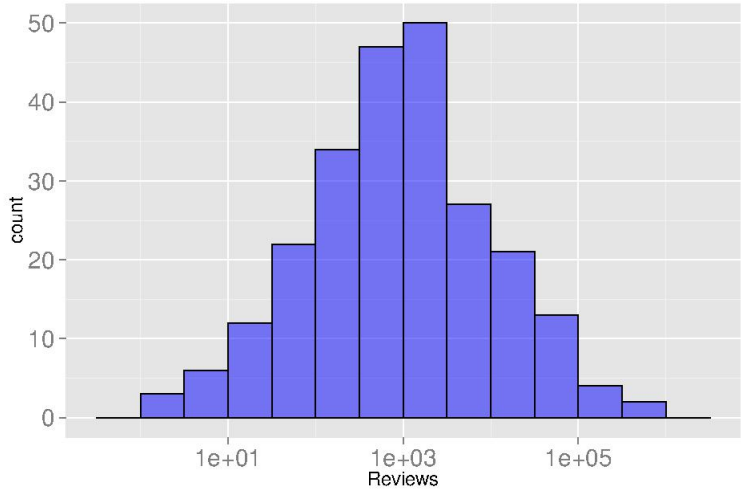
Detailed results of static analysis are shown in Table 4.1(a). Out of the 329 apps that can actually be targeted by delegation attacks, i.e. they define intent-filters and permission requests in their manifest XML files, static analysis could complete only on 273 apps (time out in 17% of the cases), because of limitations of the tool we used. Among them, 141 apps were reported as containing a flow from the source to a sink. Further automatic filtering was required to exclude vulnerability reports that did not

Table 4.2 Apps analysed with different precision level.

Options used	Number of apps
Default configuration	222
–NOCALLBACKS	40
–ALIASFLOWINS	4
–PATHALGO SOURCEONLY	1
–ALIASFLOWINS, –NOCALLBACKS, –NOEXCEPTION	6
(Timeout)	56
Total	329



(a) App installs



(b) App reviews

Fig. 4.5 Popularity (installs and reviews) of case study apps according to the official Android store.

include `getIntent` as source. The final result of static analysis consists of 53 apps classified as affected by the AWiDe vulnerabilities.

Table 4.2 shows the level of precision used in the analysis. 222 apps were analysed with the default analysis option. A slightly less precise analysis (without considering callbacks) allowed completing static taint analysis in 40 apps. Even less precise analysis was required in fewer cases. However, on 56 apps, even the less precise analysis did not deliver any result by the time out (17% of the cases).

Table 4.1(b) summarizes the results of dynamic analysis. The 329 apps with intent-filters and permissions were instrumented and analysed. Trace analysis revealed 84 apps that invoked privileged APIs after receiving an invalid Intent that violated their intent-filters. Among them, our modified version of TaintDroid detected 10 cases as vulnerable, because a tainted data from an Intent was actually used in privileged API calls.

Table 4.3 Final results of automatic tools in detecting AWiDe vulnerabilities.

Metric	Static analysis	Dynamic analysis
True positives	9	9
False positives	44	1
Analysis time	135 minutes	5 days

Manual inspection of source code was required to assess the accuracy of the analysis, to check whether reported vulnerabilities were true positives. Manual inspection of source code consists of (i) verifying that Intent data is used in the privileged call; (ii) that no (or partial) validation of Intent data is enforced before using it on privileged call; and (iii), only for static analysis, elaborating an input value that satisfies all the vulnerability preconditions.

The comparison of the results of the two approaches is summarized in Table 4.3. Among the 53 apps classified as vulnerable by static analysis, we identified 9 apps as true positive because they contain at least one vulnerable flow from source to sink that misses adequate data validation. The remaining 44 cases were false positives due to the reasons mentioned in Section 4.2 (conservative analysis and over-tainting).

After manual inspection of the results of the dynamic analysis, we classified the reported cases as 9 true positives and 1 false positive. The false positive case was due to a bug in our implementation that was difficult to solve, because it involves interference between taint tags for our analysis and privacy leak taint tags originally propagated by TaintDroid. Table 4.4 shows details of the actual discovered vulnerabilities, with the special permission related to the privileged API call. Only 3 apps have been correctly classified as vulnerable by both static and dynamic analysis.

We analysed more in depth the other cases, detected only by one or the other approach. Among the 6 cases missed by dynamic analysis, in 5 cases none of the test case generated by dynamic analysis could execute the vulnerable path detected by static analysis. The remaining case is a limitation of our implementation on a particular filter protocol.

Conversely, considering the 6 cases missed by static analysis, in 2 cases static analysis got stuck and did not complete. In the remaining 4 cases, the vulnerable data flow traverses a system library not modelled by the static analysis tool, limited just to intra-component analysis. For example, *Zirco Browser* receives (potentially malicious) URLs from other apps through Intent messages and displays them using WebView, a system component. This inter-component interaction is missed by FlowDroid hence reporting no flow.

Then, we compared the amount of time required to complete the analysis. While static analysis was quite fast (135' for the 273 apps that completed), dynamic analysis, even if fully automated, took more time (5 days). Long execution time of dynamic analysis is due to the large set of input values to test. Each Intent required at least two distinct executions in the instrumented environment (trace analysis requires running an app with and without permissions). An additional execution with TaintDroid was required for those apps that passed the trace analysis phase. The set of inputs for the 329 apps consisted of 9,756 intents. It should be noted here that we do not perform combinatorial testing, because each test Intent violates just one condition among those from the intent-filter.

A major difference was also observed in number of apps with incomplete analysis. While dynamic analysis always delivered a result, FlowDroid (static analysis) could not complete the analysis on 56 apps (17%), for reasons related to limitations of the tool version we used for the experiment.

4.4.5 Outcome

All in all, experimental results allow us to answer the research question in the following way:

- Both static analysis and dynamic analysis report the same number of true positives (both 9 cases, however, the overlap is on 3 apps). On the other hand, static analysis reports many more false positives than dynamic analysis (44 vs. 1, respectively).

The outcome of the experiments also allows us to formalize the following considerations:

- Static analysis is faster than dynamic analysis (minutes compared to days).
- Android Wicked Delegation vulnerabilities affect real-world apps. In a set of 329 apps, we found 15 apps that match the threat model defined by us.

Table 4.4 Apps correctly detected as vulnerable by static and dynamic analysis.

App name	Static Analysis	Dynamic Analysis
DAAP	Internet	—
Scid on the go	Internet	—
Short URL Evaluator	Internet	—
TunesViewer	Internet	—
OpenSudoku	Internet	Internet
Call Meter 3G	Internet	Internet
ACV	Internet	—
Moss	Internet	Internet
ZooBorns	Set Wallpaper	—
SipDroid	—	Call Phone
Lumicall	—	Call Phone
Car Cast	—	Internet
Ermete SMS	—	Read Contacts
Zirco Browser	—	Internet
Crosswords	—	Internet

4.4.6 Attacks

To show that the vulnerabilities discovered by us can be exploited to port serious attacks, we elaborated proof-of-concept attacks[6] starting from the results of our analysis techniques. Table 4.4 summarizes the permissions that can be abused on vulnerable apps.

The first attack is on *SipDroid*, an app with more than 1 million downloads from the Android market. It is an open-source SIP client that is granted the `CALL_PHONE` permission to make phone calls. A particular activity of this app defines an intent-filter to accept implicit intents with data schemes related to messaging (schemes are `sms://` and `smsto://`). However, the app's implementation fails to enforce these schemes on incoming intents. Therefore, it accepts and processes explicit intents with scheme `tel://` that does not match the intent-filter. As a result, the app dials any number specified in the Intent.

This vulnerability can be used to mount complete attacks either by, (i) making the vulnerable app dial a premium number or, (ii) performing a USSD/MMI attack [46, 15]. USSD/MMI codes are numeric strings between the "*" or "#" characters. They are

[6]These attacks are not meant to cause real damages or steal real data, their purpose is to demonstrate that a vulnerability is exploitable. In security terms they are called *proof-of-concept attacks*.

meant to access services supplied by the mobile operator or to access phone functions. This attack can lead to the factory reset of the device on a very popular device from a mainstream vendor [16], or play with operator configurations (e.g., #793#, which resets the voicemail password on T-Mobile USA).

Lumicall is another telephony app that borrows code from SipDroid, which makes it vulnerable to the same attack. Vulnerabilities in both these apps have been reported to the authors[7].

An attack related to information leak can be ported to an app detected by both the techniques. The app is *OpenSudoku*, an open-source version of the popular *Sudoku* game. Besides the normal game play, one of its functions is the possibility to import new sudoku schemes from the Internet, using the special permission `INTERNET`.

While the intent-filter allows to open streams only towards a subset of URLs (e.g., a specific MIME-Type or *.opensudoku* file extension), these restrictions are not enforced by the app code. The URL from an invalid Intent is directly interpreted as a source to load the sudoku scheme definition data.

A malicious app without permission to access the Internet can still leak sensitive data (e.g., the device serial number *1234*) by sending data in parameter as GET request. The URL should refer to a server controlled by the attacker, for example *evil.com*. The complete attack contains the URL *http://evil.com/?deviceid=1234*. When receiving this Intent, *OpenSudoku* loads the URL and leaks the device serial number to the malicious server. The vulnerability we spotted in this app has been reported to the developer who confirmed it[8].

Another interesting vulnerability that might allow an attacker publish as well as download file from a malicious host exists in the app *Short URL Evaluator*. The details of the attack is similar to the running example of Section 4.1.2. It can also be found in the vulnerability report filed by us[9].

[7] https://code.google.com/p/sipdroid/issues/detail?id=1183
https://github.com/opentelecoms-org/lumicall/issues/32

[8] https://code.google.com/p/opensudoku-android/issues/detail?id=170
https://code.google.com/p/opensudoku-android/issues/detail?id=171

[9] https://github.com/nicolassmith/urlevaluator/issues/42

4.5 Discussion

4.5.1 Considerations

Static analysis reports more false positives: Static analysis and dynamic analysis report the same number of true positives, but static analysis reports a larger number of false positives. This result suggests that both analysis methods are effective in detecting defects. However, it is important to consider the large number of false positives that are also involved (44 versus 1). Valuable development time should be invested in inspecting each potential security problem. High false positive rate could potentially make app developers miss their time-to-market objective. Thus, dynamic analysis, with lower number of false positives, is an effective option. In fact, vulnerabilities that are reported by dynamic analysis are more likely to be instances of real problems that deserve high priority in manual investigation.

Dynamic analysis requires more time: We observed a huge difference in the amount of time required by the two analyses to complete. While static analysis is very fast (minutes), dynamic analysis took a lot more (days). This probably reduces the possibility of using dynamic analysis as a tool to check code quality by app-store managers, i.e., where a large set of apps has to be considered. In that context, the more lightweight approach of static analysis is probably more appropriate. However, if we consider the case of an app developer, who just needs to inspect the quality of a single app during development, both static and dynamic analysis are viable approaches. In fact, dynamic analysis of one app took on average one hour, which is still acceptable in this second context.

TaintDroid is no longer maintained and the latest Android version supported is 4.3. Moreover, the dynamic analysis suffers the same limitations TaintDroid is prone to, such as inability to propagate taint tags to native method.

Manual filtering of security reports: Apart from the different false positive rate, security reports differ also in terms of information provided. Reports from static analysis just lists what *sink* statement is reachable with tainted input data. The developer needs to figure out by herself/himself how data can flow from the source to the sink. In our experience, this was quite easy when the source and the sink were located in the same class, or in the same method, because the control flow was quite easy to understand. However, it was much harder when the source and the sink were

in different classes. In some cases the flow was complicated by the presence of multiple methods in multiple classes, and manual inspection was more time consuming.

The report of dynamic analysis, instead, consists of an executable scenario. It is a concrete instance of an input value that triggers the discovered vulnerability in a candidate vulnerable flow. To classify one of these cases, the full understanding of the whole control flow was not required, and the developer could just focus on the particular execution flow taken by the test. Often, step-wise execution in debug mode was enough to quickly figure out the vulnerable data flow and confirm or discard the reported case.

Fast understanding and filtering of executable scenarios makes dynamic analysis reports more appropriate for a fast time-to-market business model.

User interaction: Static analysis considers all the potential flows, including those that may potentially require external events (e.g., user intervention or system event) to execute a privileged API call. On the other hand, since it is not included in our execution scenarios, dynamic analysis does not consider any user interaction.

For this reason, AWiDe vulnerabilities detected by dynamic analysis are more direct and stealthy. A malicious app could exploit them directly, just by sending an accurately crafted intent, without the help of any subsequent event to execute the intended privileged API. Secondary events (user or system event) might be required to exploit vulnerabilities reported by static analysis. However, a user who might become suspicious could block these second cases of attacks when the confirmation required to perform an attack pops up out of the blue.

Vulnerabilities reported by dynamic analysis should be considered with higher precedence, and they would require immediate attention by the developers.

4.5.2 Limitations

We acknowledge these limitations on the two approaches we propose.

Static analysis checks all the vulnerability preconditions (see Section 4.1.3) but one. The only non automated check is whether input data violates the intent filter (i.e., precondition PR_3). As such, the report of static analysis is not fully compliant with our threat model. Manual inspection was required to identify cases of missing or incomplete validation in the source codes.

The current implementation of FlowDroid is affected by over-tainting, resulting in false positives. Communications with the maintainers of FlowDroid revealed that over-tainting would be reduced after they fixed some implementation limitations. Indeed,

some cases of over-tainting that we observed in our experiment are connected to a documented bug of this tool[10].

The approach based on static analysis cannot propagate taintedness through constructs that FlowDroid cannot resolve statically, such as reflective calls and calls to native code, which can be present in Android apps. Although some support could be implemented to model native code, reflective calls are hard to be handled statically in the general case. This limitation does not affect dynamic analysis, where actual calls can be observed.

In our dynamic analysis tool, trace analysis runs the app without permissions. An error due to insufficient privileges reveals the first protected API call. As a result, for each input value, only the first AWiDe vulnerability is detected by dynamic analysis. This defect should be fixed by developers, to make our dynamic analysis tool able to detect potentially subsequent security defects. Security checks performed just before release could pose a limitation to the applicability of dynamic analysis to a fast time-to-market development model. Security verification should be more integrated in the daily development activity.

Our dynamic analysis tool does not model user interaction. On one side, this allows detecting stealthy attacks that a final user would not notice. On the other side, this restricts the number of attacks that dynamic analysis would detect. For this reason, we recommend to use static analysis to complement the limitations of dynamic analysis.

Starting from Android version 6, a different permission scheme is adopted. Common permissions, such as the Internet access, are granted by default to all the apps without the need of end-user confirmation. This change will probably reduce the number of apps that are vulnerable according to the AWiDe threat model, because many cases of vulnerability were related to the Internet permission (see Table 4.4). However, as of October 2018, more than 28% of the Android devices still run earlier versions of Android[11]. Thus, misuse of the Internet permission is still a threat on most of the Android devices and our approach is still relevant to identify security defects related to Internet. Moreover, our approach is effective on the remaining set of sensitive permissions, for example to make phone calls.

[10] https://github.com/secure-software-engineering/soot-infoflow-android/issues/43
[11] http://developer.android.com/about/dashboards/index.html

4.6 Chapter Summary

In this chapter, we presented a variant of permission re-delegation vulnerability we called the Android Wicked Delegation that additionally requires a privileged API to be executed with an incoming Intent data without validation. We also proposed two approaches, based on static and dynamic taint analysis, to automatically identify this class of security defects. Static analysis showed to be faster than dynamic analysis, but less reliable and affected by more false positives.

CHAPTER 5

Security testing of permission re-delegation vulnerabilities using Natural Language Processing and Genetic Algorithm

In the previous chapter, we proposed a technique for detecting instances of permission re-delegation vulnerabilities based on taint analysis, namely, static taint analysis and dynamic taint analysis to reveal if vulnerable apps process invalid Intents to eventually execute a privileged API with values derived from the Intent. As acknowledged in Section 4.5.2, these approaches, however, suffer from some limitations, such as being limited to cases where data from Intent is required to reach a sensitive sink. Moreover, the approaches did not differentiate between intentional task requests and actual vulnerabilities. In this chapter, we propose a more general approach.

As a cornerstone feature in Android, performing a sensitive action based on requests from other apps could be an intended feature of apps and may not always be a permission re-delegation vulnerability. To limit false alarms, an accurate analysis approach should distinguish between intended task requests and actual permission re-delegation vulnerabilities.

For example, apps that need to initiate a phone call usually do not implement this feature because they assume this feature to be already available by another app in the smart phone (e.g., the system Phone app). They simply send a request for this task to the Phone app that services such incoming requests by initiating the phone call (privileged action). This could be a privileged feature exposed by communication

apps to other apps. It is an intended feature and is neither a programming mistake nor a permission re-delegation vulnerability. However, a vulnerable version of the Phone app[1] also exposed another feature that could be used by other apps to wipe out phone data and perform factory reset. This second scenario is very uncommon among communication apps and it represents a permission re-delegation vulnerability. This vulnerability (also known as permission bypass or privilege escalation) has been seen in several Android system apps (apps that are either developed by Google or device vendors and come preinstalled on Android devices) such as *PowerNotificationWarnings* (CVE-2015-3854), *SMSDispatcher* (CVE-2015-3858 and CVE-2016-3888) and *com.android.phone* (CVE-2013-6272)).

The approach presented in the previous chapter would detect both of these scenarios as potentially problematic, because they both involve inter-app task request leading to execution of a privileged action, namely, initiating a phone call and wiping data that require the `CALL_PHONE` and `MASTER_CLEAR` permissions, respectively. While initiating a phone call is an intended and legitimate behavior, wiping the phone data, however, is the vulnerability that should be reported. To accurately report only actual security problems, cases of permission re-delegation vulnerabilities must be distinguished from legitimate cases of permission re-delegation.

In this chapter, we present a novel *P*ermission *RE*-delegation *V*ulnerability detection (*PREV*) framework, which seamlessly combines static analysis, natural language processing (NLP), machine learning, and genetic algorithm-based test generation techniques for *precise* detection of permission re-delegation vulnerabilities in Android apps.

More specifically, given a large training set of benign and non-vulnerable (denoted as *safe*) high quality apps, we first apply NLP on their app descriptions and use clustering to create clusters of highly similar apps. Then, for each cluster, we apply static analysis to infer *permission re-delegation behaviors* of the apps in the cluster, i.e., privileged actions that may be performed upon receiving incoming requests. Based on this information, we build *permission re-delegation model* of the cluster, which characterizes common permission re-delegation behaviors of the apps in that cluster.

Given an app under test (AUT for short), we first determine the cluster it belongs to based on its app description (similar declared features); we then check whether the AUT has one or more permission re-delegation behaviors that deviate from the model

[1]Vulnerability in the Samsung TouchWiz phone dialer http://www.androidauthority.com/touchwiz-vulnerability-data-wipe-117800/

of the cluster. If that is the case, each anomalous behavior is reported as a candidate permission re-delegation vulnerability.

We then apply genetic algorithm to generate proof-of-concept attacks that exploit candidate vulnerabilities and confirm whether the AUT is indeed vulnerable and exploitable. This helps us reduce false positives as well us during manual investigation.

In our empirical assessment, we built permission re-delegation models based on the top 11,796 "safe" apps downloaded from the official Android app store (*Google Play*). We evaluated our approach in terms of the accuracy in detecting permission re-delegation vulnerabilities in 1,258 real world apps (not from those top 11,796 apps) that are also available on Google Play store. In total, *PREV* detected *30* vulnerable apps among 1,258 apps with zero false alarm. We reported our findings to the app developers. We also compared our approach with *FlowDroid* [8] , *Covert* [13], and *IccTA* [56], static analysis-based approaches that can detect permission re-delegation vulnerabilities in Android apps. *FlowDroid* produced 8 false alarms, *Covert* produced 17 false alarms, and *IccTA* produced 15 false alarms. *FlowDroid* and *IccTA* did not detect any actual vulnerabilities; only *Covert* detected one actual vulnerability, which was also detected by *PREV*.

Contributions: To summarize, the main contributions presented in this chapter are:

- Extending the threat model of Chapter 4 to support a more general attack scenarios
- *PREV*, a fully automated framework for detecting permission re-delegation vulnerabilities in Android apps, based on static analysis, natural language processing, machine learning, and genetic algorithm.
- A publicly-available implementation of *PREV*[2].
- A large-scale empirical assessment, in which 11,796 apps were analyzed for learning the permission re-delegation models, and 1,258 real world apps were analyzed to detect permission re-delegation vulnerabilities.
- A comprehensive comparison with static analysis tools in terms of precision and recall.

[2]http://selab.fbk.eu/PREV/

5.1 Background

In this section, we present some background concepts used in the rest of the chapter.

Figure 5.1 shows a fragment of the manifest file of our motivating example app (Section 5.2) that will be used in this chapter. In this example, the app requests for the permission *CALL_PHONE* to initiate phone calls. Apps that hold this permission are allowed to initiate a phone call without user interaction. This app defines an activity (tag `<activity>`) — with an intent-filter (tag `<intent-filter>`). Activity *DialerActivity* can be requested by other apps to initiate a phone call, for example, when a link with phone number (e.g., `href="tel:+1234"`) is clicked within a web page, the browser sends an intent containing the `DIAL` action, the `DEFAULT` category and phone number to this app.

```
<activity android:name= ".DialPadActivity">
  <intent-filter>
   <action android:name="android.intent.action.MAIN" />
   <category android:name= "android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
<activity android:name= ".DialerActivity">
  <intent-filter>
    <action android:name="android.intent.action.DIAL" />
    <category android:name= "android.intent.category.DEFAULT" />
    <data android:scheme="tel"/>
  </intent-filter>
</activity>
<uses-permission android:name="android.permission.CALL_PHONE" />
```

Fig. 5.1 Snippet of AndroidManifest XML file.

5.1.1 Genetic Algorithm

Genetic algorithm is a population-based meta-heuristics technique proposed for solving optimization problems. An example of optimization problems is generating test inputs that are likely to expose specific program behaviors of interest. The genetic algorithm is inspired by natural evolution from biology [49]. It searches for an optimal solution by gradually evolving an initial population of random solutions through generations. Individuals more near to the final solution are rewarded with a higher probability of transmitting their chromosomes to future generations. Fittest solutions are combined together with the hope of generating fitter ones, until the optimal solution is found.

The pseudocode of the abstract genetic algorithm is shown in Figure 5.2. Initially, the algorithm generates random individuals (candidate solutions). Then, the algorithm loops through three main steps until the termination conditions (optimal solution found or timeout) are met. The steps are:

1. *AssessFitness*: this step computes the fitness of each individual solution and selects a set of fittest individuals (i.e., candidate solutions that are likely to generate the optimal solution).

2. *Crossover*: this step first pairs the individuals selected in the previous step. Then, it generates offspring from the pairs by swapping portions of their chromosomes.

3. *Mutate*: this step mutates the offspring generated in the previous step by applying certain mutation operators (such as flipping the bits). This breeds the next population for the next iteration.

Variants of the genetic algorithm are discussed in literature, with different implementations of these steps.

```
1 function GA () {
2   initialize_population_of_random_individuals()
3   while (termination_conditions_are_not_met) {
4     AssessFitness()
5     Crossover()
6     Mutate()
7   }
8 }
```

Fig. 5.2 The abstract genetic algorithm

5.2 Motivating Example

In this section, we recall permission re-delegation vulnerability with a motivating example suitable for highlighting the research gap.

Assume a Dialer app designed to replace the default Android dialer. Figure 5.3 shows the intended behavior of the Dialer app. When an Intent is sent from the DialPadActivity within the Dialer app or other apps such as the browser, a phone call is initiated. The Dialer app also allows the user to use the DialPadActivity to change or read phone configurations such as device serial number.

5.2.1 Updated Threat Model

Apps that are granted with privileges should not contain permission re-delegation vulnerabilities; otherwise privileges could be the target of attacks. Less privileged apps could exploit such vulnerabilities by crafting malicious Intent messages intended to make a vulnerable app misuse its permissions to leak sensitive data (e.g., GPS position or contacts), modify sensitive information (such as contacts or app private data) or perform costly operations (calls or SMS to premium numbers).

Figure 5.4 shows an example of attack scenario in which a permission re-delegation vulnerability in an app is exploited by a less privileged app to execute a privileged operation or API.

Recall that, we define a *privileged API* as an Android API that requires a special permission to be executed.

The scenario includes two apps: a benign but vulnerable *Dialer* app D and an *attacker* app A. Let us assume that D specifies the manifest file in Figure 5.1. Among others, D is granted with the special permission `CALL_PHONE` for initiating phone call. An intent-filter is defined to allow other apps to make a phone call via this app. When A sends an Intent message to D, D extracts the destination phone number from the message and attempts to initiate a call. We define a *public entry point* of D, a method in the code of D that is executed by the Android framework when an Intent is sent to D, e.g., the `onCreate` method in Figure 5.5.

Figure 5.5 shows the Intent handling snippet of the Dialer app. The code starts by getting the Intent sent to the app (Line 1). If the `action` in the Intent is `DIAL`, the app extracts the data from the Intent (Lines 2-4). The data contains the `scheme` and the phone number. If the `scheme` is "tel", the Dialer app then extracts the number associated to this `scheme` (Lines 7-8). Then depending on the number, that is, if the number starts with * or #, the app attempts to perform a configuration related task (e.g., getting serial number of the phone if the number is *#06#); otherwise it initiates a phone call.

In this example, the Dialer app has permission re-delegation vulnerability — the app makes configuration changes based on data that comes from other apps. This feature is supposed to be internal, i.e., it should only be performed if the number is entered by the user using the dial-pad. However, by mistake, the developer exposed the capability to change configurations to other apps. As shown in Figure 5.4, malicious apps could exploit this vulnerability, for example, by sending the code that wipes out the phone data [16, 77] or making rogue phone calls [23].

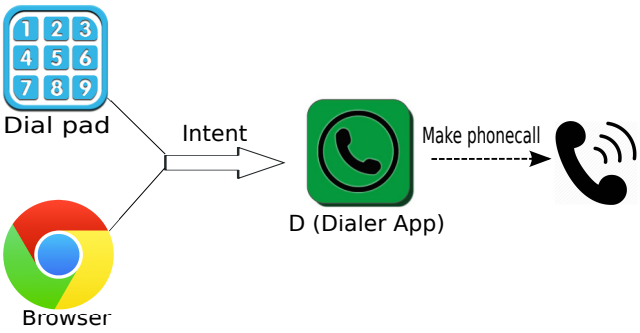


Fig. 5.3 Intended behavior of the Dialer app

Fig. 5.4 An example of attack scenario.

For example, when A sends an Intent message to D with action `DIAL` and data `tel:*#060#`, D performs the specified task without the user interaction. In essence, D performed a privileged operation on behalf of A using the data controlled by A without any user interaction.

Even if the attacker app A is not fully trusted, users may still install A since it requests no permission and can be assumed harmless. Even though Android treats distinct apps as distinct principals to provide separation among apps, security cannot be guaranteed when D contains such a permission re-delegation vulnerability. Exploiting this vulnerability in D, A is able to change phone state (e.g., wipe out the data from the phone or get the device serial number) without the required `MODIFY_PHONE_STATE` or `READ_PHONE_STATE` permission.

5.2.2 Vulnerability Preconditions

Based on the attack scenario explained above, we identify two preconditions that should be met in order to classify a case as a real permission re-delegation vulnerability.

Privileged APIs can be executed only by apps that are granted with the permission to access the sensitive resources. An attacker app that lacks the permission to access sensitive resources needs to resort to an app that holds the required access right. It needs to make the app execute privileged APIs on its behalf without the intervention of the user. Thus, the first precondition of this vulnerability is the following:

```
1  Intent intent = getIntent();
2  if (intent != null &&
3   intent.getAction().equals("DIAL")) {
4   Uri uri = intent.getData();
5
6   if (uri != null) {
7     if ("tel".equals(uri.getScheme())) {
8      String number = uri.getSchemeSpecificPart();
9
10      if (number.startsWith("*") ||
11                    number.startsWith("#")) {
12        // e.g., request USSD, MMI (wipe phone)
13        changeConfigurations(number);
14       } else {
15        makePhoneCall(number);
16       }
17    }
18  }
```

Fig. 5.5 Code snippet showing Intent handling in Dialer app

Precondition $PR_{5.1}$: Privileged API call. *While performing an action requested by an Intent message, the app executes a privileged API without user intervention.*

Using the example of Figure 5.1 and Figure 5.4, this corresponds to an app that, after receiving an Intent from an attacker app, for example, formats the device by invoking the privileged API `DevicePolicyManager.wipeData(0)`, which requires the `BIND_DEVICE_ADMIN` permission or `sendUssdRequest()` that requires `CALL_PHONE` permission.

This is a case of permission re-delegation, as described by Felt et al. [33], because an app performs a privileged action on behalf of another app that lacks the required permission.

However, as also acknowledged by Felt et al., permission re-delegations are not always vulnerabilities; they can also be legitimate cases. Permission re-delegation is legitimate when it is available by the developer's intention. In fact, in Android, inter-app communication is a cornerstone feature for app integration that involves permission re-delegation. In our running example, initiating a phone call (`makePhoneCall()`) when requested by other app is an intended behavior of the Dialer app.

An accurate vulnerability detection approach should go beyond the mere detection of permission re-delegation and it should distinguish between legitimate permission re-delegations and permission re-delegation vulnerabilities.

Hence, going beyond Felt et al.'s threat model and the detection technique proposed in Chapter 4, in this chapter we pose an additional precondition to distinguish these two cases. To consider a permission re-delegation behavior as legitimate (as intended by the developer), it should be *similar* to what can be observed on many *similar* apps. Conversely, to consider a permission re-delegation behavior as vulnerable, it should represent an *anomaly*, i.e., something uncommon among *similar* apps. Thus, the following precondition is defined:

> ***Precondition $PR_{5.2}$: Anomalous permission re-delegation.*** *It is uncommon for other similar apps to execute that privileged API upon receiving an Intent message.*

We consider an app that satisfies both of the above preconditions as vulnerable to permission re-delegation attacks. A defect that satisfies both of these two preconditions represents a security issue: an Intent message makes an app perform a privileged action on behalf of the attacker app, and moreover, this privileged action is anomalous according to similar apps. In this way, an attacker app manages to control the execution of a privileged action without having the corresponding permission. We call this an anomalous delegation. In the following sections, we propose and assess an automated approach to detect such apps containing permission re-delegation vulnerabilities.

5.3 Overview of the Approach

The *PREV* framework is a fully-automated approach for detecting permission re-delegation vulnerabilities in Android apps. It takes as input the Android package kit (apk for short) file and the app description. As output, it generates a vulnerability report that states if the app is vulnerable or not and provides test execution scenarios with proof-of-concept attacks so as to document the security issues and help the developer in fixing the vulnerabilities or security auditors vetting third party apps.

As shown in Figure 5.6, the proposed approach consists of three major steps:

1. **Model inference**: this step takes a large *training set* of safe apps as input and produces permission re-delegation models as output. It contains three sub-steps:

   (a) The first sub-step applies topic modeling and clustering techniques to group those safe apps into clusters based on their similarities in terms of functional descriptions.

(b) In the second sub-step, for each app in each cluster, static analysis is used to generate the call graph and identify the privileged APIs that can be reached from public entry-points. This provides the permission re-delegation behaviors, i.e., the privileged operations that may be performed by the apps upon receiving incoming requests via public entry points.

(c) In the third sub-step, among the reachable privileged APIs of the apps in each cluster, we determine the common APIs and the uncommon ones. Based on this information, we learn the permission re-delegation model for each cluster, which characterizes the permission re-delegation behaviors of the safe apps in the cluster.

This step is performed only once before testing a given set of new apps; however, the models may need to be incrementally updated at times, for example, when new versions of safe apps become available, apps are removed or new apps are added.

2. **Outlier detection**: this step takes the clusters and the associated permission re-delegation models obtained in the first step and the app under test (AUT) as input. It reports anomalies as output. It contains three sub-steps:

   (a) First, it classifies the AUT into one of the clusters by using the same topic modeling technique used in the previous step and a classification technique.

   (b) Then, it proceeds to the second sub-step which applies the same API reachability analysis used in the previous step and extracts the reachable, privileged APIs in the AUT. If there is no reachable, privileged API, the procedure terminates reporting that the AUT is not vulnerable.

   (c) The third sub-step applies a classification method to identify the anomalies, which are reachable privileged APIs that are anomalous according to the permission re-delegation model. The AUT is flagged as an outlier; the anomalies are reported as *candidate* permission re-delegation vulnerabilities. If the AUT does not contain any anomaly, the procedure terminates.

3. **Test case generation**: this step takes the outlier AUT, the list of candidate vulnerabilities, and the call graph produced in the previous step as input. It produces proof-of-concept attacks as output. It contains two sub-steps:

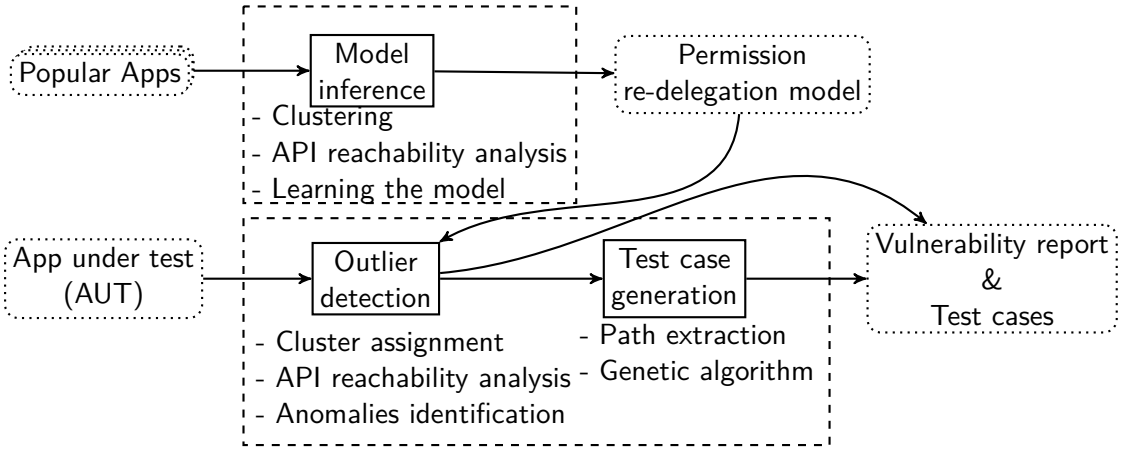


Fig. 5.6 Architecture of *PREV* framework

(a) It applies static analysis to extract *target paths* from the call graph — paths from public entry points to the calls to anomalous privileged APIs corresponding to candidate vulnerabilities.

(b) Next, it applies genetic algorithm-based technique to generate test cases that exercise the target paths. This confirms that the AUT is indeed vulnerable and exploitable. It generates a detail vulnerability report containing the anomalous privileged APIs used and the exploited target paths.

These steps are described in detail in the next sections.

5.4 Model Inference

5.4.1 Clustering

In the first step of our approach, we cluster apps that can be considered benign and non-vulnerable (safe apps) based on the similarity of their app descriptions. The intuition behind is that safe apps that are similar in terms of their descriptions should exhibit common permission re-delegation behaviors, which can be considered as legitimate.

For example, it might be common for communication-related apps to send SMS messages. However, it might be very uncommon for *safe* communication-related apps to send SMS messages *when servicing requests* coming from other apps (without involving user interaction). Essentially, while this feature is largely used internally, it is rarely exposed as a service to other apps. Thus, we can establish that sending SMS on behalf of the requesting app is not a common behavior for communication-related apps.

Whenever a new communication-related app is found to exhibit such behavior, it can be classified as an outlier.

The "safe" apps that we use are those apps that (i) come from official app store (therefore, they are scrutinized and checked by the store maintainer); and (ii) are very popular (as such, their quality is acknowledged by a large group of users). We chose Google Play as the official app store. At the time we crawled the Google Play store, it provided 30 different app categories. From each category, we downloaded, on average, the top 500 apps together with their descriptions. We then discarded apps with non-English description and those with short descriptions (less than 10 words). We are then left with 11,796 apps for clusters preparation.

The fact that top apps are suggested and endorsed by the official store makes us assume that the apps are of high quality and do not contain many security problems. However, it is important to note that our approach does not absolutely assume that all the "safe" apps which are used for learning the model are completely benign and non-vulnerable. In fact, our model is robust with respect to the inclusion of a small number of malicious or vulnerable apps in the training set, because we classify a permission re-delegation behavior as vulnerable when it deviates from the cluster norm. Therefore, as long as the majority of the apps exhibit legitimate permission re-delegation behaviors, the cluster norm will only reflect those legitimate behaviors (see Section 5.5). On the other hand, our approach does rely on the *majority* of them being truly benign and non-vulnerable. In our empirical evaluation, we will quantify how much majority is required for our approach to work in practice (see Section 5.7).

Our clustering step is inspired by the approach proposed by Gorla et al. [42], with some differences in topic classification and clustering algorithm used. Specifically, we additionally apply a NLP technique called *lemmatization* for better topic classification and we use a probability-based clustering algorithm based on Expectation Maximization (EM) algorithm and cross validation method for clustering so that the number of clusters does not need to be defined a priori. This step takes safe apps as input and produces clusters of safe apps as output. It consists of three sub-steps: 1) App descriptions preprocessing; 2) Topics discovery; and 3) Apps clustering.

App descriptions preprocessing. Our approach applies filtering, lemmatization and stemming (standard NLP techniques) to preprocess the app descriptions. The process is summarized with an example in Figure 5.7. First, it filters out non-English descriptions using Google's Compact Language Detector[37], because having one single language is necessary for clustering similar descriptions.

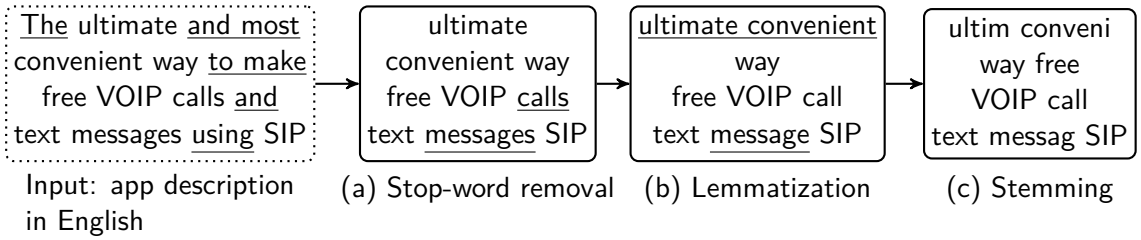


Fig. 5.7 App description processing (underline indicates what will be affected in the next step).

Second, the approach filters *stopwords* that do not contribute to topic discovery (Figure 5.7a) , such as "a", "after", "is", "in", "as", "very", etc[3]. Third, it applies lemmatization technique (Figure 5.7b) using the Stanford CoreNLP lemmatizer [88] to abstract the words having similar meanings in the descriptions so that they can be analyzed as a single item. For instance, the words "car", "truck", "motorcycle" appearing in the descriptions can be lemmatized as "vehicle". Last, it applies stemming (Figure 5.7c) technique [73] to transform the different forms of a word such as "travel", "traveling", "travels", and "traveler" into a common base form such as "travel".

Topics discovery. After the original app descriptions are preprocessed, the approach applies a topic modeling technique called Latent Dirichlet Allocation (LDA) [14] to discover the topics in the descriptions. LDA is a generative statistical model that represents a collection of text as a mixture of topics with certain probabilities, where each word appearing in the text is attributable to one of the topics. For instance, given a preprocessed app description "travel Italy group tour include dinner lunch pizza pasta restaurant", LDA generates the following topics with probabilities[4]: "Travel (20%)", "Food (37%)", "Restaurant (30%)", "Italy (13%)". Similar to Gorla et al. [42], we also use the Mallet framework [67] to perform this step. The framework allows us to choose the number of topics to be identified by LDA. Motivated by the number of Google Play categories, we chose 30.

Apps clustering. After the topics are discovered, a probability-based clustering algorithm described in [92] and implemented in the *Weka* tool [44], is used to group together apps based on common topics. This algorithm applies expectation maximization algorithm [27] and cross validation method. It is as follows:

1. The number of clusters is set to 1.

[3]see the list of common English stopwords at www.ranks.nl/stopwords

[4]To simplify the example, topics are represented by meaningful labels, which is not available in LDA.

2. The dataset is split randomly into 10 folds.

3. Expectation maximization is performed 10 times (in an attempt to escape local maximum).

4. The log-likelihood is averaged over all 10 results (log-likelihood is a measure of the "goodness" of the clustering).

5. If the log-likelihood has increased, increase the number of clusters by one and continues at step 2.

Expectation maximization is performed as follows: it starts with an initial guess of the cluster parameters (e.g., means and standard deviations of the clusters). It computes the probabilities for assignments of each instance to a cluster using the current parameters (expectation step). Then, using these cluster probabilities, it re-estimates the parameters (maximization step), and repeat the two steps again until the cluster parameters and cluster assignments stabilize.

The advantage of using this clustering algorithm is that it estimates the number of clusters in addition to providing the clusters themselves. Hence, we do not need to predefine what should be the number of clusters.

Running the clustering algorithm resulted in 30 clusters of similar sizes, each cluster containing between 3% and 4% of total apps. We manually sampled a few apps from a few clusters and verified that apps from the same clusters are indeed similar in terms of their functional descriptions.

We did not consider using Google Play categories as clusters because, while an app belongs to one Google Play category, an app's functional description may in fact incorporate multiple topics at once, which is a much richer information for clustering. Prior results [1, 42] reported that clustering by common topics produces more cohesive clusters than clustering by Google Play categories.

5.4.2 API Reachability Analysis

This step takes an app as input and produces a list of privileged APIs reachable from public entry points as output. A public entry point is an interface through which other apps, including malicious ones, can request an action via IPC. Privileged APIs are those Android APIs that require special permissions. A privileged API reachable from public entry point is a path in the call graph of the app that originates from a public entry point and that leads to a call to a privileged API.

For each app, we use static analysis to first identify the public entry points (sources) and the privileged APIs (sinks) in the app. Then, call graphs are generated and candidate vulnerable paths — paths from sources to sinks — are identified from the call graphs. To identify these APIs, we carry out the following tasks:

Public entry points identification. Public entry points are defined by *intent-filters* or the *exported* boolean attribute associated to components in the app *manifest* file, as described in Section 5.1. We model *Activities*, *Broadcast Receivers* and *Services* as possible public interfaces and their corresponding lifecycle entry methods (e.g., `onCreate()` for Activities and `onReceive()` for Broadcast Receivers) as entry points. Dynamically registered broadcast receivers are not part of our model. The design pattern usually adopted is to register dynamic receivers for broadcasts that we are interested in while our app is running and unregister them when the app is closed (e.g., in onStop() or onDestroy()). In other words, if the app is not running, any Intent broadcast to the dynamic broadcast receiver would not be caught by the app as the broadcast action (or the broadcast receiver in general) is not yet registered. Since the attack window for a dynamically registered broadcast receivers is limited to the time the app is running and the receiver is registered, we decided to inlcude only broadcast receivers that are statically registred.

The sample manifest in Figure 5.1 defines a single public interface — DialerActivity, because it defines the tag `<intent-filter>` without specifying the exported attribute (if a component specifies an intent-filter, by default the *exported* attribute is set to `true`). Our approach parses the manifest file using XOM[5], an open source library to parse XML files. Intent-filters are extracted using XPath queries.

Privileged APIs identification. The list of privileged APIs is predefined in our configuration, which is provided in the literature [10]. To identify uses of these APIs in an app, we use Soot[83] to convert the Dalvik bytecode of an app into an intermediate representation called Jimple. The Jimple code is traversed to identify *invoke* statements to those APIs that match our predefined list.

Reachable privileged APIs identification. We use *FlowDroid* [8], which extends Soot, to generate the call graph of the app. We then run a reachability analysis algorithm [78] on the call graph, to identify the privileged APIs that are reachable from public entry points.

[5]http://www.xom.nu

5.4.3 Learning the Model

Once the safe apps are clustered and the permission re-delegation behaviors are identified (reachable, privileged APIs), we need to learn the permission re-delegation model of each cluster.

At the end of this step, each cluster will have two thresholds, namely, t_{comApi} and $t_{outlier}$ (discussed below) representing the permission re-delegation model of the cluster. When analyzing an AUT in a given cluster,

- $t_{outlier}$ will be used to see if the the app is an outlier compared to the similar apps in the cluster.
- If the AUT is indeed an outlier, t_{comApi} will be used to identify which API is actually rarely used.

The permission re-delegation model characterizes the permission re-delegation behaviors of the apps in the cluster, i.e., which privileged APIs are (and are not) commonly called when servicing action requests. Information about reachable APIs is stored as the matrix M, as shown in Figure 5.8, with apps as rows and privileged APIs as columns. A cell in M is assigned to value 1 when the app in the row exposes the privileged API in the column when servicing action requests; otherwise it is assigned to value 0.

	openConnection()	connect()	sendTextMessage()	setWifiEnabled()
TrainingApp_1	1	1	1	0
TrainingApp_2	1	1	0	0
TrainingApp_3	1	1	0	0
TrainingApp_4	1	0	0	1

(a)

$\widetilde{m}$	1	0.75	0.25	0.25

(b)

Fig. 5.8 (a) Matrix M that stores information about API usage for each app in a given cluster (1=the app exposes the API, 0=otherwise), and (b) frequency vector $\widetilde{m}$ for M

A column with many cells set to 1 represents a reachable API that is commonly used by many safe apps when servicing action requests; so it can be considered as a *legitimate permission re-delegation behavior*. Conversely, a column with many cells set

to 0 represents a reachable API that is uncommon; so it should be considered as an *anomalous behavior*.

The notion of common/uncommon reachable APIs is captured by the frequency vector $\widetilde{m}$ that reflects the mean of the columns in M. Each element j of $\widetilde{m}$ is the mean of the j-th column of matrix M:

$$\widetilde{m}_j = \frac{1}{n}\sum_{i=1}^{n} M_{i,j}$$

The lengths of frequency vectors vary across clusters. On average, $\widetilde{m}$ has 218 elements.

For example, regarding the matrix shown in Figure 5.8, we have: $\widetilde{m} = [1, 0.75, 0.25, 0.25]$. The first and second elements of $\widetilde{m}$ corresponding to the APIs `openConnection()` and `connect()` have the value 1 or the value close to 1 since the APIs are frequently used while the third and the fourth elements corresponding to the APIs `sendTextMessage()` and `setWifiEnabled()` are close to the value 0 because the APIs are uncommon.

We compute a threshold called t_{comApi} to define what is common (and what is not). It is computed as the median of the values in the frequency vector, $t_{comApi} = median(\widetilde{m})$. APIs whose frequency is greater than t_{comApi} are considered as common and vice versa.

In our example, $t_{comApi} = 0.5$. The frequency of `openConnection()` is 1 and of `connect()` is 0.75; so the use of these APIs when servicing action requests is common and thus, considered as legitimate. On the other hand, the frequency of both `sendTextMessage()` and `SetWifiEnabled()` is 0.25, which is less than t_{comApi}; so they are considered as APIs uncommonly subject to permission re-delegation.

To simplify the approach, we could have set $t_{comApi} = 0$. In this way, we would identify APIs that are *never* used when servicing action requests in the given cluster as anomalous. However, for our approach to be robust with the inclusion of a few non-safe apps in the training set, we need a threshold larger than zero. We therefore consider those APIs that are *rarely* used when servicing action requests as uncommon by computing the threshold as explained above. In practice, even if t_{comApi} is not zero, it should still be close to zero.

Next, we compute the dispersion around the frequency vector $\widetilde{m}$ to understand how much an app should be far from this vector to be considered as an outlier. As suggested by literature [48], dispersion is evaluated with respect to the Euclidean distance between an app app_i and $\widetilde{m}$ with the following equation:

$$d(app_i, \widetilde{m}) = \sqrt{\sum_{j=1}^{n} (\widetilde{m}[j] - M[i,j])^2}$$

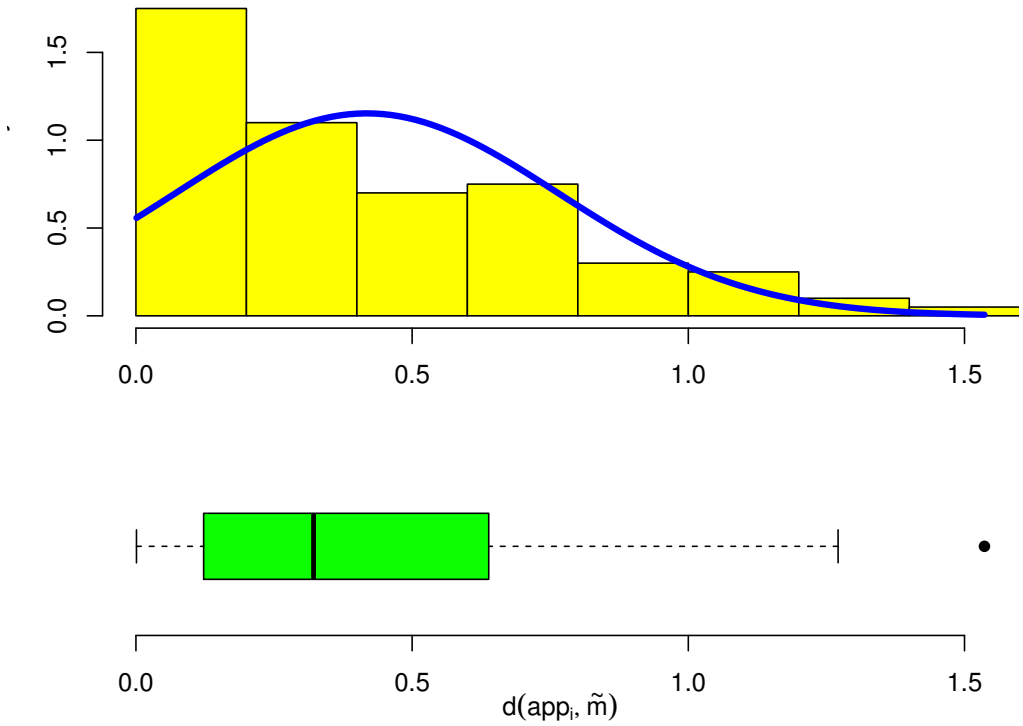


Fig. 5.9 Dispersion of the distance of apps from the frequency vector $\widetilde{m}$.

Figure 5.9 shows the dispersion of the distance. The upper part shows the histogram of the dispersion and the interpolating Gaussian curve. The lower part shows the boxplot. As we can see, in this example, apps on average have an euclidean distance of 0.6 from $\widetilde{m}$, with very few cases with a distance larger than 2.5. The app with distance larger than 3 is represented by the boxplot as an outlier dot.

We resort to the boxplot approach proposed by Laurikkala et al. [54] to detect outliers. The threshold called $t_{outlier}$ is computed in the same way as drawing outlier dots in boxplots:

$$t_{outlier} = Q_3 + step$$

$$step = 1.5(Q_3 - Q_1)$$

First we compute the difference between the upper quartile (75th percentile, $Q_3 = 0.64$ in the example) and the lower quartile (25th percentile, $Q_1 = 0.12$ in the example). In the example of Figure 5.9, this difference is 0.52. The *step* is computed by multiplying this difference by 1.5, i.e. in the example is $step = 0.78$. Eventually, $t_{outlier}$ is computed as the sum of the upper quartile Q_3 and the step. So, the threshold for the example is $t_{outlier} = 1.42$.

Any app with the distance from the frequency vector $\widetilde{m}$ larger than $t_{outlier}$ is considered an outlier. For instance, the app with $distance = 1.5$ is an outlier in Figure 5.9, because $1.5 > t_{outlier} = 1.42$, so it is represented as a dot. The frequency vector $\widetilde{m}$ and the thresholds t_{comApi} and $t_{outlier}$ represent the permission re-delegation model of this cluster.

As different clusters might exhibit different permission re-delegation models, the above process is performed for each cluster and is performed only once during this model inference step.

5.5 Outlier Detection

The second step of our approach takes as inputs the clusters and the permission re-delegation models obtained in the previous step, and the AUT. It then reports whether the AUT contains anomalous permission re-delegation behaviors. It contains three sub-steps: cluster assignment, API reachability analysis, and anomalies identification, which are explained in the following subsections.

5.5.1 Cluster Assignment

First of all we need to determine which permission re-delegation model to use among those available to compare the AUT against similar apps. That is, among the clusters we generated in Section 5.4, we need to identify the cluster the AUT belongs to. To achieve this objective, the description of the AUT is subject to the same sub-steps — app descriptions preprocessing and topics discovery — discussed in Section 5.4.1.

When topics and topic probabilities are computed, we use a simple, efficient classification algorithm called Naive Bayes (available in Weka [44]) to learn a classification model and assign the right cluster. The classifier is trained on the same "safe" 11,796 training apps we used for inferring the permission re-delegation models, using the topic probabilities of the apps as *features* and their clusters as *labels*. This classification model is then applied to the AUT, to identify the cluster with the most similar topic probabilities.

Figure 5.10 shows an example of the classification procedure. Figure 5.10(a) shows the training data, each line representing a different "safe" training app from the official store. There is a column for each topic. The value in each cell corresponds to the probability of the topic in the column given the description of the app in the row. For instance, the description of $TrainingApp_1$ is assigned to the topic "communication"

with probability 0.34, the topic "health&fitness" with probability 0.09 and "games" with probability 0.52. The last column reports the cluster number assigned to this app.

Figure 5.10(b) shows the topic probabilities for the description of the AUT and the missing cluster label. Later, the Naive Bayes classifier learnt on the training data shown in Figure 5.10(a) is used to label the AUT with the cluster whose member apps have the most similar topic probabilities with respect to the AUT.

App	*Communication*	*Health&Fitness*	...	*Games*	Cluster
$TrainingApp_1$	0.34	0.09	...	0.52	cluster20
$TrainingApp_2$	0.64	0.17	...	0.13	cluster10
$TrainingApp_3$	0.18	0.60	...	0.06	cluster26
$TrainingApp_4$	0.04	0.11	...	0.48	cluster6
...	...	...	...	...	...

(a)

AUT	0.12	0.07	...	0.78	?

(b)

Fig. 5.10 Classification model used for cluster assignment. (a) Topic probabilities with cluster labels, and (b) Topic probabilities for the AUT and missing cluster label

Note that for cluster assignment of the AUT, we do not use the clustering algorithm used in Section 5.4.1 to cluster the *safe apps + AUT* because classification would be more efficient since the group labels are already available after one-time clustering of safe apps. Given the topic probabilities of the AUT, we simply need to classify the group it belongs to, based on the existing group labels and their associated topic probabilities.

Regarding classification, we also evaluated more sophisticated classification algorithms such as Logistic Regression and Random Forest; but since the results were similar, we opted to use a simple classifier, which is Naive Bayes.

5.5.2 API Reachability Analysis

Similar to Section 5.4.2, API reachability analysis is performed on the call graph in order to identify privileged APIs that are reachable from public entry point(s). If no public entry point is found or no privileged API is reachable from public entry points, our analysis terminates here and it reports that there is no permission re-delegation

vulnerability in this AUT (because permission re-delegation vulnerability arises only when an AUT executes a reachable privileged API).

5.5.3 Anomalies Identification

	openConnection()	connect()	sendTextMessage()	setWifiEnabled()	d
$x_{aut,a}$	1	1	0	0	0.43
$x_{aut,b}$	0	0	1	1	1.48
$\widetilde{m}$	1	0.75	0.25	0.25	

Fig. 5.11 Anomalies identification by comparing the reachable privileged APIs of the apps against those in the frequency vector.

We first generate a vector x_{aut} storing the information of reachable privileged APIs in the AUT. Two examples of this vector are shown in Figure 5.11, first and second line, respectively for two apps AUT $_a$ and AUT $_b$. That is, the $i-$th element of a vector is set to 1 if API$_i$ is reachable, the element is 0 otherwise. For instance, the first and second elements of $x_{aut,a}$ are set to 1 because calls to APIs `openConnection` and `connect` are reachable from public entry points in the code of AUT $_a$. Similarly, the third and fourth elements of $x_{aut,b}$ are set to 1 because calls to APIs `sendTextMessage` and `setWifiEnabled` are reachable in the app code. This corresponds to computing new rows of the matrix M as discussed in Section 5.4.3 (see Figure 5.8).

To evaluate how different AUT$_a$ is from the cluster norm, we compute the Euclidean distance d_a between $x_{aut,a}$ and the frequency vector $\widetilde{m}$ (Figure 5.11). It is computed as $d_a = d(\widetilde{m}, x_{aut,a}) = 0.43$. We then compare d_a with the dispersion of permission re-delegation behaviors observed in the apps of the cluster (Figure 5.9). That is, we compare $d_a = 0.43$ and the threshold $t_{outlier} = 1.42$. Since $d_a < t_{outlier}$, it is concluded that the permission re-delegation behavior of AUT$_a$ is similar to those behaviors observed in the cluster and the AUT$_a$ is flagged as normal.

Likewise, for AUT$_b$, we can compute $d_b = d(\widetilde{m}, x_{aut,b}) = 1.48$. Since $d_b > t_{outlier}$, it is concluded that the permission re-delegation behavior of AUT$_b$ is substantially different from those behaviors observed in the cluster, which is a case of *anomalous* permission re-delegation, flagging the AUT$_b$ as an outlier.

When the AUT is flagged as an outlier (as in the case of AUT $_b$), there could be two cases of anomaly: 1) the AUT *does not* expose an API that *is* commonly exposed in the cluster; or 2) the AUT *does* expose an API that *is not* commonly exposed in the cluster. Clearly, we are only interested in the second case. An outlier app might

expose several privileged APIs, and the anomaly could be limited to a subset of them. Therefore, we still need to identify which privileged APIs are the anomalous ones. An API i is not commonly used for permission re-delegation in the cluster when its frequency is below the threshold t_{comApi}, i.e., $\widetilde{m}[i] \leq t_{comApi}$.

Therefore, the conditions for detecting anomalous permission re-delegation in the AUT with respect to API$_i$ are:

1. $d(\widetilde{m}, x_{aut}) > t_{outlier}$: It means that the AUT shows a permission re-delegation profile that is substantially different than the permission re-delegation profile observed on apps with similar features. So the AUT is flagged as an outlier. More conditions are required to determine what is the problematic API.

2. $\widetilde{m}[i] \leq t_{comApi}$: It means that the API$_i$ is a privileged API that is not commonly executed by the apps in this cluster when servicing action requests.

3. $x_{aut}[i] = 1$: It means that the AUT executes API$_i$ when servicing action requests coming from other apps (public entry points). In other words, the AUT exposes this privileged feature as a service to other (potentially malicious) apps.

In our running example, AUT$_b$ is an outlier because $d_b > t_{outlier}$. The privileged APIs `sendTextMessage` and `setWifiEnabled` are not commonly executed in its cluster because $t_{comApi} = 0.5$ and $\widetilde{m}[3] = 0.25$ and $\widetilde{m}[4] = 0.25$ (Figure 5.11). Therefore, the `sendTextMessage` and `setWifiEnabled` APIs satisfy the second condition. And these two APIs are exposed by our outlier App AUT$_b$. As shown in Figure 5.11, $x_{aut,b}[3] = x_{aut,b}[4] = 1$. Therefore, they also satisfy the third condition and are reported as anomalous privileged APIs.

Note that it is possible that vectors X_{aut} and $\widetilde{m}$ have different lengths. X_{aut} would have shorter length than $\widetilde{m}$ when the permission delegation model has APIs not observed in the AUT. In this case, we extend X_{aut} to $\widetilde{m}$'s length by adding zeros in the positions that correspond to the missing APIs. And we apply the same technique above to flag the anomalous APIs. On the other hand, when an AUT uses APIs that are never observed in the model, we flag it as an outlier and report those APIs as anomalous. We also use the above conditions 2 and 3 to flag the privileged APIs observed in the AUT but rarely observed in the model as anomalous.

5.6 Test Case Generation

The last step of our approach takes the outlier AUT and the list of anomalous privileged APIs as input. The objective is to generate security test cases, in the form of action

requests, that execute those anomalous privileged APIs. Such test cases represent proof-of-concept attacks for permission re-delegation vulnerabilities — executable scenarios that demonstrate the presence of security defects and that document them. Considering the fact that our main objective is *precision*, this also helps us reduce false positives that would have resulted due to the conservative static analysis.

Test case generation contains two sub-steps: path extraction and genetic algorithm, which are explained in the following subsections.

5.6.1 Path Extraction

For each anomalous privileged API, the call graph of the AUT is analyzed to identify the paths from public entry points to the calls of that privileged API. Let j be a call of the anomalous privileged API. The call graph is then traversed backward in depth-first search manner starting from node j until a public entry point node is reached. During the visits, each node is marked as visited so that loops in the graph are iterated at most once. As a result, regarding each API, we obtain a list of paths.

We then filter those paths that involve UI events. The inclusion of UI events in a path indicates that there is a user intervention (acknowledgment) before the privileged action is taken; hence it violates our first precondition for permission re-delegation vulnerability (Section 5.2.2). To identify UI events, we predefine a list of UI-related callback functions such as `onClick(), onTouch()` and Android Material Design Library functions (UI-related functions). Our tool detects paths that include a call to a function from this list and discards them. The remaining paths (denoted as *target paths*) are subject to testing next.

5.6.2 Genetic Algorithm

This sub-step generates security test inputs that execute the targets. The security test inputs we aim to generate are in the form of intent (action request) message, which is serviced by the AUT. Our goal is to generate at least one intent message that exercises a given target path. For any anomalous privileged API that we identified above, if there is at least one target path that has been exercised, our tool reports the corresponding AUT as vulnerable. We encode the intent message generation problem as an optimization problem, to be solved by a genetic algorithm. The genetic algorithm searches for an optimal solution (serviced intent message) by gradually evolving an initial population of random individuals through generations. Individuals nearer to the final solution are rewarded with a higher probability of transmitting their genes

to next generations. Fitnesses of solutions are computed using a *fitness function* and fittest solutions are combined together with the hope of generating fitter ones, until the optimum solution is found.

Individuals (or solutions) are analogous to chromosomes in genetics. In the following, we will use the term 'chromosome' to refer to both individual and solution.

A chromosome is encoded as a JSON-like data structure, which contains a set of fields and their values. A chromosome contains all necessary information for generating a concrete intent message. Table 5.1 shows the possible fields[6] and their example values of a chromosome. Chromosomes are evolved through *crossover* and *mutation*.

Field	Description	Example
$action	a string representing the action to be performed	`SMS`, `CALL`, `VIEW`, `EDIT`, ...
$category	additional information about the action to perform	`DEFAULT`, `BROWSABLE`
$extra	a (possibly empty) set of key-value pairs (not specified in the manifest file)	`"wifi_state'"`→"1", `"volume"` →10
$scheme	a value that expresses the format of the next $data field	`http` or `tel`
$data	URI that references the data to be used; e.g., the file to opened, the number to dial or the contact to access. Depending on the $scheme, this field is composed of different subfields	http://se.fbk.eu:80/profile/ceccato tel:+39.0461.314.577
If $scheme is `http`, $data represents a URL with these subfields:		
$scheme	the prefix of the URI	`http`, `https`, `ftp`, `file`, `content`
$host	the string corresponding to host name	se.fbk.eu
$port	the (optional) number corresponding to the port to use	80
$path	The path part of a URI to locate the corresponding resource	people/profile/ceccato
$pathPattern	Regular expression that the path should match	/specialdirctory.*/
If $scheme is telephony-related, $data represents a number to call/text with these subfields:		
$scheme	the prefix of the URL	`tel`, `sms`, `smsto`, `voicemail`, `mms`
$uri	the number/code to dial or to text to	+39.0461.314.577

Table 5.1 Fields of an intent message.

The test case generation work-flow for a given target path is summarized in Figure 5.12. Firstly, the *static analysis* component analyzes the Apk files of the AUT to extract the possible fields and values of intent messages that may exercise the target path, which are to be used as *seeds* for generating chromosomes (explained in the following).

[6]None of the fields are compulsory in an intent message.

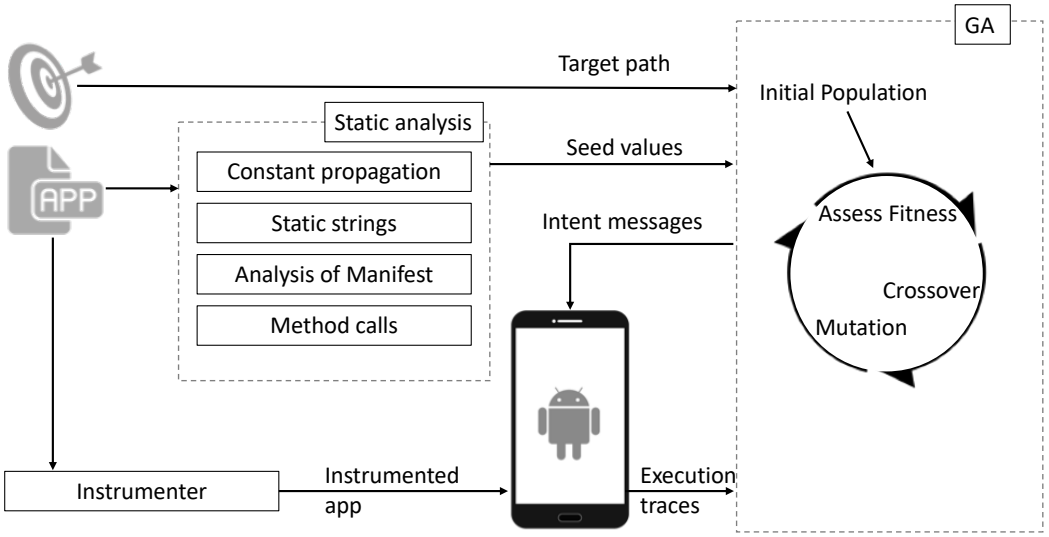


Fig. 5.12 Work-flow of the test case generation process.

Static analysis first identifies the app component that contains the target path. It then analyzes the intent-filter associated with that component in the *manifest file* and the *component code* to extract the possible fields of the intent messages that may be serviced by the component. For example, in our running example in Figure 5.1, by analyzing the manifest file, we can identify that an intent message requires *$action*, *$category*, and *$data* fields so as to be serviced by the *DialerActivity* component. Note that additional fields may be identified by analyzing the component code, since not all the fields are necessarily specified in the manifest file. From the component code, we also extract the *string constants* through simplified constant propagation and code scanning. Simplified constant propagation is applied to extract the values of string constants used as parameters in functions related to intents (e.g., `Intent.getAction().equals(ACTION)`) in the corresponding component code. The technique is simplified because, for scalability reasons, we do not track the propagation of string constants through string operations such as `substring`. Code scanning is applied to extract the string constants (such as static strings) from the component code.

The following explains how the seed values for these fields are extracted:

$action field: its seed values are extracted from the action values specified in the manifest file (e.g. `DIAL` in Figure 5.1). If no action is specified in the manifest file, its seed values are assigned with the string constants extracted from the corresponding

component code (explained above).

$category field: its seed values are extracted from the category values specified in the manifest file and also from the component code relevant to checking the category in intent messages (e.g. the value "Browsable" found in `Intent.hasCategory("Browsable")`). If no such value is available, similar to the *$action* field, the seed values are assigned with the string constants extracted from the corresponding component code (explained above).

$extra field: this field requires a list of key-value pairs. Since the manifest file does not specify extras, its seed values are extracted through static analysis of the component code. More specifically, static analysis is used to identify method calls that access *$extra* fields of intents and extract the keys (e.g. in `Intent.getIntExtra("id")`, `id` is extracted as a key). Simplified constant propagation is used if the key parameter in the method call is a constant. The data type of the value is identified based on method signature (e.g. integer for `getIntExtra`). Default values for those keys are sometimes available as parameters, e.g. in `Intent.getIntExtra("id", -1)`, -1 is a default value for `id`. If a default value is available, it is extracted as a seed value for the corresponding key. If no default value is found after static analysis of the component code, the seed values for a given key are assigned with the constants of the same data type extracted through scanning of the component code. The key is also annotated with its data type.

$scheme field: its seed value (typically only one value) is determined from the manifest file (e.g. `tel` in Figure 5.1). The value of this field defines the format of *$data* field (explained next). Currently, we support 15 different *$scheme*s that are grouped into two classes: for resources such as network and contacts (e.g., `"http", "file", "content"`) and for telephony (e.g., `"tel","sms","voicemail", "mms"`). Custom *$scheme*s (e.g., `"fb"` for Facebook) are also supported, when they are specified in the manifest file.

$data field: this field specifies data to be used to perform the requested task. It has sub-fields depending on the *$scheme* in use, such as *$host*, *$port*, *$path*, *$uri*. Similar to the above, the seed values of these sub-fields are also extracted through analysis of the manifest file and the component code. The *$data* field is generated when it is

specified in the manifest file.

$pathPattern field: this sub-field is usually specified in the manifest file as a regular expression (regex) consisting of the wildcards, asterisk (`*`), a period followed by an asterisk (`.*`). The Android framework uses PatternMatcher, a simple pattern matcher that is safe to use on untrusted data and does not provide full regex support. According to the documentation[7], an asterisk (`*`) matches a sequence of 0 to many occurrences of the immediately preceding character, while a period followed by an asterisk (`.*`) matches any sequence of 0 to many characters. The seed value for this field is, thus, generated as the shortest string accepted by the regex. For example, given a pathPattern `"/movies.*/"`, a string `"/movies/"` is generated.

For each of the field, we also include `NULL` value in its seed values. All the extracted fields and their seed values are then stored in a Database to be later used by the GA component to generate the chromosomes.

The *instrumenter* component in Figure 5.12 instruments the AUT bytecode (based on Soot) to insert hooks at method/API invocations to trace which methods and APIs are invoked at runtime. The instrumented app is then run (in our case in the Android emulator) to process the intent messages generated by the GA component. The execution traces are logged.

The following explains how the *genetic algorithm (GA)* component works:
Initialize population of random chromosomes. The GA generates a population of 150 chromosomes. For each chromosome, the algorithm starts with initializing all the possible fields identified above. For the *$action* and *$category* fields, their values are randomly selected with uniform probability from their seed values extracted above. Notice that a null value may also be selected.

For the *$extra* field, we need to generate keys and values. Keys are selected randomly with uniform distribution from the ones extracted above. The value for each selected key is picked from its seed values with 70% probability or generated randomly with 30% probability. The randomly generated value is of the same data type annotated at the key. To generate a value of string data type, we give a high probability of generating a random string of up to 10 characters, based on our experience. If it is of numeric data type, we give a high probability of generating a random value close to the default value if available (i.e., added or subtracted a small value from the default value).

[7]https://developer.android.com/guide/topics/manifest/data-element

A similar algorithm is used to generate values for other fields. For example, for *$host* field, a value is picked from its seed values with 70% probability or randomly generated with 30% probability.

The same process is repeated to generate a random number of chromosomes. For each generated chromosome, a corresponding security test case in the form of an intent message that can be executed in the Android emulator is generated.

Figure 5.13(a) shows a chromosome for the running example in Figure 5.1 with its *$action* and *$category* fields set to, respectively, `CALL` and `DEFAULT`. The field *$extra* contains the key `count` with the integer value `0`. The field *$scheme* is set to `tel` and the subsequent *$uri* field contains the phone number. Figure 5.13(b) shows another chromosome containing the same set of fields but with different values for some of them.

Figure 5.14 shows an ADB command that sends a concrete Intent message corresponding to a chromosome.

Assess fitness of chromosomes: this step computes the fitness of each chromosome. The objective of a security test case is to exercise the target path, from a public entry point to the anomalous privileged API. Based on the execution traces logged by the instrumented code (*Instrumenter* component), the GA component determines the actual path exercised by a given test case and then uses a fitness function to compute the fitness of its corresponding chromosome.

The *fitness function* we use is similar to the approach-level introduced in [90] for the Daimler Evolutionary Testing System. However, instead of evaluating how many nodes are executed to see how far we are from the target, we compute the percentage of call edge executed, i.e., the ratio of call edges executed by the test case and the total edges present in the target path.

Crossover: From the population of chromosomes, we use Binary Tournament algorithm to select two chromosomes based on their fitness values. The two chromosomes reaching the final of the tournament are removed from the population and subject to crossover. We pose the constraint of performing crossover only between chromosomes having the same *$scheme*. That is, if the two selected chromosomes have different *$scheme*s, they are put back into the population and the tournament is restarted.

This constraint is meant to combine only intents with compatible fields. Different schemes may imply different sub-fields of the subsequent *$data* fields. For instance, the `tel` scheme requires the *$data* field to contain only a phone number, while the `http` scheme requires the *$data* field to be composed of *$host*, *$port* and *$path* (see Table

```
INTENT:
 [
  {$action: "CALL"},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 0}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
   {$uri: "+39.0461.314.577"}
   ]
  }
 ]
```

(a) Chromosome A

```
INTENT:
 [
  {$action: ""},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 10}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
    {$uri: ""}
   ]
  }
 ]
```

(b) Chromosome B

```
INTENT:
 [
  {$action: "CALL"},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 0}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
    {$uri: ""}
   ]
  }
 ]
```

(c) Chromosome C

```
INTENT:
 [
  {$action: ""},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 10}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
   {$uri: "+39.0461.314.577"}
   ]
  }
 ]
```

(d) Chromosome D

Fig. 5.13 Examples of chromosomes

5.1). Intents with the same *$scheme* ensures that *$data* are composed of compatible sub-fields that, thus, can be exchanged.

We adopt a structured crossover operator that operates field-wise by crossing over fields of the same type. This is to preserve syntactic validity during evolution. When

```
adb shell am start
    -n com.example/.DialerActivity
    -a CALL
    -c DEFAULT
    -ei "count" 0
    -d tel://+39.0.461.314.577
```

Fig. 5.14 Example ADB command that sends an intent message.

two chromosomes A and B are selected to crossover, two new chromosomes (offspring) C and D are generated using the following steps:

1. chromosome A is cloned as chromosome C;
2. chromosome B is cloned as chromosome D;
3. one or more fields of chromosome C are randomly selected;
4. for those selected fields that are also present in chromosome D, their values are swapped.

To illustrate the crossover process, let us assume that *Chromosome A* and *Chromosome B* shown in Figure 5.13(a) and Figure 5.13(b), respectively, are selected for crossover. They have the same *$scheme*, i.e., *"tel"*; therefore, crossover is allowed. Firstly, *Chromosome C* and *Chromosome D* are cloned from *Chromosome A* and *Chromosome B*, respectively. Then, assuming that the field *$uri* is randomly selected, the *$uri* values of *Chromosome C* and *Chromosome D* are swapped, resulting in two new chromosomes as shown in Figure 5.13(c) and Figure 5.13(d).

The same process is repeated to select pairs of chromosomes from the population and crossover. This results in a new population of chromosomes, having roughly the same size as the original one (last remaining chromosomes with different *$scheme*s are discarded).

Mutation: Given a new chromosome generated through crossover, the values of *some of its fields* are mutated with a low probability of 30%, i.e., they have 70% probability of not being mutated. Depending on the type of the field, different mutation approaches are used to ensure that the generated intent messages are well-formed and accepted by the app.

- The values of the *$data*, and *$scheme* fields are mutated (with 30% probability); the mutation is performed by only swapping the original value with one of the seed values, selected with uniform probability.

- The value of the *$action*, *$category* and *$pathPattern* fields are not mutated.
- For the *$extra* field, the keys are not mutated. The values of the keys are mutated with 30% probability. The mutation is performed as follows: the original values of the keys are swapped with one of their seed values with 50% probability or mutated with 50% probability; if the original value is a string, it is mutated by deleting, inserting or replacing a character in the string with a random character; if the original value is a numeric, it is mutated by adding or subtracting an offset; small offsets are given high probability and the probability of large offsets decreases exponentially.
- For all other fields, the mutation process similar to the *$extra* field is applied. That is, the mutation is performed with 30% probability. For the mutation, the original values are swapped with one of their seed values with 50% probability or mutated with 50% probability.

Timeout: The stopping criteria of the GA is set as 500 generations.

Note that the tuning parameters — the population size, the mutation probabilities, the value selection probabilities, and the stopping criteria — we used above are decided based on our preliminary assessment of the test generation algorithm, which we ran on a set of randomly selected apps. We found that higher probabilities of mutating a chromosome (>30%) and lower probabilities of selecting a value from seeded ones for mutation (<50%) usually results in the loss of good solutions. On the other hand, the test generation was not very effective when we used much lower probabilities of mutating a chromosome (e.g., 10%) and much higher probabilities of selecting seeded values (e.g., 90%). Some of the fields in intent messages, such as *$action*, *$category* *$pathPattern*, and *$key* are not mutated at all because it would only result in ill-formed intent messages that would be rejected by the AUT. Overall, this ensures that the population is evolved towards better generations.

5.7 Evaluation

In this section we evaluate *PREV* and compare with state-of-the-art static analysis-based techniques that can be used to detect permission re-delegation vulnerabilities.[8]

[8]We did not compare with test generation-based approaches because we could not find an adequate or available tool that might be suitable for detecting permission re-delegation vulnerabilities.

Our goal is to detect the vulnerabilities as many as possible at an affordable cost. Therefore, our main evaluation criteria is precision and cost. In addition, we also investigate the recall and report the results.

More specifically, the following research questions are investigated:

- $RQ_{5.1}$ *(Precision)*: Is *PREV* precise at detecting permission re-delegation vulnerabilities in Android apps?
- $RQ_{5.2}$ *(Cost):* Is the cost (in terms of analysis time) of using our approach affordable in practice?
- $RQ_{5.3}$ *(Recall)*: Does *PREV* miss permission re-delegation vulnerabilities?
- $RQ_{5.4}$ *(Comparison):* Does *PREV* perform better than static analysis tools that can detect permission re-delegation vulnerabilities?
- $RQ_{5.5}$ *(Robustnesses):* Is *PREV* robust against the inclusion of anomalies in the training set?
- $RQ_{5.6}$ *(Threshold):* What is the impact of other threshold values on vulnerability detection accuracy of *PREV*?

Our evaluation was conducted on a machine equipped with an Intel Core i7 2.4 GHz processor, 16 GB ram, running Apple Mac OS X 10.11. Our tool is instrumented to log the analysis time.

5.7.1 Subject Apps

Our subject apps include a total of 1,258 real world apps that come from the official Google Play store. Since our approach works on compiled apps, the availability of source code is not a requirement. Nevertheless, 595 of our subject apps are open source projects, which offer us the possibility to inspect the source code, determine the correctness of vulnerability reports generated by the tools, and analyze the causes of vulnerabilities. The following explains our selection process of subject apps:

First, we obtained a list of app package names from the directory of AndroZoo[2], an app crawling research project that lists app names from many official and unofficial app repositories. We then randomly sampled the package names from this list. Additional app package names are also taken from a repository that collects open source Android apps, namely F-Droid[32]. We then checked that sampled apps from AndroZoo apps

and from F-Droid are available in the official app store, which ensure that our subject apps are real world apps.

We then filtered apps that are too popular (more than 1 million downloads), to increase the chance of selecting apps that are interesting for our experiment, i.e., apps with a good chance of containing vulnerabilities. Popular apps, distributed by well-reputed companies, are probably already subject to intensive security review.

To be able to apply our analysis, we additionally require that app description is in English, that contains at least 10 words so that natural language processing and topic discovery can be performed.

Eventually, there remained 1,258 apps — 663 closed source and 595 open source — from the official app store together with their descriptions. For open source apps, we acquired source code to conduct manual verification later.

5.7.2 Metrics

To answer our research questions, we report results in terms of these metrics:

- *True positives (TP):* Number of actual vulnerable apps correctly reported as vulnerable;
- *False positives (FP):* Number of vulnerable apps incorrectly reported as vulnerable (false alarms);
- *False negatives (FN):* Number of vulnerable apps that are missed (not reported by the tool);
- *Analysis time:* The time (measured in minutes) taken by the tool to analyze a subject app;

To answer $RQ_{5.1}$ and $RQ_{5.4}$, we quantify the precision (Pr) of the tool based on *true positives* and *false positives* (Pr=TP/(TP+FP)). More specifically, when a tool reports a vulnerability, when source code is available, we manually inspect the part associated with the reported vulnerability. When the source code is not available, we resort to the test case generated by the tool and observe the actual runtime behavior exercised by the test case. We then determine if the report is a true positive or a false positive. Note that since *FlowDroid*, *Covert*, and *IccTA* do not generate test cases, we experimented them only on the open source apps so that we can verify their vulnerability reports.

We answer $RQ_{5.2}$ by using the *analysis time* to quantify the cost of using the tool.

To answer $RQ_{5.3}$ and $RQ_{5.4}$, we measure the recall (also called the probability of detection, Pd) of the tool based on *true positives* and *false negatives* (Pd=TP/(TP+FN)).

However the challenge here is to establish the *false negatives*, we would need to thoroughly inspect the source code of the subject apps, and determine if they are vulnerable or *absolutely safe*. This would require an overwhelming effort. Therefore, instead of conducting a security review of all the subject apps to label them as safe/vulnerable, we conduct a *controlled experiment* in which we apply *two mutation operators* to inject security faults that reflect realistic permission re-delegation vulnerabilities into a set of randomly selected apps (see subsection 5.7.3). This provides us a benchmark for evaluating $RQ_{5.3}$, where all the apps in this benchmark are vulnerable by construction.

We answer $RQ_{5.4}$ by comparing the results of our tool, *PREV*, against three state-of-the-art tools in detecting permission re-delegation problems in Android apps, namely *FlowDroid*, *Covert*, and *IccTA*.

5.7.3 Controlled Experiment

In this sub-section, we explain how we conducted the controlled experiment to answer $RQ_{5.3}$. Based on the real-world permission re-delegation vulnerabilities reported by Porter et al. [33], we defined two mutation operators to inject permission re-delegation vulnerabilities. Mutation operators are μ_1: *Expose API* and μ_2: *Expose component*.

Mutation operator μ_1: Expose API. To provide its services, an app may be granted with special permissions to perform privileged operations (APIs). If these operations are security-sensitive, the app developer may limit these operations among its internal components only and may not intend to accept action requests of these operations from external apps.

Mutation μ_1 is used to inject a security defect by contradicting the developer's intention of limiting the exposure of a privileged API to action requests. The precondition to apply this mutation operator is:

- There is an exposed component that can be called by other apps to request an action from this app;
- The app contains a call to a privileged API;
- But this privileged API is not exposed to action requests (i.e., the call to this API is not reachable from public entry points).

If this precondition is met by a given app, we apply μ_1 by exposing the privileged API to action requests via the exposed component. Hence, the postcondition after applying μ_1 is:

- The privileged API is now exposed (i.e., reachable from at least one public entry point) and thus, the app is vulnerable to permission re-delegation attacks.

For example, in Figure 5.15, the left-hand side shows the original code of an app; *onCreate* is an exposed component; the method m contains a call to *sendSMS*, but m is not reachable from any public entry point. The right-hand side shows the mutated code, which exposes a privileged API by adding a call to the method m in the exposed component.

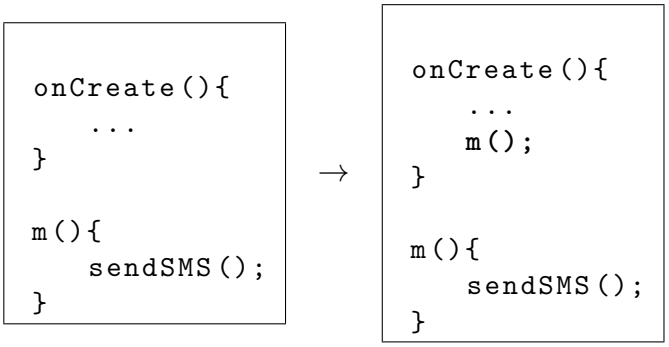


Fig. 5.15 Applying mutation operator μ_1: *Expose API*

Mutation operator μ_2: Expose component. Similarly to μ_1, μ_2 is used to inject a security defect by contradicting the developer intention of limiting the exposure of a privileged operation. However, in this case, the mutation is on the component visibility to other apps.

The precondition to apply this operator is:

- A component is not exposed; so it cannot be called by other apps;
- The component contains a call to a privileged API;
- But the call to this API is not reachable from public entry points.

If this precondition is met by a given app, we apply μ_2 by changing the visibility of the component in the *manifest* file of the app. Hence, the postcondition after applying μ_2 is:

- The component is now exposed; so the privileged API call is now reachable from at least one public entry point.

For example, in Figure 5.16, the left-hand side shows the original code; the method *onCreate* in the activity A is not exposed and it contains a call to a privileged API *sendSMS*, which is also not reachable from any public entry point. The right-hand side shows the mutated code, which specifies in the manifest file that activity A is publicly callable.

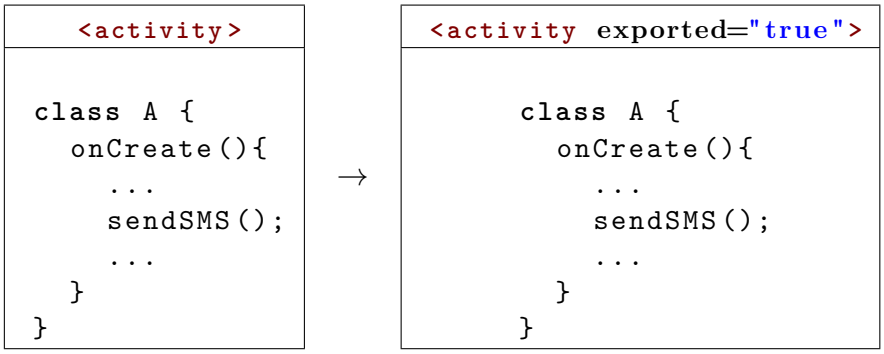


Fig. 5.16 Mutation operator μ_2: *Expose component*

5.7.4 $RQ_{5.1}$: Precision

We ran our tool on the 1,258 subject apps. Each app under test (AUT) was subject to outlier detection and test case generation steps shown in Figure 5.6.

Given the available clusters (see Section 5.4), we map each AUT to the cluster with most similar app descriptions. We then obtain the permission re-delegation model inferred on the corresponding cluster. Next, we ran API reachability analysis on the AUT, to identify those privileged APIs that are reachable from public entry points. Out of all the 1,258 apps, 401 apps contain reachable privileged APIs.

We then performed anomaly identification, which basically checks if the identified reachable privileged APIs are common or anomalous according to the permission re-delegation model. *PREV* detected that in 324 apps, the reachable privileged APIs are common according to the permission re-delegation model and therefore, they were classified as safe. The remaining 77 apps were classified as candidate vulnerable apps because the reachable APIs in those apps are anomalous. These candidate vulnerable apps were then subject to the test case generation phase of *PREV*, to automatically generate proof-of-concept attacks. *PREV* successfully generated attacks for 30 of these apps. (Note: we also used open source apps from those 77 apps to evaluate recall in Section 5.7.6.)

We then face the challenge of manually analyzing the apps for which a test case is generated, to label the analysis results as true positive (real vulnerability) or false

positive (false alarm). To classify a reported app as vulnerable, first we check that permission re-delegation has occurred, i.e., a test case makes the app execute a privileged API. Then we verify whether this case of permission re-delegation is a vulnerability. To limit subjectivity in verifying this second condition we adopt these guidelines (explained in more detail later on actual results):

- *Custom protocol*: the vulnerability can be triggered only with a particular message that follows an application-specific invocation protocol;
- *System intents*: the vulnerable component subscribed for system-generated events, but it fails to check whether the notified event is actually generated by the system;
- *Misuse of libraries*: the vulnerable app performs an insecure use of a library that deals with sensitive data;
- *App description*: a permission re-delegation causes the vulnerable app to perform a privileged task that is not explicitly specified as a feature in the app description.

Detailed results of this empirical validation are shown in Table 5.2, they are the closed source apps followed by the open-source apps, that are detected as vulnerable by *PREV*. The first column shows the app reported as vulnerable. The last two columns ('TP' and 'FP') indicate whether the reported vulnerable app is a true positive or a false positive, respectively, based on our manual inspections.

Analyzing the 1,258 subject apps, *PREV* reported 30 vulnerable apps — 23 closed source apps and 7 open-source apps.

Manual inspection on the reported vulnerabilities revealed that, for all of them, the test cases generated by *PREV* actually reached a privileged API. Moreover, all the cases represent permission re-delegation vulnerabilities according to our guidelines. In the following we discuss some of those cases in detail to highlight the reason for their classifications.

Custom protocol: app components may use custom invocation protocols such as private *actions* only known to the app (e.g., not specified in the intent-filter) or use specific values in custom *extra* parameters. By private action, we mean an action that is purely associated to the AUT, because (i) the action name is not mentioned in the app intent-filter, where the call protocol is exposed to other apps; and (ii) the action name has the same prefix as the app package name, e.g. the action `com.example.TEST_ACTION` for the app `com.example.TestApp`, so it is not from an included library of from the Android framework. Thus, so this action is likely meant only for internal use, only by components of the AUT.

Table 5.2 Vulnerability report of *PREV* on open source and closed source apps.

App	TP	FP
com.mendhak.gpslogger	✓	
com.seafile.seadroid2	✓	
org.ligi.ajsha	✓	
org.linphone	✓	
org.tigase.messenger.phone.pro	✓	
org.totschnig.myexpenses	✓	
org.ttrssreader	✓	
bestvalleygames.turningvalley	✓	
com.akgun.uknews	✓	
com.appportunity.androidpreviewer	✓	
com.appreka.mycoop	✓	
com.appsdv.smsmefitr	✓	
com.aurorasi.aurorasfa	✓	
com.bimandika.Congratulationsmalonepost	✓	
com.braingen.devanagarinotepad	✓	
com.dinosaur.dinosaur_vs_zombie	✓	
com.fmplural.radio	✓	
com.innogang.kollywoodNews	✓	
com.javirurro.games.spaceshipzigzag	✓	
com.josejoaquin.traductor	✓	
com.magmamobile.game.SpiderSolitaire2	✓	
com.netdania	✓	
com.npes87184.s2tdroid	✓	
com.rbsoftware.pfm.personalfinancemanager	✓	
com.reverbnation.artistapp.i739749	✓	
com.softdx.qrscanner	✓	
com.superfanu.bryantbulldogrewards	✓	
com.vent	✓	
lv.delfi.ru	✓	
piproduction.frankthejew	✓	

We assume that components which use such custom protocols are intended to be private, not visible to other external apps. Therefore, when there is a permission re-delegation scenario in which intent messages can invoke such components, we assume that this is a developer's mistake, i.e., a privileged feature that was menat to be used only interally, is re-delegated to other apps. Thus, we classify the case as a permission

re-delegation vulnerability. This guideline was applied to classify 8 vulnerable cases, for instance in *com.netdania* and *piproduction.frankthejew*.

System intents: Apps may subscribe for notification of system events via intent filters — events that only the Android platform can generate. For example, the app *com.superfanu.bryantbulldogrewards* is meant to be notified when the boot is complete, i.e., it has registered an intent-filter to receive an intent with *ACTION_BOOT_COMPLETED* action, that only the system can send. But the code of this app does not validate that this notification was actually sent by the system, because it does not check that the intent action is actually *ACTION_BOOT_COMPLETED*. Any intent sent to this component is blindly assumed to come from the system, and processed as such. Therefore, when the intent filter specifies a system action, but the app code does not validate intent data, we assume that it is a programming mistake and we classify the case as a permission re-delegation vulnerability. This guideline was applied to classify 3 vulnerable cases.

Misuse of libraries: Apps may be granted special permissions to use special libraries that deal with sensitive data; but they are expected to adhere to the security policies specified in the library documentations. Therefore, when an app uses a special library in a way that violates the library security policies, we assume that it is a programming mistake and classify the case as a permission re-delegation vulnerability. Several apps such as *com.appsdv.smsmefitr* and *com.aurorasi.aurorasfa* use the OneSignal and Google Analytics libraries, respectively, without enforcing permissions and the components using them process broadcasts without verifying that the intent, with the protected broadcast action `ACTION_MY_PACKAGE_REPLACED` (i.e., that can only be sent by the system), comes from the *PackageManager*. This guideline was applied to classify 4 vulnerable cases.

App description: If none of the previous considerations apply, we resort to the features described in the app descriptions to understand if permission re-delegation is intentional. From the description, one can find out about the primary features of an app (e.g., accessing the camera by a photography app). It might be the intention of the developer to expose these primary features to other apps (and let other apps request this app to take pictures on their behalves). When features that require privileged permission but not described in the description are exposed, we assume that it is not the developer's intention and classify the case as a permission re-delegation vulnerability. For instance the app *com.appportunity.androidpreviewer* is described as a gallery of apps, to help a developer keep track of her/his apps. However, the app exposes camera

features to other apps, without mentioning it in the description. This guideline was applied to classify 15 vulnerable cases.

It could be the case that multiple guidelines may be applied. For example, the app *com.bimandika.Congratulationsmalonepost* misuses a library without enforcing library component permission and it does not validate the sender for (presumed) system intents.

Based on these results, we can formulate the subsequent answer to $RQ_{5.1}$:

> *Analyzing 1,258 apps,* PREV *reported 30 vulnerable apps without any false alarm. The implication is that security analysts could use* PREV *to precisely identify vulnerabilities without any false alarm.*

5.7.5 $RQ_{5.2}$: Cost

We investigate the cost of *PREV* in terms of analysis time. First we discuss the cost of learning permission re-delegation models.

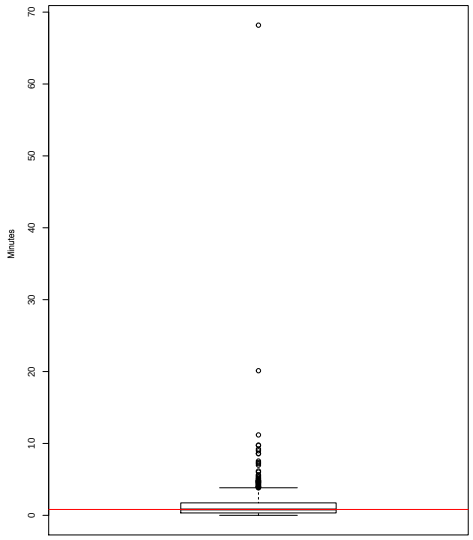
Before testing any given AUT, the permission re-delegation models were learnt by running our tool on 11,796 apps, as described in Section 5.4. *PREV* took 250 hours to learn the models from 11,796 training apps. The bottleneck was static analysis. Most of the time was spent on extracting reachable APIs. Natural language processing, clustering, and model inference was quite fast (a magnitude of minutes). Since Android is always evolving (e.g., change of permission mechanism), the models should be re-learnt whenever newer set (or versions) of reference apps becomes available. However, for testing a given set of apps under test, this step needs to be conducted only once. Moreover, as suggested by Harman et al. [45], static analysis can be done incrementally by analyzing a new app or a new version as soon as it is posted on the app store. It is not necessary to conduct static analysis on the whole training set. This would reduce the training time significantly less than 250 hours. Since this training step does not have any real-time requirement, we consider this training cost as affordable.

Next, we discuss the cost of analyzing a given AUT. Outlier detection was performed on all the 1,258 apps; test case generation occurred only on the 77 apps that were reported as potentially vulnerable by the outlier detection phase of *PREV*. In the following, we first discuss the time taken for outlier detection of 1,258 apps and then discuss the test case generation time for the 77 apps reported by outlier detection.

The time taken to perform outlier detection analysis is displayed in the boxplot in Figure 5.17. As shown in the boxplot, on average, it took less than one minute and a half to complete the analysis. Only a few apps took a longer time (represented as

outlier dots in the boxplot) and this was due to the use of complex libraries in the AUT.

Time taken to perform test case generation is shown in Figure 5.18. Test case generation took longer than outlier detection; on average it takes less than 25 minutes per app. This is due to the genetic algorithm exploring the search space when trying to find an executable scenario that represents proof of concept attack. However, this duration depends on the timeout setting in the experimental configuration.

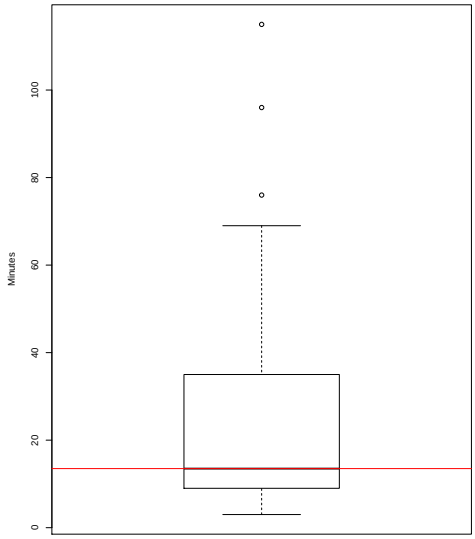


mean	median	sd
1.275191	0.8166667	2.335136

Fig. 5.17 Time (in minutes) spent during static analysis (above) and descriptive statistics (below)

Considering these results we can formulate the subsequent answer to $RQ_{5.2}$:

Outlier detection phase took 1.3 minutes on average. Test case generation phase took on average 25 minutes, but it is performed only for the apps detected as potentially vulnerable by outlier detection. It will take longer to use PREV *when the models need to be re-learnt. But this is typically an offline activity, with no real-time requirement. Therefore, in general it would only take a magnitude of minutes to determine if an app is vulnerable. Overall we consider that the cost of using* PREV *is affordable in practice.*



mean	median	sd
24.9	13.5	25.25886

Fig. 5.18 Time (in minutes) spent during test case generation (above) and descriptive statistics (below)

5.7.6 $RQ_{5.3}$: Recall

To answer $RQ_{5.3}$, we evaluate if *PREV* misses any vulnerabilities. Note, however, that the main objective of *PREV* is achieving high *precision* over existing approaches. We use two sets of *known vulnerable* apps. The first set of apps consists of 20 mutant apps which were subject to the mutation operators presented in Section 5.7.3. The second set of apps are 35 apps that were used earlier in Section 5.7.4.

Regarding the first set of apps, we started by a random sample of apps, consisting of 50 apps from the official store and 80 apps from open-source repository. We apply mutations to these apps to inject permission re-delegation vulnerabilities. Since an app might contain multiple components and multiple privileged APIs, the same operator might create several distinct mutated versions for the same app.

However, due to implementation limitations, our mutation operators sometimes fail in generating a working vulnerability in the following cases:

- The mutated app crashes;
- The privileged API call is in a path that is not realizable from public entry points, due to certain path conditions in place;

- A path involves a UI event, such as a click event;
- A mutated component only accepts intents sent by the system.

We manually checked the mutants and discarded such cases.

Finally, our first set of apps consists of 20 mutants, generated from 14 different original apps, vulnerable by construction — 11 from closed source apps and 9 from open-source apps. Each mutant contains one injected vulnerability; some mutants are generated from the same original app but injected with different vulnerabilities.

The results of *PREV* on this benchmark is shown in Table 5.3. The top part shows the results corresponding to the mutants of closed source apps and the bottom part shows the results corresponding to the mutants from open-source apps. The mutant name (first column) is a concatenation of the original app name, the mutation operator, and the unique-id of the component subject to mutation. As shown in Table 5.3, the same app *com.nextcloud.client* could be used to generate three distinct mutants. A tick-mark "✓" is present in the second column (TP) when *PREV* generated a test case for the corresponding mutant. Conversely an x-mark "✗" is reported in the third column (FN) when no test case was generated.

As shown in Table 5.3, among the mutated closed source apps, *PREV* detected 7 out of 11 vulnerable apps but missed 4. Among the mutated open-source apps, *PREV* detected 8 out of 9 vulnerable apps but missed 1 case, i.e., *com.nextcloud.client _10040299_exposed_13*.

PREV was not able to generate inputs for 5 apps in total. *PREV* failed in generating a test case for `lwcr46lion.lwp_exposed_360`, because a successful test case would have required a media URL (e.g., a URL pointing to .mp4 file) with an advertisement impression to display. When apps are expecting an intent *$data* field with a URI that meets some conditions, these conditions are usually specified in the intent-filter. As discussed in Section 5.6.2, the test case generation phase relies on intent-filters in order to seed the *$data* field. However, this component does not specify an intent-filter at all (only the attribute `exported` set to `true`). While the test case generation can seed other intent fields, such as `$action` and `$extra`, from the component's code even if they are not specified in the manifest file, the `$data` field is seeded either from the intent-filter or the component code *only* when the specification exists in the manifest file. As this component does not specify an intent-filter, our approach failed in generating the `$data` field that was essential to test this app. Similarly, the remaining 4 apps require inputs that could not be generated automatically and therefore, are missed.

Regarding the second set of vulnerable apps, we first looked at the 77 apps, used in Section 5.7.4, that are reported as candidate vulnerable apps by our static analysis

Table 5.3 Vulnerability report of *PREV* for mutated apps.

Mutant	TP	FN
com.colortime.mandala_exposed_99	✓	
com.compasskeyboards.skullkeyboards_exposed_63	✓	
com.khampat.damdawi.in_exposed_63	✓	
com.khampat.damdawi.in_exposed_158		✗
com.khampat.damdawi.in_exposed_165	✓	
com.kisstakoala.vehiclesfree_direct_1	✓	
com.mademin.avoidthecircles_direct_1	✓	
com.mademin.avoidthecircles_exposed_47		✗
com.mfoundry.mb.android.mb_252070299_exposed_331	✓	
com.tdelphiblog.LazyShaker_exposed_33		✗
lwcr46lion.lwp_exposed_360		✗
com.futurice.android.reservator_17_exposed_2	✓	
com.junjunguo.pocketmaps_8_exposed_22	✓	
com.newsblur_138_direct_1	✓	
com.newsblur_138_direct_5	✓	
com.nextcloud.client_10040299_exposed_13		✗
com.nextcloud.client_10040299_exposed_81	✓	
com.nextcloud.client_10040299_exposed_108	✓	
net.mypapit.mobile.myposition_12_direct_1	✓	
org.ligi.gobandroid_hd_258_exposed_35	✓	

phase due to anomalous usage of APIs. *PREV* correctly reported 30 apps as vulnerable (as discussed in Section 5.7.4). The remaining 47 apps are not reported as vulnerable because our final test generation phase was unable to generate proof-of-concept test cases. Out of these 47 apps, 16 are open source apps and thus, we were able to manually inspect the source code of these 16 apps. Manual investigation revealed the following:

- Five apps are actually vulnerable but missed by our tool. The reason is because our test generator was unable to generate the intent messages as required by those apps due to the similar problems explained above (missing intent-filter specifications).
- Four apps are not exploitable as the components are protected by custom permissions
- Seven apps involve UI event-based paths that include user interactions and hence, are not vulnerable (see Precondition $PR_{5.1}$ in Section 5.2.2).

In summary, the second set of apps contains 35 vulnerable apps. *PREV* detected 30 of them and missed five vulnerable apps.

Considering these results we can formulate the subsequent answer to $RQ_{5.3}$:

> PREV *detected (15+30) out of (20+35) vulnerable apps and missed 10 cases. The implication is that security analysts can use* PREV *to detect 82% of the apps containing permission re-delegation vulnerabilities.*

5.7.7 $RQ_{5.4}$: Comparison

To investigate $RQ_{5.4}$, we compare *PREV* with *FlowDroid* [8], *Covert* [13], and *IccTA* [56]. *FlowDroid* is a state-of-the-art tool that implements static taint analysis for Android apps. *Covert* is a compositional analysis tool where apps bundle is analyzed together to see if there is a potential composite ICC vulnerability. Analyzing a single app, however, provides the potential vulnerabilities a malicious app can exploit in the app under analysis. *IccTA* is ICC (and Inter-app communication) based taint analysis tool that is developed on top of FlowDroid and IC3 [70]. IC3 is used to resolve targets in ICC, while FlowDroid is used to perform static taint analysis. We configured the sources for *IccTA* to be the ICC method `getIntent()` and all the APIs that require special permission as sinks (Pscout [10]). Flows detected in such a way can reveal permission re-delegation if data that is sent from another app can is used in someway in a sensitive sink API.

We ran *FlowDroid*, *Covert*, and *IccTA* on the open source apps (595 open source apps that we used in Section 5.7.4 and 20 mutants that we used in Section 5.7.6). We ran these tools on the closed source apps as well, but we shall only discuss their results based on the analysis of open source apps. This is because these tools reported a large number of vulnerabilities in the closed source apps and it was difficult to verify them because we cannot inspect the source code and the tools do not generate proof-of-concept test cases.

Results of FlowDroid. We configured *FlowDroid* to detect permission re-delegation cases (as done in [26]) by setting `getIntent()` as taint sources and all the privileged API calls (from [10]) as sinks. However, we kept the default analysis options. Thus, *FlowDroid* reports any tainted data from public entry points reaching to the sinks as permission re-delegation vulnerabilities, in terms of source-sink pairs.

The results of *FlowDroid* on open source apps are shown in Table 5.4. The first column shows the names of the apps reported as vulnerable by FlowDroid. The second column reports whether the reported vulnerable app is a true positive; the last column

reports whether the reported vulnerable app is a false positive. In total, FlowDroid reported 8 open source apps as vulnerable; all of them are due to the APIs that requires the special permission to access Internet. However, manual checks conducted using the guidelines presented in Section 5.7.4 revealed that, although cases of permission re-delegation indeed occur, all the cases are legitimate because they are clearly intended by the developers; hence they are false positives (reported in 'FP' column).

Table 5.4 Vulnerability report of *FlowDroid* on open source apps.

App	TP	FP
ch.fixme.status		✗
com.mschlauch.comfortreader		✗
com.ringdroid		✗
cz.romario.opensudoku		✗
de.jkliemann.parkendd		✗
de.syss.MifareClassicTool		✗
org.mult.daap		✗
sk.halmi.fbeditplus		✗

Regarding the 20 mutated apps, *FlowDroid* did not report any of them as vulnerable. Thus, it missed all the vulnerabilities.

FlowDroid did not detect vulnerabilities (false negatives) because it applies data-flow analysis only, while our approach includes control-flow analysis as well. In particular, *FlowDroid*, like any other taint analysis based approach, reports a vulnerability only when data from taint sources (e.g., public entry points that may receive intent messages from other apps) are used in the sinks (e.g., privileged API calls). But in the context of permission re-delegation vulnerabilities, some privileged API calls require no input data, e.g., the act of turning on/off the WiFi is a privileged action that require no input data (i.e., just an event of receiving an intent message could trigger this action). As such, *FlowDroid* cannot detect it. By contrast, our approach does not impose such a constraint.

Conversely, *FlowDroid* sufferred false positives, because it does not distinguish between legitimate cases of permission re-delegation and vulnerabilities. In fact, any data dependency between Intent data and data used in a privileged call is reported as a vulnerability by *FlowDroid*, even if it is a legitimate case of permission re-delegation, i.e., a feature specified in the app description.

For example, *com.github.nicolassmith.urlevaluator* is an app that receives shortened URLs and expands them into the longer versions, and this feature is exposed as a service

to other apps. To complete this task, the app needs to query a remote short-URL repository to get the long URL versions. *FlowDroid* is correct in reporting that there is a path from a public entry point (using `getIntent()` to read incoming data from other apps) to a privileged API `openConnection()`. However, since this is an intended feature of the app as clearly specified in the app description, the vulnerability report is a false alarm.

***Results of* Covert.** The results of *Covert* on the open source apps is shown in Table 5.5. The first column shows the names of the apps reported as vulnerable by *Covert*. The second column reports whether the reported vulnerable app is a true positive and the last column reports whether the reported vulnerable app is a false positive. *Covert* reported hundreds of vulnerabilities in the open source apps. However, only 18 of them are related to permission re-delegation (Intent spoofing). Out of these 18 vulnerabilities, only one is a true positive and the rest are all false positives. We manually investigated each vulnerability in the source code of the apps and classified the false positive reports as follows:

- Intentional behavior: Some reported apps receive data from other apps (components) and use privileged APIs. These are the cases of permission re-delegations. But those apps also explicitly declare them as intended features in their app descriptions. For example, `com.newsblur`, `com.mschlauch.comfortreader`, `com.duckduckgo.mobile.android` and `se.anyro.nfc_reader` are browser, NFC reader and news/document readers apps, respectively; and this was clearly declared in their Play Store descriptions. Therefore, these are false positive reports. Most reports fall under this category.

- Intra-component Intent: In some apps, the incoming Intent is from component within the same app (i.e., result of `startActivityForResult` call). Hence, these vulnerability reports are false positive as the Intent originates from the same app. This is observed in apps such as `org.marcus905.wifi.ace` and `com.hectorone.multismssender`.

- Private components: In some apps, the reported components are not exported. Therefore, they are only accessible within the same app and thus, not exploitable.

- System only Intent: In some apps, Intent returned from AccountManager is reported to be potentially dangerous. However, the intent comes from the Android system and we consider this as safe and hence, classify the report as false positive.

Some reported cases correspond to more than one category above.

Regarding the 20 mutated apps, *Covert* report none of them as vulnerable. Thus, it missed all the vulnerabilities.

Table 5.5 Vulnerability report of *Covert* on open source apps.

App	TP	FP
be.brunoparmentier.openbikesharing.app		✗
com.briankhuu.nfcmessageboard		✗
com.duckduckgo.mobile.android 1		✗
com.hectorone.multismssender		✗
com.mschlauch.comfortreader		✗
com.newsblur		✗
com.seafile.seadroid2		✗
com.xperia64.timidityae		✗
de.hirtenstrasse.michael.lnkshortener		✗
de.jkliemann.parkendd		✗
de.syss.MifareClassicTool		✗
de.yazo_games.mensaguthaben		✗
net.kervala.comicsreader		✗
org.glucosio.android		✗
org.marcus905.wifi.ace		✗
org.nerdcircus.android.klaxon		✗
org.tigase.messenger.phone.pro	✓	
se.anyro.nfc_reader		✗

***Results of* IccTA.** Table 5.6 presents the results of running *IccTA* on our open source apps dataset. *IccTA* produced several reports for most apps. After manually investigating each report, we found out that all reports are false positives. Similar to *Covert*, we classified each false positive report as follows:

- Involves user interaction: Apps such as `com.newsblur`, `com.ringdroid`, `sk.halmi.fbeditplus` and `org.smc.inputmethod.indic` require the user to interact. If a user is involved, it is either an intended behavior or an action that can be aborted by the user. Therefore, we do not consider this report as a vulnerability (See *Precondition* $PR_{5.1}$ in subsection 5.2.2).
- Private components: In some apps, the reported components are not exported. Therefore, they are only accessible within the same app and thus, not exploitable. This was observed in `com.alfray.timeriffic`, `mobi.boilr.boilr`, `org.sixgun.ponyexpress` and `moe.minori.pgpclipper`.

- Overtainting: For apps such as `de.syss.MifareClassicTool`, *IccTA* produced a false positive because of overtainting. For example, an activity instance containing an untrusted field is tainted. This instance is then used in a callback function but the field does not actually influence the invocation of any privileged API; hence this is a false positive.

Regarding the 20 mutated apps, *IccTA* did not report any of them as vulnerable. Thus, it missed all the vulnerabilities.

Table 5.6 Vulnerability report of IccTA on open source apps.

App	TP	FP
com.alfray.timeriffic		✗
com.commonsware.android.arXiv		✗
com.newsblur		✗
com.mschlauch.comfortreader		✗
com.ringdroid		✗
cz.romario.opensudoku		✗
de.jkliemann.parkendd		✗
de.syss.MifareClassicTool		✗
mobi.boilr.boilr		✗
moe.minori.pgpclipper		✗
org.jfet.batsHIIT		✗
org.sixgun.ponyexpress		✗
org.smc.inputmethod.indic		✗
se.anyro.nfc_reader		✗
sk.halmi.fbeditplus		✗

Table 5.7 summarizes the overall comparison among *PREV*, *FlowDroid*, *Covert*, and *IccTA*.

Table 5.7 Summary of comparison between *PREV*, *FlowDroid*, *Covert*, and *IccTA*

	Open source apps		**Mutated apps**	
Tool	TP	FP	TP	FN
PREV	7	0	15	5
FlowDroid	0	8	0	20
Covert	1	17	0	20
IccTA	0	15	0	20

Considering these results we can formulate the subsequent answer to $RQ_{5.4}$:

> PREV *significantly outperforms* FlowDroid, Covert, *and* IccTA *in detecting permission re-delegation vulnerabilities, because these tools generated only false alarms and missed the vulnerabilities detected by* PREV, *except only one vulnerability detected by* Covert.

5.7.8 $RQ_{5.5}$: Robustness

PREV detects vulnerable apps based on the permission re-delegation models that are learned on a large number of "safe" (benign and non-vulnerable) training apps. The detection accuracy might degrade when the quality of training apps degrades. Even though we carefully selected the "safe" apps (see Section 5.4), there is still a risk that some apps with security defects are included in the training set. In this case, our models may characterize these defects and *PREV* would not detect them as anomalies according to these models.

However, since *PREV* adopts a threshold-based algorithm, we made the assumption that it is robust against the inclusion of a *small* number of non-safe apps in the training set. Here we validate this assumption and quantify the robustness against the presence of non-safe apps in the training set.

Given vulnerability v, occurring in the an app a that was assigned the cluster c, we define *robustness(v)* as the number of occurrences of vulnerability v that need to be included in the training set (and in particular in cluster c) to turn *PREV* not able to detect v.

To this aim, we gradually degrade the training set by replacing the safe apps with *contaminated apps* — apps that contain the same vulnerable permission re-delegation behaviors as those apps *PREV* correctly detected as vulnerable in Section 5.7.4. Learning is repeated on this new, degraded training set and the same experiment as in Section 5.7.4 is repeated, to assess if *PREV* is still able to detect the same vulnerabilities despite the contaminated training set. If *PREV* can still detect the same vulnerabilities, more and more contaminated apps are added to the training set and the process is iteration, until *PREV* can no longer detect the vulnerabilities because of the contaminated testing set; the number of contaminated apps at the last iteration gives us a robustness measure with respect to vulnerability v and cluster c.

More specifically, we consider the API frequency matrix M that was originally constructed at model inference step for a given cluster (see Section 5.4.3. An example of this matrix is shown in Figure 5.19(a). Rows represent apps in the training set

and columns represent privileged APIs; a cell contains the value 1 when the API in the column is reachable from a public entry point and the value 0 if the API is not reachable.

Camera.open() in Figure 5.19 is a privileged API associated with a permission re-delegation vulnerability that *PREV* detected on the subject app `com.appportunity.androidpreviewer`. *PREV* detected the vulnerability based on the permission re-delegation model learnt on the training apps of the cluster to which the subject app belongs (cluster 20). Now, we degrade this training set by modifying the API *Camera.open()* as reachable in $TrainingApp_2$, a training app from the same cluster as `com.appportunity.androidpreviewer`. This in fact corresponds to changing a value from 0 to 1 in the third column of the matrix. The change is highlighted in boldface in Figure 5.19(b). In this way, we contaminate the training set, by making the privileged but uncommon API *Camera.open()* more frequent and, thus, less likely anomalous.

	openConnection()	connect()	**Camera.open()**	setWifiEnabled()
$TrainingApp_1$	1	1	1	0
$TrainingApp_2$	1	1	**0**	0
$TrainingApp_3$	1	1	0	0
$TrainingApp_4$	1	0	0	1

(a)

	openConnection()	connect()	**Camera.open()**	setWifiEnabled()
$TrainingApp_1$	1	1	1	0
$TrainingApp_2$	1	1	**1**	0
$TrainingApp_3$	1	1	0	0

(b)

Fig. 5.19 API frequency matrix M. 1=the app exposes the API, 0=otherwise. (a) Original training set. (b) Degraded training set.

In this experiment, *PREV* still detected the vulnerability due to the reachable API *Camera.open()* when the original training set is degraded with *one contaminated app*; but it was not able to detect the vulnerability anymore when it is degraded with *two contaminated apps*.

The experimental results are shown in Table 5.8. We can observe that our approach can still detect the same vulnerable apps (as using the original training set) even if the training set includes a few contaminated apps, ranging from 2-7 apps depending on the cluster. The table also shows the sizes of the clusters to which the vulnerable apps belongs.

Table 5.8 Numbers of contaminated apps in training sets that make vulnerability detection fail

App	Number of Contaminated Apps	Cluster size
bestvalleygames.turningvalley	5	443
com.akgun.uknews	7	337
com.appportunity.androidpreviewer	2	438
com.appreka.mycoop	5	537
com.appsdv.smsmefitr	3	410
com.aurorasi.aurorasfa	5	537
com.bimandika.Congratulationsmalonepost	3	410
com.braingen.devanagarinotepad	5	469
com.dinosaur.dinosaur_vs_zombie	5	443
com.fmplural.radio	3	293
com.innogang.kollywoodNews	7	337
com.javirurro.games.spaceshipzigzag	3	410
com.josejoaquin.traductor	3	311
com.magmamobile.game.SpiderSolitaire2	3	410
com.netdania	5	537
com.npes87184.s2tdroid	7	337
com.rbsoftware.pfm.personalfinancemanager	5	537
com.reverbnation.artistapp.i739749	7	337
com.softdx.qrscanner	5	537
com.superfanu.bryantbulldogrewards	3	410
com.vent	5	537
lv.delfi.ru	5	469
piproduction.frankthejew	3	410
com.mendhak.gpslogger	4	380
com.seafile.seadroid2	2	537
org.ligi.ajsha	2	537
org.linphone	2	380
org.tigase.messenger.phone.pro	5	453
org.totschnig.myexpenses	5	537
org.ttrssreader	5	410

This data is also shown in Figure 5.20. The histogram displays the distribution of robustness for the different vulnerabilities, that *PREV* detected in official apps. The first bar on the left-hand side shows that 4 vulnerabilities have robustness=2. It means that these cases can be still detected when the training set is contaminated by adding

an app with this same vulnerability, but they cannot be detected if contamination consists of 2 apps with the same vulnerability.

The majority of the vulnerabilities — 13 apps corresponding to the highest bar in Figure 5.20 — can be still be detected with a quite contaminated training set (robustness=5).

The highest robustness (the last bar at the right-hand side of the graph) is observed for 4 apps. To make *PREV* fail in detecting these cases, the training set should be contaminated by adding 7 apps with the same vulnerability.

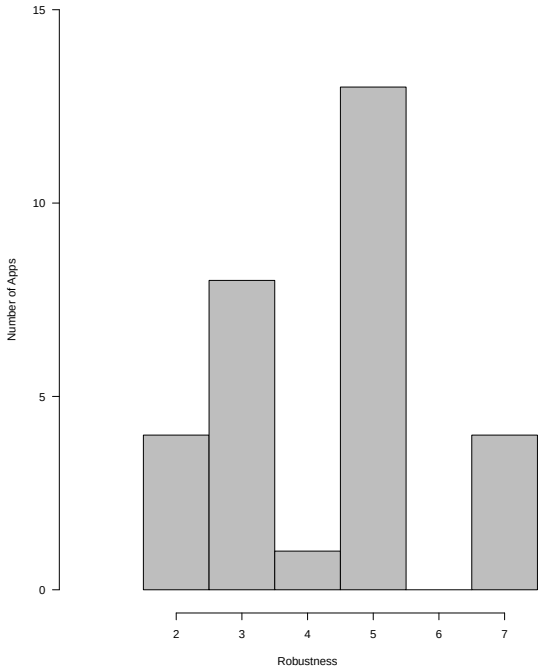


Fig. 5.20 Histogram for level of training set contamination

Considering these results we can formulate the subsequent answer to $RQ_{5.5}$:

PREV *is robust against the inclusion of a small number of non-safe apps.*

5.7.9 $RQ_{5.6}$: Threshold

The vulnerability detection algorithm is based on the boxplot approach, and a case is classified as an outlier when its distance d is larger than the threshold $t_{outlier}$. We are interested in studying the impact of different thresholds on the vulnerability detection capability of our approach.

To investigate this research question, we consider the list of apps correctly detected as vulnerable in Section 5.7.4. It consists of 30 apps (either open source and closed source), which have been classified as vulnerable by *PREV* and have been manually verified.

The experiment consists of changing the threshold value and verifying how many of these apps can still be detected as vulnerable. We expect that the larger the threshold we use, the fewer vulnerable apps would be detected.

To compute new threshold values, we add a multiplication factor α in the definition of threshold from Section 5.4.3. The threshold definition is updated as follows:

$$t_{outlier} = Q_3 + \alpha 1.5(Q_3 - Q_1)$$

If $\alpha = 1$, we have our original definition of threshold. New threshold values are obtained by using different values of α, and are used to repeat the classification procedure.

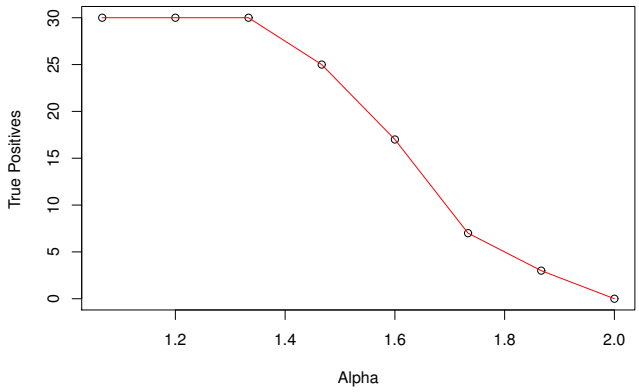


Fig. 5.21 Impact of different thresholds on vulnerability detection.

Figure 5.21 shows the results of this experiment. It shows the true positives against the increasing values of α. The number of true positives is the same until $\alpha \leq 1.33$. It then drops significantly and reaches 17 (nearly half of the initial value of 30) at $\alpha = 1.6$. Eventually, no vulnerability can be detected at $\alpha \geq 2$.

Considering these results, we can answer to RQ_6 in this way:

PREV *is sensitive to the threshold used to detect outliers. In particular, the number of detected vulnerabilities drops to almost half when increasing the threshold by a 30% factor, and no app is classified as vulnerable with a threshold increased by a 100% factor.*

5.7.10 Limitation and Discussion

Despite the positive results obtained, our current approach may be affected by some limitations that we discuss here:

Static analysis. Data sanitization functions sanitize input data from other apps or discard them when the data violates security policies. Hence, there may be no security issue when privileged actions are performed using sanitized data. Currently, our static analysis step does not deal with data sanitization. Therefore, we cannot distinguish between the malicious data and the benign data. As a result, our static analysis approach might be prone to false positives.

We detect vulnerabilities that involve the interaction among two apps, i.e., an attacker app that sends an intent message to the AUT via public entry points. So our analysis is *inter-app*. However, when the message reaches the AUT, our static analysis technique is limited to intra-component. If the component of the AUT, which receives the attacker app's message, interacts with a different component, i.e. if a path exists between two or more different components of the AUT, our approach will miss the path and thus, may result in a false negative.

Moreover, our static analysis misses if sinks are invoked via reflective calls or are in native code. Call graphs generated by static analysis are in general incomplete and unsound. Static analysis may miss call edges in the presence of complex code such as obfuscated code, native code and reflection and dynamic code loading. This could result in false negatives when the missing edges are those from public entry points to a privileged API. It may also generate spurious additional edges for code such as Thread runnables. In this case, it may result in false positives.

Test generation. Like any other test generation-based approach, the code coverage that can be achieved by our test generation is not complete. Therefore, it may not be able to generate proof-of-concepts for some of the target paths. This again may result in false negatives. In future, we could mitigate this by improving our test generation algorithm, e.g., by applying a better fitness function that combines node-based and edge-based formulations [90]. An alternative approach is to use symbolic execution, which might provide better code coverage. But it also comes with its own limitations; while it is generally not scalable due to path explosion and constraint solving problems [19], it will be challenging to perform symbolic execution of Android code which have no single entry point and which tends to be highly event-driven.

Inadequate training. If our training apps are not sufficient or representative of real world, safe apps, the inferred models would be incomplete. Consequently, our approach may be inaccurate. For example, incomplete training set could miss some cases of

legitimate permission re-delegation. These cases would be later classified as anomalous when occurring in the apps under test and they will be classified as security problems, resulting in false positives. Conversely, if our training set contains more non-safe apps than the thresholds, it would result in false negatives. Our best effort to address this limitation was to use only apps from top, most-downloaded apps from well-reputed companies as the training apps. Since the learning requires tens of thousands of apps, it is impractical to manually check all of them and make sure they are all actually "safe".

App description. Our approach is based on the apps descriptions available in the App Store for learning the permission re-delegation models. We assume that these descriptions are consistent with apps' permission re-delegation behaviors, i.e., apps with similar descriptions implement similar permission re-delegation behaviors. However, this assumption could be violated in practice; in fact two apps with a similar description could implement different permission re-delegation behaviors.

A related problem is in our AUT, when its description is not informative or not detailed enough to assign it to the correct cluster. In this case, a sub-optimal cluster might be identified for the AUT, leading to incorrect classification of its behavior.

Dynamic permissions. In recent versions of Android, permissions can be granted and revoked by the user during runtime. Our approach does not consider this dynamic revocation possibility. Instead, our approach assumes the worst case scenario, where all the permissions requested by a vulnerable app are always granted by the user and, thus, if they are exposed, a permission re-delegation vulnerability is reported.

To summarize, our approach, like many other approaches based on machine learning and program analysis (that has various practical limitations), is neither sound nor complete. However, based on our empirical results, we can argue that our tool resulting from such an approach can automatically detect many permission re-delegation vulnerabilities with a very low false alarm rate (zero false alarm in our experiments) in a matter of minutes. Hence, the implication is that while our approach is neither sound nor complete, it has practical benefits since the detected vulnerabilities come at almost zero cost. As we make our tool and dataset publicly available, researchers can replicate and/or repeat the experiments to validate or refute our claims.

5.8 Chapter Summary

In this chapter, we extended the threat model of Chapter 4 and we presented a novel approach to precisely detect and test permission re-delegation vulnerabilities by

combining static analysis, natural language processing, machine learning, and genetic algorithm-based test generation techniques. Our approach detects vulnerable apps that abnormally expose privileged actions to other (potentially malicious) apps and distinguishes them from legitimate permission re-delegation cases. It also generates proof-of-concept attacks to prove the vulnerabilities and security reports to document them. We evaluated our approach on 1,258 real world apps from official Google Play store. Our approach automatically detected 30 apps that are vulnerable to permission re-delegation attacks without any false alarm, significantly outperforming static analysis tools — *FlowDroid*, *Covert*, and *IccTA*. The analysis was done in a matter of minutes.

CHAPTER 6

Detecting Second Order Permission Re-delegation Vulnerability

In previous chapters, we presented different automatic approaches to detect permission re-delegation vulnerabilities in Android apps. In this chapter, we extend the threat model to capture what we call *Second Order Permission Re-delegation* composite scenarios. Moreover, we propose an automatic technique to detect this variant of vulnerabilities.

As defined by Felt et al. [33], permission re-delegation vulnerability occurs when an app performs a privileged action on behalf of another app that does not hold the required permission. The scenario to exploit this vulnerability involves *two* apps and is shown in Figure 6.1. In this example, a malicious attacking app sends a crafted intent to exploit a vulnerable utility app to toggle the WiFi setting. The vulnerable utility app uses the available Android framework API `setWifiEnabled(boolean)` to perform this privileged action that requires the `CHANGE_WIFI_STATE` permission.

Permission re-delegation vulnerability detection techniques presented in the previous chapters assumed that a privileged operation was performed when a permission-protected API method (e.g., `setWifiEnabled(boolean)`) is invoked after receiving an Intent from an attacking app.

However, a more complicated scenario involves system apps, that cannot be detected by simply identifying invocation of Android framework API methods in the app-under-analysis. This scenario is shown in Figure 6.2. In this example, the attacker app

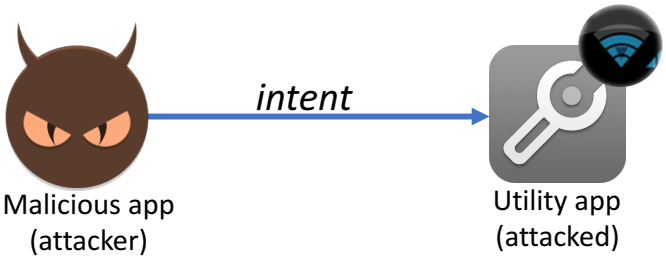


Fig. 6.1 Example of Permission Re-delegation.

(i.e., *Malicious app*) sends a specially crafted intent message to a vulnerable app (i.e., *Communication app*). Since the vulnerable app is granted the required permission, it in turn requests a system app (i.e., system *Phone* app) to complete a privileged action — making a phone call in this case. The system app performs such an action, because it recognizes that the request is coming from an app with the required permission, i.e., the *Communication app*. Even if the attacking app does not hold the `CALL_PHONE` permission that is required to make a phone call, the malicious app can effectively initiate a phone call using the vulnerable communication app that acts as a proxy. We call this second scenario *Second Order* Permission Re-delegation. This scenario is more complex than the previous one because:

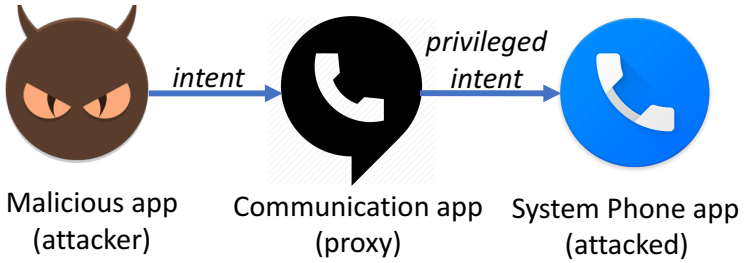


Fig. 6.2 Example of Second Order Permission Re-delegation.

- While the previous variant of this vulnerability involves just two apps (*attacker* and *attacked*), the new variant involves three apps (the *attacker*, the *proxy* and the *attacked system* app);
- It is hard for the *system* app to block the attack, because the request is coming from a benign (although vulnerable) *proxy* app that is granted special permissions by the end-user;
- The vulnerability in the *proxy* app cannot be detected by the existing automated approaches (e.g., [26, 13]) and techniques proposed in previous chapters, because

the vulnerability does not involve calls to permission-protected API methods from the Android APIs, but a primitive to send inter-component (or inter-app) Intent messages;

- Inter-component communication is a very common behavior in Android apps and cannot be considered a vulnerability in general. If outgoing messages were all considered suspicious, an overwhelmingly high number of security notices would have to be reported, thus, too many false alarms for such an approach to be effective. Therefore, a more fine-grained analysis is required.

Several multi-app analysis techniques were proposed in literature in order to analyze colluding apps [76, 13, 93, 28]. These techniques are based on the assumption that the colluding apps are present at the same time on the device and are collaborative in order to perform unintended operation (e.g., one app collects user data, the other app leaks to the Internet without raising alarm). Therefore, given a pool of apps, these approaches were mainly focused on exposing potential collusion rather than vulnerabilities due to programming mistakes.

Different from the technique proposed in Chapter 4, in this approach we do not set prerequisites on attacker controlled data because we base our analysis on call graph (instead of data flow) so that we can capture instances of this vulnerability that do not also involve attacker controlled data. In Figure 6.2, whether the malicious app controls the data (phone number) or not, if the Communication app performs the call (for example to a hard-coded telephone number), it's considered vulnerable as it might be costing the end-user money.

In this chapter, we describe *Second Order Permission Re-delegation* vulnerability and explain why existing multi-app vulnerability analysis techniques are not efficient in detecting it. We then, propose an approach based on static analysis to detect this vulnerability and to synthesize prof-of-concept attack scenarios that document how this vulnerability can be exploited. We, then, empirically assess the overall approach on more than 10k real world apps, showing first of all that this vulnerability is prominent among apps and, second, that our approach is effective in detecting and testing it.

Contributions: To summarize, the main contributions presented in this chapter are:

- We describe *Second Order Permission Re-delegation* vulnerability and we explain why existing multi-app vulnerability analysis techniques are ineffective in detecting it;

- We propose a novel approach based on static analysis to detect instances of this vulnerability;
- We resort to test case generation to synthesize prof-of-concept attack scenarios that document how this vulnerability can be exploited;
- We empirically assess the overall approach on more than 10k real world apps, showing that this vulnerability is prominent among apps and that our approach is effective in detecting and testing it.

6.1 Motivation and Updated Threat Model

```
1  public class MainActivity extends
       Activity {
2   public void onClick(View v) {
3    Intent intent = new Intent(this,
         Message.class);
4    intent.putExtra("num", gui.getNumber
         ());
5    intent.putExtra("msg", gui.
         getMessage());
6    startActivity(intent);
7   }
8  }

9  public class Message extends Activity
       {
10  public void onCreate(Bundle saved) {
11   Intent i = getIntent();
12   String dest = i.getExtra("num");
13   String text = i.getExtra("msg");
14   smsManager.sendTextMessage(dest,
         null, text, null, null);
15  }
16 }
```

(a)

```
1  public class MainActivity extends
       Activity {
2   public void onClick(View v) {
3    Intent intent = new Intent(this,
         Call.class);
4    intent.putExtra("num", gui.
         getNumber());
5    startActivity(intent);
6   }
7  }

8  public class Call extends Activity
9   public void onCreate(Bundle saved)
        {
10   Intent i = getIntent();
11   String dest = i.getExtra("num");
12   Uri uri = Uri.fromParts("tel",
         dest, null);
13   String action = Intent.
         ACTION_CALL;
14   Intent intent = new Intent(action
         , uri);
15   startActivity(intent);
16  }
17 }
```

(b)

Fig. 6.3 Example of two apps with (a) First-Order Permission Re-delegation and (b) Second-Order Permission Re-delegation vulnerabilities.

In this section, we update our previous threat model starting from a motivating example.

6.1.1 First Order Permission Re-delegation

Figure 6.3 contains two examples, corresponding to two apps with different variants of Permission Re-delegation vulnerabilities. The first example (in Figure 6.3(a)) shows

two components of an app meant to send SMS. The `onClick` method in `MainActivity` is activated when the end-user clicks on the *Send* button. This activity collects data filled by the end-user in the appropriate text fields. The destination number and the message to be sent (lines 4 and 5) are read from the graphical user interface and stored in an *Intent* message as the (extra) fields `num` and `msg`.

This Intent is sent to the public `Message` activity at line 6. This second activity is in charge of dispatching messages. It reads destination and message content from the Intent (line 12 and line 13, respectively) and then sends the actual SMS message at line 14 using the `sendTextMessage` API method. To succeed, this API call requires the app to be granted the special permission `SEND_SMS`.

However, this app is vulnerable to Permission Re-delegation because of a programming mistake. In fact, the `Message` activity is public, i.e., the component accepts Intents not only from components within the same app (as it should be), but also from any other apps installed on the same device.

Thus, an attacker app without the `SEND_SMS` permission could exploit `Message` activity to send SMS on its behalf. The attacker just needs to send an Intent with arbitrary destination number and message content to `Message` activity that, without checking the sender's permission, would perform the privileged operations and sends the SMS.

The approaches proposed in Chapter 4 and Chapter 5 could be used to detect this vulnerability.

6.1.2 Second Order Permission Re-delegation

Figure 6.3(b) shows a TELEPHONY app meant to let the user make phone calls. This app contains a variant of the previous vulnerability.

When the end-user clicks on a button in the `MainActivity`, the (already typed) number to be called is retrieved from the graphical user interface at line 4 and used to fill the Intent that is then sent to the `Call` activity.

The `Call` activity reads the number from the incoming Intent (line 11) and uses it to create a URI that is eventually used to make the phone call. To make the phone call, instead of calling a permission-protected API method (as the SMS app did), this app instead prepares a new Intent that specifies the special *Action* string `"ACTION_CALL"` together with the number to call (line 14). This Intent is sent to the vendor-dependent Android system Phone app that eventually performs the actual phone call, after checking the sender's (i.e., in this case the TELEPHONY app) permissions. To use

the Action string `"ACTION_CALL"` in an Intent, this app must hold the `CALL_PHONE` permission. Otherwise the app is denied permission to send the Intent.

This app is also vulnerable, as the `Call` activity is exposed to other apps installed on the same device. This activity does not check the sender's permission before sending the privileged Intent to make the call. Therefore, this app is vulnerable to Permission Re-delegation.

6.1.3 Considerations

The SMS app completed the privileged task within its code by calling a permission-protected *API method*. Thus, we say that it is vulnerable to (First-Order) Permission Re-delegation.

The TELEPHONY app, instead, completed the privileged task by deferring the request to a system app, by sending an Intent with a privileged *Action string*. Thus, we say that this app is vulnerable to *Second Order* Permission Re-delegation, because another app (i.e., the system Phone app) is involved in the scenario.

First-Order Permission Re-delegation vulnerabilities can be statically detected by identifying static calls to privileged API methods (for example as done in [13, 33, 21, 26, 98, 97, 55, 24]). Second Order Permission Re-delegation vulnerabilities, however, need additional analysis to be detected. In fact, sending Intents is a quite common Android ICC process, and most of the time it does not require permissions. Tagging all the places where Intents are sent as potentially vulnerable would result in a huge number of false positives. Precise string analysis is required to identify what *Action string* is used in the Intent, to distinguish regular Intents from privileged Intents. To the best our knowledge, no work addresses specifically Second Order Permission Re-delegation vulnerabilities.

Though Second Order Permission Re-delegation involves three apps (the attacker, the proxy and the attacked), previously proposed compositional analysis techniques for colluding apps such as [76, 13, 93, 28], cannot be used because of either of the following reasons:

- **Attacker app not present**: compositional analysis techniques require bundles of apps to be present in order to infer their possible interaction. Given an app under analysis, determining if it has Second Order Permission Re-delegation vulnerability should be done regardless of the apps it will be installed together with. If fact, attacking apps could update their code at any time, rendering the initial collusion analysis ineffective.

- **Attacked app not present**: in Second Order Permission Re-delegation, the attacked apps are system apps (e.g., the system Phone app) that complete actions on behalf of other apps that have the required permission. In order to perform an offline composite vulnerability analysis, the system apps should be present during analysis time. However, system apps are vendor dependent and can be heavily customized — an analysis with a bundle of apps from a given vendor could have a result different from another vendor.
- **API as sinks**: in previous works, sensitive security sinks are considered APIs that require permission. In Second Order Permission Re-delegation however, the sensitive sinks are the primitive ICC methods (e.g., `startActivity()`). Considering ICC is very common in Android apps and in most cases it is not security sensitive, a precise analysis is required in order to identify the communications that are sensitive.

6.2 Static analysis and Test Case Generation

6.2.1 Overview

The overview of our approach is shown in Figure 6.4. It is composed of two main phases (i) static analysis, to determine the presence of a potential vulnerability; and (ii) test case generation, to automatically generate a scenario that documents the security defect.

The static analysis phase takes a (compiled) Android app as an input and produces a list of candidate vulnerabilities, and the paths in the app call graph for each vulnerability.

The second phase applies instrumentation and deploys the test case generation technique introduced in Section 5.6.2 in order to generate Intent messages that execute the vulnerable paths detected in the first phase.

6.2.2 Static Analysis

Source-sink detection: In order to identify Second Order Permission Re-delegation vulnerabilities, we need to first define sources and sinks. Sources are all the public entry-points of exported components. First of all, we detect the exported components by listing those mentioned in the *manifest* file of the app under analysis. Exported components are those components for which an *intent-filter* is defined and/or those

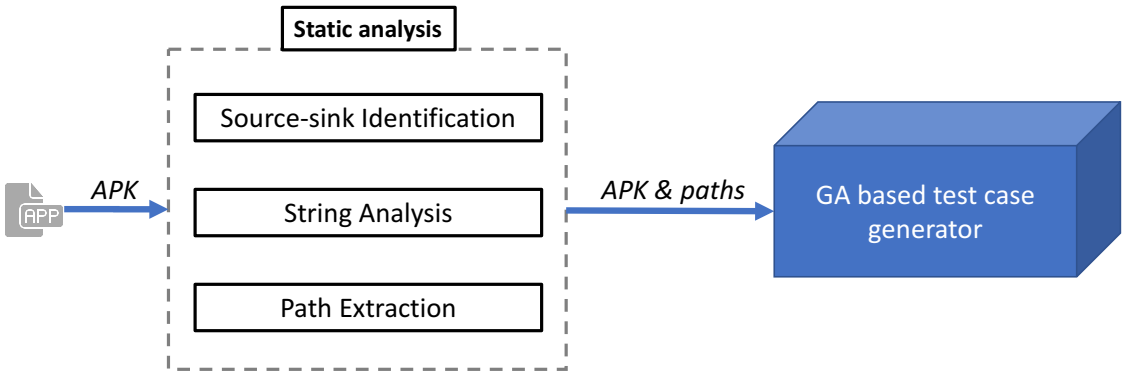


Fig. 6.4 Overview of the proposed approach. The static analysis phase takes a compiled Android app (APK file) and produces a modified version of the app and paths. The test case generation phase then generates inputs executing the paths.

that are explicitly marked as "exported" (i.e., the `exported` attribute of the component definition is set to `true`).

Then, the bytecode of these exported components is analyzed to detect their entry-points, such as `onCreate()` for an Activity and `onReceive()` for a BroadcastReceiver. The lists of entry-points for all the different Android components are taken from the documentation of the components lifecycle (for example Activity[1]).

Sinks are those Android ICC calls, i.e., the API methods that can be used to send Intents, such as `startActivity()`. The list of ICC methods that send Intents is available in the Android documentation[2].

However, considering the fact that ICC is the backbone of the Android framework, ICC is used very often and, most of the times, without involving privileged Action strings. Thus, labeling all calls to ICC API methods as sinks would produce overwhelmingly large number of false positives that would require laborious manual verification and filtering. Thus, a more usable security report should identify those ICC calls that involve privileged Action strings and could lead to Second Order Permission Re-delegation. To this end, we deploy string analysis.

String Analysis: Sensitive sinks should be filtered to keep those calls to ICC API methods that use privileged Action strings. For example, in Figure 6.3(b) `Call` class, line 15 is a sensitive sink because the Intent sent in `startActivity` uses the Action string `"ACTION_CALL"` (line 14), that requires the `CALL_PHONE` permission to be able

[1]https://developer.android.com/reference/android/app/Activity
[2]https://developer.android.com/reference/android/content/Context

to send it. Therefore, in order to classify an ICC API method call as a sensitive sink, we first need to compute the Action string used by the Intent.

Figure 6.5 shows three ways (out of 6) to create and send an Intent that are relevant to Second Order Permission Re-delegation. The remaining three are not considered as they are mostly used to instantiate internal or external components of non-system apps and are interesting for analysis of colluding apps (the attacked apps we consider are system apps). In Figure 6.5(a) and (b), the Action string is specified as an actual parameter in the call to the Intent constructor. In Figure 6.5(c), the empty constructor is called and the Action string is added later using the setter `setAction`.

We, therefore, apply the following algorithm to compute the Action string:

- For each sink (i.e., a call to ICC), we traverse data dependencies backward, until we reach a call to the constructor of the Intent used in the sink;
- From the call to the constructor, we traverse data dependencies in forward direction to detect calls to `setAction`;
- We perform constant propagation to compute the value of the Action string used as actual parameter of `setAction` (case in Figure 6.5(c)) and as actual parameter of the Intent constructor (cases in Figures 6.5(a) and (b)).

With the string values used as Action string, we are able to filter out regular ICC Intents that are not relevant to this vulnerability and we keep only a subset of sinks that are related to privileged task requests, i.e., those with privileged Action strings.

Path Extraction: The call graph of the app is then analyzed to identify the paths from public entry-points to the filtered sinks. The call graph is traversed backward in depth-first search manner starting from the sink nodes, until a public entry-point node is reached. During the visits, nodes are marked as visited so that loops in the graph are iterated at most once.

The output of this step is a list of paths from public entry-points to sinks. However, these paths could contain false positives if, for example, the app performs input validation. In order to reduce false positives that could have arisen because of such inclusions, we resort to fuzzing to see if an attacking app could make the potentially vulnerable app perform the privileged task.

```
1  Intent intent = new Intent(Intent.ACTION_CALL);
2  ...
3  startActivity(intent);   // sink b/c of tainted intent
```

(a)

```
1  Intent intent = new Intent(Intent.ACTION_CALL,
2  Uri.parse("tel:" + number));
3  ...
4  startActivity(intent); // sink b/c of tainted intent
```

(b)

```
1  Intent intent = new Intent();
2  intent.setAction(Intent.ACTION_CALL);
3  ...
4  startActivity(intent); // sink b/c of tainted intent
```

(c)

Fig. 6.5 Variants of ICC to send Intents and make a phone call.

6.2.3 Test Case Generation

Once list of paths is available from the static analysis phase, the next step is to generate test cases that execute the paths. To this end, we first instrument the app to trace method execution and install the app on an Android emulator.

The generated test cases are Intent messages that are sent to the app under test via the Android Debug Bridge (ADB). Our goal is to generate at least one intent message that exercises a given path. We generate test cases using the same genetic algorithm based approach that was presented in Section 5.6.

6.3 Empirical Validation

In this section, we evaluate the relevance and the effectiveness of our approach. The empirical investigation is guided by the following research questions:

- **RQ**$_{6.1}$: What are the most commonly privileged Action strings used in privileged Intents?
- **RQ**$_{6.2}$: Is the proposed approach effective in detecting actual Second Order Permission Re-delegation vulnerabilities?
- **RQ**$_{6.3}$: How long does the proposed approach take to analyze an app?

The first research question $\mathbf{RQ}_{6.1}$ investigates how often privileged Intents are used by privileged apps to verify if it is a prominent scenario. In fact, only when privileged Action strings are used, Second Order Permission Re-delegation vulnerabilities are possible. The second research question $\mathbf{RQ}_{6.2}$ investigates whether the proposed approach could be useful to a developer, to detect real security problems. Finally, $\mathbf{RQ}_{6.3}$ investigates the cost of using our approach in terms of the time taken to analyze a given app. A short analysis time is beneficial for tool adoption in a real production environment.

6.3.1 Subject Apps and Experimental settings

We considered apps coming from distinct sources:

- *DataSet*$_1$*: Popular app.* We collected in total 10K+ from the official Android app store, i.e., Google Play. They are collected by downloading those apps that were listed as the most popular in 2015.
- *DataSet*$_2$*: Non-popular apps.* We took 600 open source apps from F-Droid[3] that are also available on Google Play (i.e., real world open source apps). Moreover, we randomly sampled 600 not so popular (less than 1 million downloads) Google Play apps from the Androzoo [3] repository.

The experiment has been conducted on a machine equipped with an Intel Core i7 2.4 GHz processor, 16 GB RAM running Apple Mac OS X 10.11. For input generation, we used a Nexus 5 Android 4.4 emulator running on the same host machine.

6.3.2 RQ$_1$: Popularity of Privileged Actions Strings

Unfortunately, the official documentation of Android misses a single place that maps Intent Action strings with the permissions they require the app to hold. This information is partial to some Action strings, and spread in several distinct places. Thus, we decided to collect the privileged Action strings from their actual uses in apps.

We applied static analysis presented in Section 6.2, to get the list of all the Action strings used by popular apps (*DataSet*$_1$). Moreover, we also considered the Action strings available in Pscout [10]. However, the Action strings in Pscout are partial as they are referring to old Android versions, e.g., they miss the `INSTALL_PACKAGE` Action string which can be used to install additional packages onto a phone without the user

[3]https://f-droid.org/en/packages/

knowing. Moreover, the Action strings in Pscout are not labeled as privileged (that can only be sent by the system) or normal (that can also be sent by normal apps).

All these Action strings need to be manually filtered to keep only the privileged ones. Manual filtering consists in checking each Action string in the Android documentation to verify if a particular permission is needed to use it in an Intent. When the documentation was not exhaustive to label an Action string, we checked it against the Android source code. The final list of Action Strings is presented in Table 6.1.

android.intent.action.CALL
android.intent.action.CALL_PRIVILEGED
android.intent.action.INSTALL_PACKAGE
android.intent.action.UNINSTALL_PACKAGE
android.intent.action.SET_ALARM
android.intent.action.SET_TIMER
android.media.action.IMAGE_CAPTURE
android.media.action.VIDEO_CAPTURE
android.settings.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
com.android.launcher.action.INSTALL_SHORTCUT
com.android.launcher.action.UNINSTALL_SHORTCUT
android.bluetooth.adapter.action.REQUEST_ENABLE
android.bluetooth.adapter.action.REQUEST_DISCOVERABLE

Table 6.1 Extracted Action Strings

After completing the Action string labeling process, we went back to apps in *DataSset1* and we counted how often they are used to send Intents.

Figure 6.6 show how often privileged Action strings occur in our dataset, i.e., how often they are used by developers to create Intents.

As we can see, the most common privileged Action strings are those related to capturing pictures (in 720 apps) and videos (in 207 apps). Then, requesting the user to enable Bluetooth is the third common task (143 apps). Adding app shortcuts on the home screen and removing apps are also a quite common privileged task (70 and 51 apps, respectively). Other relevant tasks are turning the discoverability of the device by other devices (24 apps), removing shortcut from the screen (22 apps) and setting the alarm (22 apps).

Considering these results, we can formulate the subsequent answer to $\mathbf{RQ}_{6.1}$:

Usage of privileged Action strings to request tasks to other apps is common in Android apps, and the most frequently occurring privileged Action strings are related to taking pictures and recoding videos.

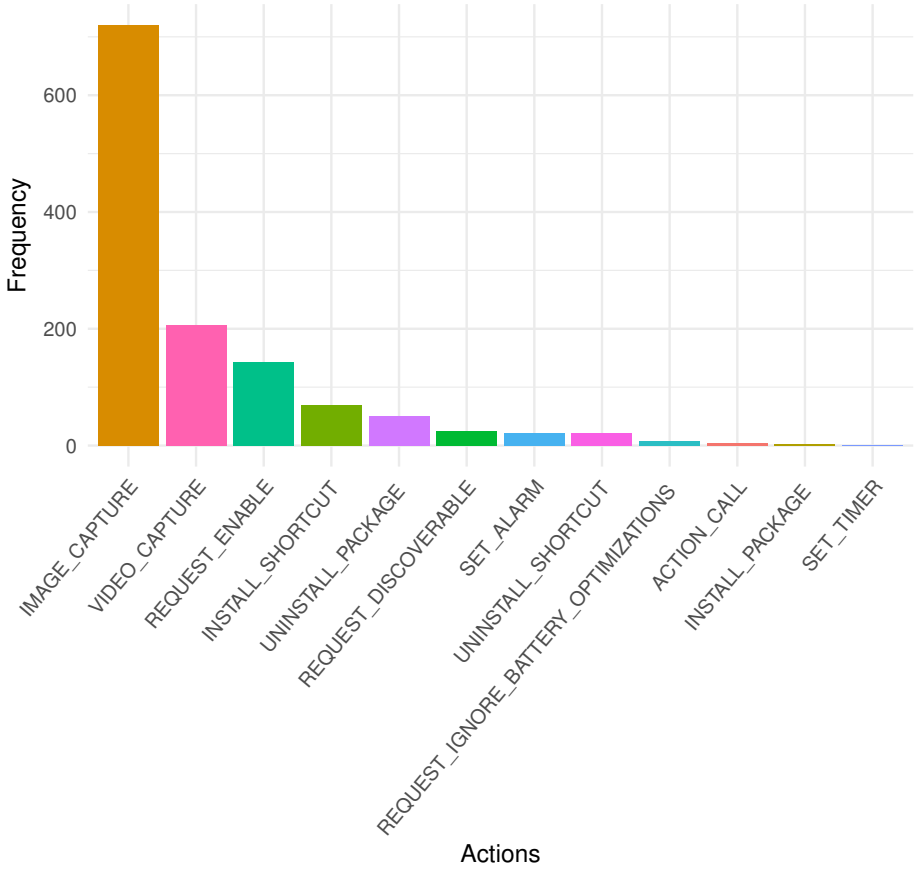


Fig. 6.6 Privileged actions and their frequencies.

6.3.3 $RQ_{6.1}$: Detection of Vulnerable Apps

The proposed approach was applied on all the data sets, i.e., $DataSet_1$ and $DataSet_2$. Apps are subject to static analysis to detect candidate vulnerabilities. Since static analysis is conservative and might result in several false positives, we apply the second phase of our approach, namely, ICC fuzzing. Test case generation creates proof-of-concept attacks for real vulnerabilities and can be used to debug, fix and test vulnerable apps.

Table 6.2 shows the results, dividing apps for the source they came from, 19 vulnerable apps from Google Play, 3 from Androzoo and 5 from F-Droid. For each app (first column) the table report the privileged Action string used to send privileged Intents.

The proof-of-concept test cases have then been manually validated. The most common Action strings occurring in vulnerable apps is `INSTALL_SHORTCUT` with 11

vulnerable apps, then `IMAGE_CAPTURE` is used in 9 vulnerable apps, `CALL` occurs in 4 apps and `REQUEST_ENABLE` occurs in 3 vulnerabilities.

App name	Action string
Top Google Play apps	
cn.mstars.activity	CALL
com.mv.notas (though2 paths)	IMAGE_CAPTURE
com.myntra.android	IMAGE_CAPTURE
com.app.app909fe1a3ad62	IMAGE_CAPTURE
com.app.app017f4c0fe778	IMAGE_CAPTURE
com.app.appd4403633f62b	IMAGE_CAPTURE
com.fm.weaponmod	IMAGE_CAPTURE
app.fastfacebook.com	IMAGE_CAPTURE
com.clearhub.wl	INSTALL_SHORTCUT
com.gtp.nextlauncher.theme.bloodysweetlove (through 3 paths)	INSTALL_SHORTCUT
com.mosoyo.wildanimals	INSTALL_SHORTCUT
com.mosoyo.watergalaxy	INSTALL_SHORTCUT
com.mosoyo.news7	INSTALL_SHORTCUT
com.mosoyo.bubbles6	INSTALL_SHORTCUT
com.moniappteam.iosnonofree.toptips4uuu	INSTALL_SHORTCUT
com.fotoable.makeup	INSTALL_SHORTCUT
com.app.all.video.downloader	INSTALL_SHORTCUT
com.ImaginationUnlimited.Poto	INSTALL_SHORTCUT
collage.instagram.b612.retrica.camera360.picsart.fotoable.instamag	INSTALL_SHORTCUT
Androzoo apps	
com.mvl.CasinoArizona	CALL
com.app.app0e87ec29c687	IMAGE_CAPTURE
com.yinzcam.nfl.eagles	IMAGE_CAPTURE
F-Droid apps	
org.sipdroid.sipua	CALL
org.lumicall.android	CALL
org.thecongers.mtpms	REQUEST_ENABLE
a2dp.Vol	REQUEST_ENABLE
net.bluetoothviewer	REQUEST_ENABLE

Table 6.2 Apps vulnerable to Second Order Permission Re-delegation and privileged Action strings they expose.

In the following, we discuss some cases of those reported in Table 6.2, commenting the attack preconditions and the potential impact of an exploit.

CALL: This is probably the most intuitive vulnerability the corresponding attack scenario is shown in Figure 6.2. An attacker app (that is granted no privilege) cannot send Intents with the privileged Action string `CALL` (`Intent.ACTION_CALL`). Thus, the attacker hijacks a vulnerable app with this permission to accomplish this task.

The attacker sends a regular Intent to the vulnerable app. When receiving this Intent, the vulnerable app sends a second intent with the `CALL` Action string, that is received by the system Phone app to initiate a phone call that does not require user confirmation.

Moreover, when sending the Intent, if the attacker app is also able to control the data that is used as phone number by the vulnerable app, then the attacker app can make phone calls to arbitrary phone number, including to premium numbers that actually cost money to the end-user.

Best programming practices recommend to use the Action string `DIAL`[4], which lets the end-user decide whether to complete the phone call.

We have created a video[5] demonstrating one of the reported apps, namely Sipdroid, being exploited by a malicious app in a poof-of-concept attack scenario. This app is being used by millions of users.

INSTALL_SHORTCUT: If the `INSTALL_SHORTCUT` Action string is exposed, an attacker can exploit the vulnerable app in two ways depending on the following preconditions. (i) If the boolean Intent extra "duplicate" is not set to false, an attacker can perform denial-of-service attack by filling the user's home screen with multiple icons of the same app, thus making the end-user unable to launch other apps. (ii) If the attacker can control what Data, Action string and Category strings are used by the vulnerable apps to broadcast the privileged Intent to create the shortcut, the attacker could create a fake shortcut that resembles, for example, a banking app. The fake app could, then, redirect the end-user to a fake login screen, effectively mounting a phishing attack.

This permission should probably not be re-delegated to other apps. However, specific use cases might require this, for instance, when the app is launched the first time by the user by clicking on the app icon and the developer intends to create a home screen shortcut. In these cases, the developer could adopt security practices to minimize the risk.

- Saving in the shared preference the fact that the shortcut was already created (to avoid duplications);
- Before sending the Intent, setting the "duplicate" Extra to false, to enforce that no duplicate home screen icons are created;

[4]https://developer.android.com/reference/android/content/Intent
[5]https://youtu.be/ruhuBXDXiWU

- Not using data from incoming Intents to create shortcuts. Alternatively, validating external data before using them to request shortcut creation.

IMAGE_CAPTURE: This Action string can be used by a privileged app to request the the camera app to capture an image and return it. Additionally, the Extra `EXTRA_OUTPUT` specifies the name of the file where to save the picture. If an app that uses this Action string is hijacked by an attacker who also controls Intent extra `EXTRA_OUTPUT`, then the attacker might specify a path she/he can access, where the captured image will be written. In this way, the attacking app can eventually take a picture without the need to be granted the `CAMERA` and the `WRITE_EXTERNAL_STORAGE` permissions.

We can answer $\mathbf{RQ}_{6.1}$ by stating that:

> The proposed approach is effective in finding Second Order Permission Re-delegation vulnerabilities, and proof-of-concept attacks were generated for 27 vulnerable apps.

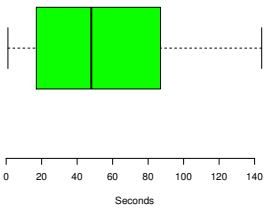
6.3.4 $\mathbf{RQ}_{6.3}$: Analysis Time

To investigate $\mathbf{RQ}_{6.3}$, we measured the amount of time spent to analyze the case study apps. To this aim, we instrumented the analysis script with the Linux date utility to log the time before starting the analysis and at its conclusion. Their difference (in seconds) is the actual amount of time spent performing static analysis. Once an app is statically detected to contain potential Second Order Permission Re-delegation, we run the genetic algorithm based test case generator to generate inputs that execute the vulnerability.

Figure 6.7 shows the boxplot of the time (in seconds) needed to perform static analysis, together with descriptive statistics (mean, median and standard deviation). On average, static analysis takes 54 seconds to complete on one app (including source-sink identification, string analysis and path extraction). In the worst case, the analysis takes up to 140 seconds (i.e., 2'20").

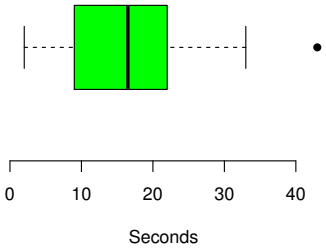
Figure 6.8 shows how long it took to complete test case generation (in seconds). Test case generation was quite fast, because none the vulnerabilities found with static analysis is tested in more than one minute. On average, it took just 16".

Based on these analysis time data, we can answer $\mathbf{RQ}_{6.3}$ in the following way:



mean	median	sd
53.85185	48	39.71505

Fig. 6.7 Time (in seconds) spent during static analysis (above) and descriptive statistics (below).



mean	median	sd
16.75	16.5	9.6

Fig. 6.8 Time (in seconds) spent during test case generation (above) and descriptive statistics (below).

The cost of out approach is affordable, because static analysis of an app takes less than 1 minute in the average case, and less then 2.5 minutes in the worst case, while test case generation took 16 minutes on average and less than one minute in the slowest case.

6.3.5 Limitations

The first step of our proposed approach is based on static analysis of compiled code. Hence, the proposed approach suffers from the inherent problems of static analysis techniques, such as not precisely reasoning about obfuscated codes. However, while obfuscated apps are found when getting them from the official app store, when this tool is used by software developers to assess their products, the non-obfuscated version is also available, to get more precise results.

The vulnerabilities detected by our approach have sources and sinks in the same component. Vulnerabilities that involve multiple components of the same app are not detected. For example, if Activity A is exposed but the sensitive sink (e.g., `startActivity`) resides in Activity B that is activated using ICC by Activity A, our prototype tool will not detect the vulnerability. This limitation could be overcome by using a *composite* constant propagation approach, as the one proposed by Octeau et al. [71], and extend our approach by performing inter-component static analysis.

Considering the fact that there is no documented list of Actions strings that require special permission, in this work we considered the Action strings extracted from popular apps from the official app store. Though the top apps could be representative of apps in the Google Play Store, it is possible that our list misses relevant Action strings. Detailed analysis of the Android code is needed to elaborated a more complete list of the privileged Action strings, similar to what done in Pscout [10] to identify privileged method calls.

6.4 Chapter Summary

In this chapter, we described a peculiar vulnerability that is overlooked and that threatens the security of Android apps, namely, Second Order Permission Re-delegation. We also proposed a novel approach based on static analysis to detect this novel kind of vulnerability in compiled Android apps. Moreover, we exploited genetic algorithm to automatically test these vulnerabilities and create proof-of-concept attack scenarios. Empirical results suggest that the proposed new vulnerability is common in popular apps and that our solution helps in identifying it.

CHAPTER 7

Anflo: Detecting anomalous information flows in Android apps

In the previous chapter, we presented a peculiar case of permission re-delegation vulnerability called Second Order Permission Re-delegation. We presented why current detection techniques for colluding apps do not work. We then presented our proposed static analysis based automatic detection technique augmented with genetic algorithm based test case generation. In this chapter, we presenet an approach for detecting vulnerabilities and malicious behaviors in Android apps when their code is anomalous with respect to information flow.

Often, to implement their features, apps may need to exchange data with other apps in the same smartphone or externally with a remote server. For instance, a camera app may share a picture with a multimedia messaging app for sending it to a friend. The messaging app, in turn, may send the full contacts list from the phone directory to a remote server in order to identify which contacts are registered to the messaging service so that they can be shown as possible destinations.

As such, sensitive information may *legitimately* be shared with other apps or remote servers. On the other hand, sensitive information might be exposed unintentionally by defective/vulnerable apps or intentionally by malicious apps (malware), which threatens the security and privacy of end users. Existing literatures on information leak in smartphone apps tend to overlook the difference between legitimate data flows and potentially malicious information leak. Whenever information flow from a sensitive

source to a sensitive sink is detected, either statically [91], [72], [70, 56, 8], [87], [65], [53] or dynamically [29], it is reported as potentially problematic. The common sensitive sinks in these works are logs, files, communication mediums such as SMS and the Intent.

In this chapter, we address the problem of detecting anomalous information flows with improved accuracy by classifying cases of information flows as either *normal* or *anomalous* according to a reference information flow model. Based on this classification approach, we will show that this technique can be used to identify different kinds of vulnerabilities including permission re-delegation and malicious behaviors. More specifically, we build a model of sensitive information flows based on the following features:

- Data source: the provenance of the sensitive data that is being propagated;
- Data sink: the destination where the data is flowing to; and
- App topic: the declared functionalities of the app according to its description.

Data source and *data sink* features are used to reflect information flows from sensitive sources to sinks and summarize how sensitive data is handled by an app. However, these features are not expressive enough to build an accurate model. In fact, distinct apps might have very different functionalities. What is considered legitimate of a particular set of apps (e.g., sharing contacts for a messaging app) can be considered a malicious behavior for other apps (e.g., a piece of malware that steals contacts, to be later used by spammers). An accurate model should also take into consideration the main functionalities that is declared by an app (in our case the *App topic*). One should classify an app as anomalous only when it exhibits sensitive information flows that are not consistent with its declared functionalities. This characteristic, which makes an app anomalous, is captured by the *App topic* feature.

In summary, our approach focuses on detecting apps that are anomalous in terms of information flows compared to other apps with similar functionalities. Such an approach would be useful for various stakeholders. For example, app stores (e.g., Google) can focus on performing more complex and expensive security analysis only on those apps that are reported as anomalous, before publishing them. If such information is available to end users, they could also make informed decision of whether or not to install the anomalous app. For example, when the end-user installs an app, a warning might be displayed stating that this particular app sends contact information through the Internet if this is anomalous with respect to other apps with similar functionalities.

In the context of BYOD (bring your own device) where employees use their own device to connect to the secure corporate network, a corporate security analyst might focus manual analysis on those anomalous flows that might compromise the confidentiality of corporate data stored in the end-user personal devices.

Contributions: To summarize, the main contributions presented in this chapter are:

- An automated, fast approach for detecting anomalous flows of sensitive information in Android apps through a seamless combination of static analysis, natural language processing, model inference, and classification techniques;
- The implementation of the proposed approach in a tool called *AnFlo* which is publicly available[1]; and
- An extensive empirical evaluation of our approach based on 596 subject apps, which assesses the accuracy and runtime performance of anomalous information flow detection. We detected 2 previous-unknown vulnerable apps related to anomalous flows. We also analyzed 18 malware apps and found anomalies in 6 of them.

7.1 Motivation

To deliver the expected features, apps often access sensitive data. It is important that code handling such data follows secure coding guidelines to protect user privacy and security. However, fast time-to-market pressure often pushes developers to implement code quickly, sometimes overlooking security implications and release apps without proper testing. As a result, apps might contain *defects* that leak sensitive data unintentionally. Apps may also contain *security vulnerabilities* such as permission re-delegation vulnerabilities [33], which could be exploited by malicious apps installed on the same device to steal sensitive data. Sensitive data could also be intentionally misused by *malicious* apps. Malicious apps such as malware and spyware often implement hidden functionalities not declared in their functional descriptions. For example, a malicious app may declare only entertainment features (e.g., games) in its description, but it steals user data or subscribes to paid services without the knowledge and consent of the end-user.

Defective, vulnerable and malicious apps all share the same pattern, i.e., they (either intentionally or unintentionally) deal with sensitive data in an anomalous way,

[1]Tool and dataset available at http://selab.fbk.eu/anflo/

i.e., they behave differently in terms of dealing with sensitive data compared to other apps that declare similar functionalities. Therefore, novel approaches should focus on detecting anomalies in sensitive data flows, caused by mismatches between expected flows (observed in benign and correct apps) and actual data flows observed in the app under analysis. However, the comparison should be only against *similar* apps that offer similar functionalities. For instance, messaging apps are expected to read information from phone contact list and share it with their server, but they are not expected to do the same with, for example, with list of installed apps.

Figure 7.1 shows examples of common and uncommon information flow in different categories of Android apps. Figure 7.1 (a) shows a messaging app (Communications category) collecting contacts information and sending it over the Intent. This could be acceptable as messaging app needs to check which user's contacts are also registered to the same messing service. According to the app's business logic, therefore, it is required to upload the contacts details. Figure 7.1(b) shows a fitness tracking app (Health and Fitness category) collecting heart rate and sending it to the remote server. This behavior is also is considered benign and correct because the fitness tracking app might require to perform complex processing on the data and show the progresses the user made in a dashboard.

Figure 7.1(c) instead shows a messaging app (Communications category) collecting heart rate information and passing over to their server. This behavior is suspicious as communications apps usually not collect health or fitness information and/or share via the Internet. Therefore, behaviors that are common for specific category of apps could be abnormal some other category.

7.2 Detection of Anomalous Information Flow

7.2.1 Overview

The overview of our approach is shown in Figure 7.2. It consists of two main phases — *Learning* and *Classification*. The input to the *learning phase* is a set of apps that are trusted to be benign and correct in the way sensitive data is handled (we shall denote them as *trusted apps*). It has two sub-steps — feature extraction and model inference. In the feature extraction step, (i) topics that best characterize the trusted apps are inferred using natural language processing (NLP) techniques and (ii) information flows from sensitive sources to sinks in the trusted apps are identified using static

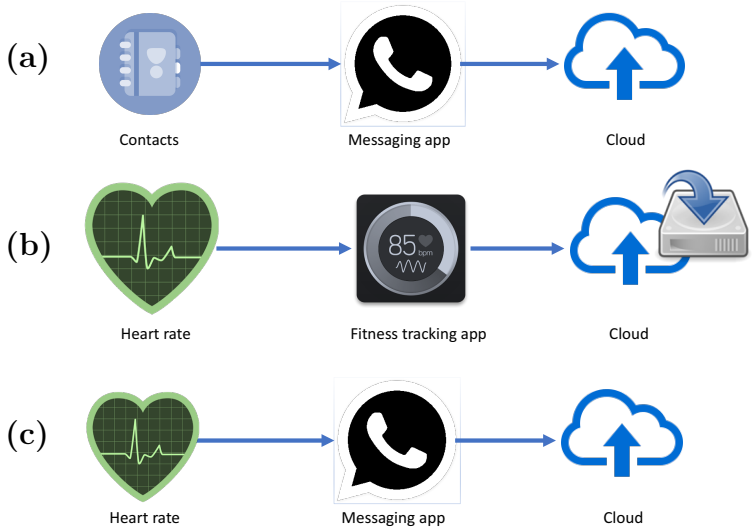


Fig. 7.1 Different sensitive information flow scenarios: (a) A messaging app sharing contacts details to their cloud server (b) A fitness tracker app sharing user heart rate detail to their cloud server (c) A messaging app sharing user heart rate detail to their cloud server

taint analysis. In the model inference step, we build sensitive information model that characterizes information flows regarding each topic.

These models and a given app under analysis (we shall denote it as *AUT*) are the inputs to the *classification phase*. In this phase, basically, the dominant topic of the AUT is first identified to determine the relevant sensitive information flow model. Then, if the AUT contains any information flow that violates that model, i.e., is not consistent with the common flows characterized by the model, it is flagged as *anomalous*. Otherwise, it is flagged as *normal*.

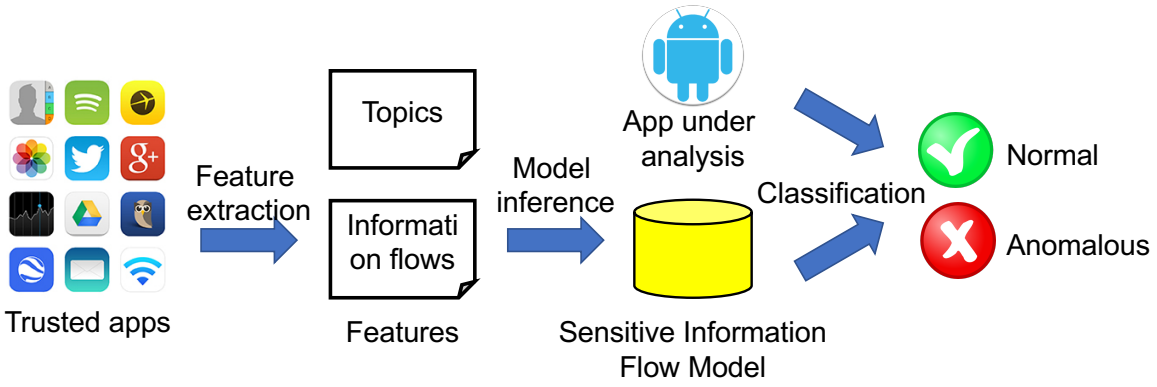


Fig. 7.2 Overview of the approach.

Though we did not encounter in our training set and subject apps, cases of more than one dominant topic (cases where the dominant topic is not a single topic but rather more than one topics equality contributing) can be handled by considering each dominant topic separately. For each dominant topic, the sensitive information flows present in the AUT are compared with the different models and the intersection of the reported anomalous flows is taken.

We implemented this approach in our tool *AnFlo*[2], to automate the detection of anomalous information flows. However, a security analyst is required to further inspect those anomalous flows and determine whether or not the flows could actually lead to serious vulnerabilities such as information leakage issues.

7.2.2 Feature Extraction

Topic Analysis

We adopted the same topic analysis technique discussed in Section 5.4.1. However, in this chapter, we cluster apps in a different way.

We fist apply *data pre-processing* (such as language detection, lemmatization and stop-word removal) to cleanse app descriptions and then we perform *topic discovery* (using the Latent Dirichlet Allocation (LDA) technique [14], implemented in a tool called Mallet [67]) on the pre-processed descriptions to extract the topics that are likely to be associated with.

Figure 7.3 shows the functional description of our running example app called *Best-Travel*, and the resulting output after performing topic classification that is described in Section 5.4.1. We define *Dominant Topic* as the topic with the highest probability according to the results of LDA. In Figure 7.3, "Travel" is the dominant topic as it has the higrhest probability of 70%. Then, the topics "Communication", "Finance", and "Photography" have the 15%, 10%, and 5% probabilities, respectively, of being the functionalities that the app declares to provide.

Note that, similar to Section 5.4.1, we did not consider Google Play categories as topics even though apps are grouped under those categories in Google Play. This is motivated by the recent studies [1, 42] that report NLP-based topic analysis on app descriptions produces more cohesive clusters of apps than those apps grouped under Google Play categories. However, we later investigate how results would change if we used Google Play categories instead.

[2]A demo video of the web app version of the tool in action can be found here: https://youtu.be/Meua3sJvL3g

The ultimate and most convenient way of traveling. Use BestTravel while on the move, to find restaurants (including pictures and prices), local transportation schedule, ATM machines and much more.

App name	Travel	Communication	Finance	Photography
BestTravel	70%	15%	10%	5%

Fig. 7.3 Example of app description and topic analysis result.

Static Analysis

Sensitive information flows in the trusted apps are extracted using static taint analysis. Taint analysis is an instance of flow-analysis technique, which tags program data with labels that describe its provenance and propagates these tags through control and data dependencies. A different label is used for each distinct source of data. Tags are propagated from the operand(s) in the right-hand side of an assignment (uses) to the variable assigned in the left-hand side of the assignment (definition). The output of taint analysis is information flows, i.e., what data of which provenances (*sources*) are used/accessed at what program operations, for instance on channels that may leak sensitive information (*sinks*).

Our analysis focuses on the flows of sensitive information into sensitive program operations, i.e., our taint analysis generates tags at sensitive API calls (e.g. GPS and phone contacts) and traces the propagation of tags into API calls that perform sensitive operations such as, sending messages and Bluetooth packets. These sensitive APIs usually belong to dangerous permission group and hence, the APIs that we analyze are those privileged APIs that require to be specifically authorized by the end user. Sources and sinks are the privileged APIs available from PScout [10]. The APIs that we analyze also include those APIs that enable Inter Process Communication (IPC) mechanism of Android because they can be used to exchange data among apps installed on the same device.

As a result, our taint analysis generates a list of ($source \rightarrow sink$) pairs, where each pair represents the flow of sensitive data originating from a source into a sink.

APIs (both for sources and sinks) are grouped according to the special *permission* required to invoke them. For example, all the network related sink functions, such as `openConnection()`, `connect()` and `getContent()` are all modeled as *Internet* sinks, because they all require the `INTERNET` permission to be executed.

Figure 7.4 shows the static taint analysis result on the "BestTravel" running example app from Figure 7.3. It generates two ($source \rightarrow sink$) pairs that correspond to two sensitive information flows. In the first flow, data read from the GPS is propagated

through the program until it reaches a statement where it is sent over the network. In the second flow, data from the phone contacts is used to compose a text message.

App: BestTravel		
GPS	→	Internet
Contacts	→	SMS

Fig. 7.4 Example of static analysis result.

Our tool, *AnFlo*, runs on compiled byte-code of apps to perform the above static taint analysis. It relies on two existing tools — IC3 [70] and IccTA [56]. As Android apps are usually composed of several components, to precisely extract inter-component information flows, we need to analyze the links among components. *AnFlo* uses IC3 to resolve the target components when a flow spans over multiple components. IC3 uses a solver to infer all possible values of complex objects in an inter-procedural, flow- and context-sensitive manner. Once inter-component links are inferred, *AnFlo* uses an inter-component data-flow analysis tool called IccTA to perform static taint analysis. We customized IccTA to produce flows in a format as presented in Figure 7.4 and paths in a more verbose format to facilitate manual checks.

7.2.3 Model Inference

When results of topic analysis and of static analysis are available for all the trusted apps, they are used to build the *Sensitive Information Flow Model*. Such a model is a matrix with sensitive information sources in its rows and sinks in its columns, as shown in Figure 7.5.

Firstly, apps with the same dominant topic are grouped together to build a sensitive information flow model corresponding to that specific topic. Each group is labeled with the dominant topic. Next, each cell of the matrix is filled with a number, representing the number of apps in this group having the corresponding ($source \rightarrow sink$) pair.

Figure 7.5 shows a sample sensitive information model regarding the topic "Travel". There are 36 distinct flows in the apps grouped under this dominant topic. The matrix shows that there are ten apps containing *GPS position* flowing through the Internet (one of them being the *BestTravel* app, see Figure 7.4); eight apps through text messages and three apps through Bluetooth. Similarly, the matrix shows that *Contacts* information flows through SMS in seven apps and through Bluetooth in eight apps.

From this model, we can observe that for Travel apps it is quite *common* to share the position of the end user via Internet and SMS. However, it is quite *uncommon* to

Topic: "Travel"	**Sinks**		
Sources	Internet	SMS	Bluetooth
GPS	10	8	3
Contacts	0	7	8

Fig. 7.5 Example of sensitive information flow model.

share position data via Bluetooth, because it happened only in three cases. Likewise, the phone contacts are *commonly* shared only using text messages and Bluetooth but not through Internet.

To provide a formal and operative definition of *common* and *uncommon* flows, we compute a threshold denoted as τ. Flows that occur more than or equal to τ are considered as *common*; flows that never occur or that occur fewer than τ are considered as *uncommon* regarding this topic.

Although our model assumes or trusts that the *trusted* apps are benign and correct, it is possible that some of them may contain defects, vulnerabilities or malware. This problem is addressed by classifying those flows occurring less than the threshold τ as uncommon, i.e., our approach tolerates the presence of some anomalous flows in the reference model since these flows would still be regarded as uncommon. Hence, our approach works as long as the *majority* of the trusted apps are truly trustworthy.

To compute this threshold, we adopt the box-plot approach proposed by Laurikkala et al. [54], considering only flows occurring in the model, i.e., we consider only values greater than zero. τ is computed in the same way as drawing outlier dots in boxplots. It is the lower quartile (25th percentile) minus the step, where the step is 1.5 times the difference between the upper quartile (75th percentile) and the lower quartile (25th percentile). It should be noted that τ is not trivially the lower quartile; otherwise 25% of the apps would be *outliers* by construction. The threshold is lower, i.e., it is the lower quartile minus the step. Therefore, there is no fixed amount of outliers. Outliers could be few or many depending on the distribution of data. Outliers would only be those cases that are really different from the majority of the training data points.

In the example regarding topic "Travel" in Figure 7.5, the threshold is computed considering only the five values that are > 0. The value for the threshold is $\tau_{Travel} = 7$. It means that GPS data sent through Internet (GPS $\rightarrow$ Internet) or text messages (GPS $\rightarrow$ SMS) are *common* for traveling apps. Conversely, even though there are three trusted apps which send GPS data through Bluetooth (GPS $\rightarrow$ Bluetooth), there are too few cases to be considered common, and this sensitive information flow will be considered *uncommon* in the model. Likewise, phone contacts are *commonly* sent

through text messages and Bluetooth, but it is *uncommon* for them to be sent through the Internet, since this *never* occurs in the trusted apps.

7.2.4 Classification

After the Sensitive Information Flow Models are built on trusted apps, they can be used to classify a new AUT. First of all, features must be extracted from the AUT. The features are the topics associated with the app description and the sensitive information flows in the app. As in Section 7.2.2, first data pre-processing is performed on the app description of the AUT. Then, topics and their probabilities are inferred from the pre-processed description using the Mallet tool. Among all the topics, we consider only the *dominant topic*, the one with the highest probability, because it is the topic that most characterizes this app. We then obtain the Sensitive Information Flow Model associated with this dominant topic.

To ensure the availability of the Sensitive Information Flow Model, the Mallet tool is configured with the list of topics for which the Models are already built on the trusted apps. And given an app description, the Mallet tool only generates topics from this list. The more diverse trusted apps we analyze, the more complete list of models we expect to build.

For example, Figure 7.6(a) shows the topics inferred from the description of a sample AUT "TripOrganizer". The topic "Travel" is highlighted in bold to denote that it is the dominant topic.

Next, sensitive information flows in the AUT are extracted as described in Section 7.2.2. The extracted flows are then compared against the flows in the model associated with the dominant topic. If the AUT contains only flows that are *common* according to the model, the app is considered consistent with the model. If the app contains a flow that is not present in the model or a flow that is present but is *uncommon* according to the model, the flow (and thus the app) is classified as anomalous. Anomalous flows require further manual inspection by a security analyst, because they could be due to defects, vulnerabilities, or malicious intentions.

For example, Figure 7.6(b) shows three sensitive information flows extracted from a "TripOrganizer" app. Since the dominant topic for this app is "Travel", these flows can be checked against the model associated with this topic shown in Figure 7.5. Regarding this model, earlier, we computed that the threshold is $\tau_{Travel} = 7$ and the flow (Contacts $\rightarrow$ SMS) is common (see Section 7.2.3). Therefore, flow 1 observed in "TripOrganizer" (Figure 7.6(b)) is consistent with the model. However, flow 2 (Contacts $\rightarrow$ Internet) and flow 3 (GPS $\rightarrow$ Bluetooth), highlighted in bold in Figure 7.6(b), are

uncommon according to the model. As a result, the AUT "TripOrganizer" is classified as *anomalous*.

App name	**Travel**	Books	Tools	Game
TripOrganizer	**78%**	11%	4%	7%

(a) Topics classification

App: TripOrganizer			
1:	Contacts	→	SMS
2:	**Contacts**	→	**Internet**
3:	**GPS**	→	**Bluetooth**

(b) Sensitive information flows

Fig. 7.6 Classification of the app under analysis.

7.3 Empirical Assessment

In this section, we evaluate the usefulness of our approach and report the results. With the main goal being able to detect anomalous information flows in Android apps (whether benign, vulnerability or malicious), we would like to investigate how the reported anomalous flows can be useful in, for example, identifying vulnerabilities or in spotting malicious behaviors. We assess our approach by answering the following research questions:

- $\mathbf{RQ}_{7\_Vul}$: Is *AnFlo* useful for identifying *vulnerable apps* containing anomalous information flows?
- $\mathbf{RQ}_{7\_Time}$: How *long* does *AnFlo* take to classify apps?
- $\mathbf{RQ}_{7\_Topics}$: Is the *topic* feature really needed to detect anomalous flows?
- $\mathbf{RQ}_{7\_Cat}$: Can *Google Play store categories* be used instead of *topics* to learn an accurate Sensitive Information Flow Model?
- $\mathbf{RQ}_{7\_Mal}$: Is *AnFlo* useful for identifying *malicious* apps?

The first research question $\mathbf{RQ}_{7\_Vul}$ investigates the result of *AnFlo*, whether it is useful for detecting anomalies in vulnerable apps that, for example, may leak sensitive information. $\mathbf{RQ}_{7\_Time}$ investigates the cost of using our approach in terms of the

time taken to analyze a given AUT. A short analysis time is essential for tool adoption in a real production environment.

Then, in the next two research questions, we investigate the role of *topics* as a feature for building the Sensitive Information Flow Models. In particular, $\mathbf{RQ}_{7\_Topics}$ investigates the absolute contribution of topics, by learning the Sensitive Information Flow Model without considering the topics and by comparing its performance with that of our original model. To answer $\mathbf{RQ}_{7\_Cat}$, we replace topics with the *categories* defined in the official market, and we compare the performance of this new model with that of our original model.

Finally, the last research question $\mathbf{RQ}_{7\_Mal}$ investigates the usefulness of *AnFlo* in detecting malware based on anomalies in sensitive information flows.

7.3.1 Benchmarks and Experimental Settings

Trusted Apps

AnFlo needs a set of trusted apps to learn what is the *normal* behavior for "correct and benign" apps. We defined the following guidelines to collect trusted apps: (i) apps that come from the official Google Play Store so they are scrutinized and checked by the store maintainer, and (ii) apps that are very popular so they are widely used and reviewed by a large community of end-users and programming mistakes are quickly notified and patched.

In this experiment, we used the same trusted apps dataset that is used in Chapter 5. From each category (a total of 30 categories), we downloaded[3], the top 500 apps together with their descriptions. After cleaning apps with non-English and short descriptions, we are left with 11,796 apps for building references models.

Additionally, we measured if these apps were actively maintained by looking at the date of the last update. 70% of the apps were last updated in the past 6 months before the Play Store was crawled, while 32% of the apps were last updated within the same month of the crawling. This supports the claim that the trusted apps are well maintained.

The fact that the trusted apps we use are suggested and endorsed by the official store, and that they collected good end-user feedback allows us to assume that the apps are of high quality. Nevertheless, as explained in Section 7.2.3, our approach is robust against the inclusion of a small number of anomalous apps in the training set since we adopt a threshold to classify anomalous information flows.

[3]Collected during a period of time in 2016

Subject Benign Apps

AnFlo works on compiled apps and, therefore the availability of source code is not a requirement for the analysis. However, for this experiment sake, we opted for open source projects, which enable us to inspect the source code and establish the *ground truth*.

The F-Droid repository[32] represents an ideal setting for our experimentation because (i) it includes real world apps that are also popular in the Google Play Store, and (ii) apps can be downloaded with their source code for manual verification of the vulnerability reports delivered by *AnFlo*.

The F-Droid repository was crawled in July 2017 for apps that meet our criteria. Among all the apps available in this repository, we used only those apps that are also available in the Google Play Store, whose descriptions meet our selection criteria (i.e., description is in English and it is longer than 10 words). Eventually, our experimental set of benign apps consists of 596 AUAs.

Subject Malicious Apps

To investigate if *AnFlo* can identify malware, we need a set of malicious apps with their *declared* functional descriptions. Malicious apps are usually repackaged versions of popular (benign) apps, injected with malicious code (aka Trojanized); hence the descriptions of those popular apps they disguise as can be considered as their app descriptions. Hence, by identifying the original versions of these malicious apps in the Google Play Store, we obtain their declared functional descriptions.

We consider the malicious apps from the Drebin malware dataset [7], which consists of 5,560 samples that have been collected in the period of August 2010 to October 2012. We randomly sampled 560 apps from this dataset. For each malicious app, we performed static analysis to extract the *package name*, an identifier used by Android and by the official store to distinguish Android apps[4]. We queried the official Google Play store for the original apps, by searching for those having the same package name. Among our sampled repackaged malicious apps, we found 20 of the apps in the official market with the same package name. We analyzed their descriptions and found that only 18 of them have English descriptions. We therefore performed static taint analysis on these 18 malware samples, for which we found their "host" apps in the official market. Our static analysis crashed on 6 cases. Therefore, our experimental set of malicious apps consists of 12 AUAs.

[4]Even if it is easy to obfuscate this piece of information, in our experiment some apps did not rename their package name

7.3.2 Results

Detecting Vulnerable Apps

First of all, *AnFlo* was used to perform static taint analysis on the 11,796 trusted apps and topic analysis on their descriptions from the official Play Store. These apps are used as training set. It then learns the Sensitive Information Flow Models based on the dominant topics and extracted flows as described in Section 7.2.3. Then, the AUAs from the F-Droid repository (Section 7.3.1) have been classified based on the Sensitive Information Flow Models.

Out of 596 AUAs, static taint analysis reported 76 apps to contain flows of sensitive information that reach sinks, for a total of 1428 flows. These flows map to 147 distinct source-sink pairs.

App	Topic	Source	Sink
a2dp.Vol	communications utility	Bluetooth Bluetooth Bluetooth admin Broadcast sticky Modify audio settings	IPC Modify audio settings Modify audio settingss Modify audio settingss Modify audio settingss
com.alfray.timeriffic	tools and utility	IPC	Write settingss
com.dozingcatsoftware.asciicam	photo editors	Camera	IPCs
com.matoski.adbm	**utility**	**Access WiFi state**	**IPC**
com.mschlauch.comfortreader	**books & readers**	**IPC**	**Internet**
com.newsblur	productivity	IPC	Access network states
com.Pau.ImapNotes2	documents manager	Authenticate accounts Authenticate accounts Get accounts Get accounts Get accounts	IPCs Authenticate accountss Authenticate accountss Get accountss Manage accountss
com.thibaudperso.sonyCamera	tools & utility	IPC NFC	Change WiFi states NFCs
fr.neamar.kiss	security tools	IPC	Access WiFi states
fr.ybo.transportsrennes	utility	Access coarse location Access fine location	Internets Internets
mobi.boilr.boilr	financial	IPC Wake lock	Wake lock Wake lock
org.smc.inputmethod.indic	languages learning	IPC	Vibrate
pw.thedrhax.mosmetro	books & readers	Access WiFi state	Change WiFi state
se.anyro.Nfc_reader	communications utility	IPC	NFC

Total subject apps: 596 **Total anomalous apps: 14** **Vulnerable apps: 2** **Not vulnerable apps: 12**

Table 7.1 Anomaly detection based on Sensitive Information Flows Model.

Out of these 76 apps, 14 AUAs are classified as *anomalous*. Table 7.1 shows the analysis results reported by *AnFlo*. The first column presents the name of the app. The second column presents the app's dominant topic. The third and fourth columns present the source of sensitive data and the sink identified by static taint analysis, respectively. As shown in Table 7.1, in total *AnFlo* reported 25 anomalous flows in these apps. We manually inspected the source code available from the repository to determine if these anomalous flows were due to programming defects (vulnerabilities). Two apps are found to be vulnerable (highlighted in boldface in Table 7.1), they are *com.matoski.adbm* and *com.mschlauch.comfortreader*.

com.matoski.adbm is a utility app for managing the ADB debugging interface. The anomalous flow involves data from the WiFi configuration that leak to other apps through the Inter Process Communication. Among other information that may leak, the SSID data, which identifies the network to which the device is connected to, can be used to infer the user position and threaten the end user privacy. Hence, this programming defect leads to information leakage vulnerability that requires corrective maintenance. We reported this vulnerability to the app owners on their issue tracker.

com.mschlauch.comfortreader is a book reader app, with an anomalous flow of data from IPC to the Internet. Manual inspection revealed that this anomalous flow results from a permission re-delegation vulnerability because data coming from another app is used, without sanitization, for opening a data stream. If a malicious app that does not have the permission to use the Internet passes a URL that contains sensitive privacy data (e.g., GPS coordinates), then the app could be used as proxy to leak information. We also reported this vulnerability to the app developers.

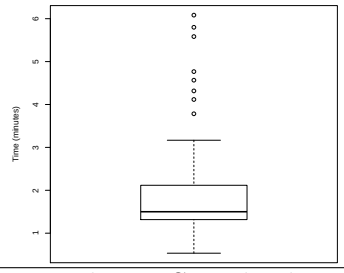
Regarding the other 12 AUAs, even though they contain anomalous flows compared to trusted apps, manual inspection revealed that they are neither defective nor vulnerable. For example, some apps contain anomalous flows that involve IPC. Since data may come from other apps via IPC (source) or may flow to other apps via IPC (sink), such flows are considered dangerous in general. However, in these 12 apps, when IPC is a source (e.g., in `com.alfray.timeriffic`), data is either validated/sanitized before use in the sink or is used in a way that do not threaten security. On the other hand, when IPC is a sink (e.g., in `com.dozingcatsoftware.asciicam`), the destination is always a component in the same app, so the flows are not actually dangerous.

Since *AnFlo* helped us detect 2 vulnerable apps containing anomalous information flows, we can answer $\mathbf{RQ}_{7\_Vul}$ by stating that *AnFlo* is useful for identifying vulnerabilities related to anomalous information flows.

Classification Time

To investigate $\mathbf{RQ}_{7\_Time}$, we analyze the time required to classify the AUAs. We instrumented the analysis script with the Linux *date* utility to log the time (in seconds) before starting the analysis and at its conclusion. Their difference is the amount of time spent in the computation. The experiment was run on a multi-core cluster, specifically designed to let a process run without sharing memory or computing resources with other processes. Thus, we assume that the time measurement is reliable.

Classification time includes the static analysis step to exact data flow, the natural language step to extract topics from description and the comparison with the Sensitive Information Flow Model to check for consistency. Figure 7.7 reports the boxplot of the time (in minutes) needed to classify the F-Droid apps and the descriptive statistics. On average, an app takes 1.9 minutes to complete the classification and most of the analyses concluded in less than 3 minutes (median = 1.5). Only a few (outliers) cases take longer analysis time.



Mean	Median	Standard Deviation
1.9	1.5	1.07

Fig. 7.7 Boxplot of classification time.

Topics from App Description

We now run another experiment to verify our claim that topics are important features to build an accurate mode ($\mathbf{RQ}_{7\_Topics}$). We repeated the same experiment as before, but using only flows as features and without considering topics, to check how much detection accuracy we lose in this way.

We still consider all the trusted apps for learning the reference model, but we only use static analysis data. That is, we do not create a separate matrix for each topic; instead we create one single big matrix with sources and sinks for all the apps. This Sensitive Information Flow Model is then used to classify F-Droid apps and the results are shown in Table 7.2. As we can see, only four apps are detected as *anomalous* by

App	Topic	Source	Sink
a2dp.Vol	Communications utility	Bluetooth	Modify audio settings
		Bluetooth admin	Modify audio settings
		Broadcast sticky	Modify audio settings
		Modify audio settings	Modify audio settings
com.Pau.ImapNotes2	Document manager	Authenticate accounts	IPC
		Authenticate accounts	Authenticate accounts
		Get accounts	Authenticate accounts
		Get accounts	Get accounts
		Get accounts	Manage accounts
com.thibaudperso.sonyCamera	Utility	IPC	Change wifi state
		NFC	NFC
se.anyro.Nfc_reader	Communications utility	IPC	NFC

Total subject apps: 596 Total anomalous apps: 4 Vulnerable apps: 0 Not vulnerable apps: 4

Table 7.2 Anomaly detection using only information flows as a feature (no topic feature).

this second approach, and all of them were already detected by our original approach. Manual inspection revealed that all of them are not vulnerable.

This suggests that topic is a very important feature to learn reference models in order to detect a larger amount of anomalous apps. In fact, when topics are not considered and all the apps are grouped together regardless of their topics, we observe a *smoothing* effect. Differences among apps become less relevant to detect anomalies. While in the previous model, an app was compared only against those apps grouped under the same topic. Here, an app is compared to all the trusted apps. Without topic as a feature, our model loses the ability to capture the characteristics of distinct groups and, thus, the ability to detect deviations from them.

Play Store Categories

To investigate $\mathbf{RQ}_{7\_Cat}$, instead of grouping trusted apps based on topics, we group them according to their app categories as determined by the official Google Play Store. First of all we split trusted apps into groups based on the market category they belong to[5]. We then use static analysis information about flows to build a separate source-sink matrix per each category. Eventually we compute thresholds to complete the model.

We then classify each AUT from F-Droid by comparing it with the model of the corresponding market category. The classification results are reported in Table 7.3. Ten apps are reported as containing anomalous flows and most of them were also detected by our original approach (Table 7.1). Two apps reported by this approach were not reported by our proposed approach, which are *com.angrydoughnuts.android.alarmclock* and *com.futurice.android.reservator*. However, they are neither the cases of vulner-

[5]At the time of the crawling, in Google Play Store, a popular app was assigned to only one category.

App	Category	Source	Sink
a2dp.Vol	Transportation	Bluetooth Bluetooth Bluetooth Bluetooth admin Broadcast sticky Modify audio settings	IPC Bluetooth Modify audio settings Modify audio settings Modify audio settings Modify audio settings
com.Pau.ImapNotes2	Productivity	Authenticate accounts Authenticate accounts Get accounts Get accounts Get accounts	IPC Authenticate accounts Authenticate accounts Get accounts Manage accounts
com.alfray.timeriffic	Tools	IPC	Write settings
com.angrydoughnuts. android.alarmclock	Tools	Wake lock	Vibrate
com.dozingcatsoftware.asciicam	Photography	Camera	IPC
com.futurice.android.reservator	Business	Get accounts	IPC
com.matoski.adbm	**Tools**	**Access wifi state**	**IPC**
com.thibaudperso.sonyCamera	Media and video	IPC NFC	Change wifi state NFC
mobi.boilr.boilr	Finance	Wake lock	Wake lock
se.anyro.Nfc_reader	Communication	IPC	NFC
Total subject apps: 596	**Total anomalous apps: 10**	**Vuln. apps: 1**	**Not vuln. apps: 9**

Table 7.3 Anomaly detection using Google Play categories as a feature instead of topics.

abilities nor malicious behaviors. Only one flow detected by this approach is a case of vulnerability, namely *com.matoski.adbm*, highlighted in boldface, which was also detected by our proposed approach. Hence, this result supports our design decision of using topics.

Comparison of the Models

RQ	Features	False positives	Unique flows	True Positive
RQ_{7_Vul}	Flows + Topics	12	5	2
RQ_{7_Topics}	Flows	4	0	0
RQ_{7_Cat}	Flows + Market categories	9	2	1

Table 7.4 Summary of model comparison result.

Table 7.4 summarizes the result of the models comparison. The first model (first row) considers both data flows and description topics as features. Even though this

approach reported the largest number of false positives (12 apps, second column), we were able to detect 2 vulnerabilities (last column) by tracing the anomalies reported by this approach. It also detected 5 additional anomalous apps that other approaches did not detect ('Unique' column).

The second model (second row) considers only data flows as a feature. Even though the number of false positives drops to 4, we were not able to detect any vulnerability by tracing the anomalies reported by this approach. This result suggests that modeling only flows is not enough for detecting vulnerabilities.

When market categories are used instead of description topics (last row), the false positives drops to 9 (25% less compared to our proposed model). It detected 2 additional anomalous apps that other approaches did not detect ('Unique' column). Tracing the anomalies reported by this approach, we detected only one out of the two vulnerabilities that we detected using our proposed approach. This result suggests that topics are more useful than categories for detecting vulnerable apps containing anomalous information flows.

Detecting Malicious Apps

Anomalies in the flow of sensitive data could be due to malicious behaviors as well. The goal of this last experiment is to investigate whether *AnFlo* can be used to identify malware ($\mathbf{RQ}_{7\_Mal}$). To this aim, we use the Sensitive Information Flow Model (learned on the trusted apps) to classify the 18 AUAs from the Drebin malware dataset. Data flow features are extracted using static analysis from these malicious apps. However, static taint analysis crashed on 6 apps because of their heavy obfuscation. Since improving the static taint analysis implementation to work on heavy obfuscated code is out of the scope of this approach, we run the experiment on the remaining 12 apps. Topics are extracted from the descriptions of the original versions of those malware, which are available at the official market store.

The malicious apps have been subject to anomaly detection, based on the three distinct feature sets: (i) flows and topics; (ii) only flows; and (iii) flows and market categories. The classification results are shown in Table 7.5. The first column reports the malware name (according to ESET-NOD32[31] antivirus) and the second column contains the name of the original app that was repackaged to spread the malware. The remaining three columns report the results of malware detection by the three models based on different sets of features: a tick mark ("✔") means that the model correctly detected the app as anomalous, while a cross ("✖") means no anomaly detected.

Malware Name	Repackaged App Name	Flows + Topics	Flows only	Flows + Market cat.
PJApps	com.appspot.swisscodemonkeys.steam	✔	✔	✔
DroidKungFu (v.1)	com.gp.jaro (variant)	✔	✔	✔
DroidKungFu (v.2)	com.gp.jaro (variant)	✔	✖	✔
Spy.GoldDream	com.rechild.advancedtaskkiller	✔	✔	✔
Anserver	com.sohu.blog.lzn1007.WatermelonProber	✔	✖	✔
TrojanSMS.Agent	org.baole.app.blacklistpro	✔	✔	✔
Plankton.I	com.anndconsulting.mancala	✖	✖	✖
PJApps.B	com.estrongs.android.pop	✖	✖	✖
Spy.Geinimi.E	com.littlekillerz.legendsarcana	✖	✖	✖
Spy.Geinimi.D	com.magicwach.rdefense	✖	✖	✖
Plankton.A	com.wangsong.costwatcher	✖	✖	✖
PJApps.I	es.cesar.quitesleep	✖	✖	✖

Total malware apps: 12 Total anomalous malware detected: 6
Missed malware (not anomalous): 6

Table 7.5 Results on malicious apps.

While the model based on topics and the model based on market categories classified the same 6 AUAs as malicious, the model based on only flows classified only 4 AUAs as malicious.

All the malware except `TrojanSMS.Agent` are the cases of privacy sensitive information leaks such as device ID, phone number, e-mail or GPS coordinate, being sent over the network or via SMS. One typical malicious behavior is observed in `Spy.GoldDream`. In this case, after querying the list of installed packages (sensitive data source), the malware attempts to kill selected background processes (sensitive sink). This is a typical malicious behavior observed in malware that tries to avoid detection by stopping security products such as antiviruses. Botnet behavior is observed in `DroidKunFu`. A command and control (C&C) server command is consulted (sensitive source) before performing privileged actions on the device (sensitive sink).

As shown in Table 7.5, when only static analysis features are used in the model, two malicious apps are missed. This is because this limited model compares the given AUT against all the trusted apps, instead of only these apps from a specific subset (grouped by the common topic or the same category). A flow that would have been anomalous for the specific topic (or the specific category) might be normal for another topic/category. For example, acquiring GPS coordinates and sending them over the

network is common for navigation or transportation apps. However, it is not a common behavior for tools apps, which is the case of the `Anserver` malware.

The remaining 6 apps in the dataset were consistently classified as not-anomalous by all the models. These false negatives are mainly due to the malicious behaviors being not related to sensitive information flows, such as dialing calls in the background or blocking messages. Another reason is due to obfuscation used by malware to hide the sensitive information flows. Static analysis inherently cannot handle very resilient variants of code obfuscation.

7.3.3 Limitation and Discussion

In the following, we discuss some of the limitations of our approach and of its experimental validation. The most prominent limitation to adopt our approach is the availability of trusted apps to build the model of sensitive information flows. In our experimental validation, we trusted top ranked popular apps from the official app store, but we have no guarantee that they are all immune to vulnerabilities and to malware content. However, as explained in Section 7.2.3, our approach is quite robust with respect to the inclusion of a small number of defective, vulnerable, or malicious apps in the training set, as long as the majority of the training apps are benign and correct. This is because we use a threshold-based approach that models flows that are common to a large set of apps. Thus, vulnerable flows occurring on few training apps are not learnt as normal in the model and they would be classified as *anomalous* when observed in a given AUT.

A flow classified as *anomalous* by our model needs further manual analysis to check if the anomaly is a vulnerability, is a malicious behavior or is safe. Manual inspection could be an expensive task that might delay the delivery of the software product. However, in our experimental validation, manual filtering on the experimental result took quite short time, on average 30 minutes per app. Considering that the code of the app to review was new to us, we expect a shorter manual filtering phase for a developer who is quite familiar with the code of her/his app. All in all, manual effort required to manual filter results of the automated tool seems to be compatible with the fast time-to-market pressure of smart phone apps.

When building sensitive information flow models, we also considered grouping of apps by using clustering technique based on the topics distribution, instead of grouping based on the dominant topic alone. But we conducted preliminary experiments using this method and observed that grouping of apps based on dominant topics produce more cohesive groups, i.e., apps that are more similar.

Inherently, it is difficult for static analysis-based approaches including ours to handle obfuscated code. Therefore, if training apps are obfuscated (e.g., to limit reverse engineering attacks), our approach may collect incomplete static information and only build a partial model. Additionally, if the AUT is obfuscated, our approach may not detect anomalies. Incorporating our approach with dynamic analysis might provide a better way to address obfuscation limitation.

7.4 Chapter Summary

In this chapter, we proposed a novel approach to analyze and classify the flows of sensitive information in Android apps. In our approach, trusted apps are first analyzed to extract topics from their descriptions and data flows from their code. Topics and flows are then used to learn Sensitive Information Flow models. We can use these models for analyzing new Android apps to determine whether they contain anomalous information flows. Our experiments show that this approach could detect anomalous flows in vulnerable and malicious apps quite fast.

CHAPTER 8

Related Work

The work presented in this thesis is related to research works that deal with permission re-delegation problems or information leaks (closely-related vulnerabilities) on smart phones. In this chapter, we summarize related works by topics.

8.1 Permission re-delegation

The most related work regarding the threat model is the work by Felt et al. [33], which first introduced the permission re-delegation problem and proposed an approach for detecting it. Chin et al. [21], Lu et al. [59], and Zhong et al. [98] also presented similar approaches to detect permission re-delegation vulnerabilities. These approaches identify all possible entry points of an app and then perform data-flow analysis starting from an entry point until a privileged API is reached. A permission re-delegation vulnerability is reported whenever there exists a path from a public entry-point to a privileged API call. However, Felt et al. and Chin et al. acknowledged that their approaches detect permission re-delegation cases but cannot distinguish between intended cases and vulnerable ones, and thus, possibly produce false alarms.

We present an extension of this threat model by imposing additional preconditions. We detect permission re-delegation vulnerability in an app only when either:

1. The permission re-delegation occured while processing data that should instead be rejected because it is inconsistent with an intent-filter (Chapter 4)

2. The permission re-delegation in the app is anomalous with respect to permission re-delegations observed among similar apps (Chapter 5)

3. The permission re-delegation consists in acting as a proxy to send a privileged Intent to a system app (second-order) (Chapter 6)

4. The information flow leading to permission re-delegation is anomalous with respect to information flows observed on other apps (Chapter 7)

Empirical evidence suggests that our approaches perform better (in terms of precision) either without producing false alarms or fewer false alarms than state-of-the-art tools and techniques.

Zhang et al. [97] proposed Appsealer, a runtime patch to mitigate permission re-delegation problem. It performs static data-flow analysis to determine sensitive data-flows from sources to sinks and apply the patch before the invocation of a privileged API such that the app alerts the user of a potential permission re-delegation attack and requests the user's authorization to continue. This is an alternative way of distinguishing legitimate and anomalous permission re-delegation by relying on the user. Lee et al. [55] proposed Sealant, which is similar to Appsealer; it additionally extends the Android framework to monitor vulnerable inter-app communication at run-time and prevents attacks. By contrast, our approaches apply different techniques to infer abnormal permission re-delegations automatically without relying on the end user.

Roppdroid [24] mitigates permission re-delegation problems via resource virtualization and modified ICC mechanism. It enhances Android's ICC mechanism by adding ICC provenance so that the system can monitor the permissions of the apps involved in the ICC. Sensitive resources are then virtualized in such a way that privileges are not leaked to the sender app.

Though the taint analysis based approach proposed in Chapter 4 requires a modified version of Android (TaintDroid [29], discussed below) in order to perform dynamic taint analysis, the remainder of the proposed approaches do not require modification of the Android framework unlike Sealant and Roppdroid.

8.2 Data Flow Analysis

Information leaks in smart phone apps is a quite investigated security problem. Information flow in mobile apps is analyzed either statically [8], [91], [41], [72], [70], [87], [65], [56], [53], [10], [51], [13], [93], [17], [58] or dynamically [29], to detect disclosure of

sensitive information. Tainted sources are system calls that access private data (e.g., global position, contacts entries), while sinks are all the possible ways that make data leave the system (e.g., network transmissions). An issue is detected when sensitive information could potentially leave the app through one of the sinks. In the following, we discuss some of these approaches and explain the major differences between our approaches.

Most previous works use static taint analysis to detect (inter-/intra-component) privacy leaks. FlowDroid [8] is an intra-component static analysis tool for Android apps that reasons about information flow at the level of variables, by finding paths from sources to sinks. Droidsafe [41] is a static information flow analysis tool that can detect potential information leaks in Android apps. The authors showed that it performs better than FlowDroid in terms of the detection of actual information flows. Similar to FlowDroid, it can also be adapted to detect permission re-delegation problems. Static analysis part of our taint analysis based approach (in Chapter 4) relies on an adapted version of FlowDroid. IccTA [56] extends FlowDroid to include inter-component communication by modeling the life-cycle and callback methods by instrumenting the code of the app. To do so, it relies on IC3 [70] that uses a COAL solver to infer all possible values of complex objects in an inter-procedural, flow- and context-sensitive manner.

Compared to these approaches, the static analyses used in our anomaly detection and genetic algorithm based approach to detect (first-order) permission re-delegation vulnerability (Chapter 5) and second-order permission re-delegation vulnerability detection (Chapter 6) are different because our approaches are based on call-graph analysis instead of data-flow analysis. This helps us generalize and include cases even when no data is involved in order to exploit the permission re-delegation vulnerability.

Amandroid [91] has been conceived to detect privacy data leaks due to inter-component communication (ICC) in Android apps. Epicc [72] identifies ICC connection points by using inter-procedural data-flow and points-to analysis. IC3 improves Epicc by using complex techniques to resolve targets and values used in ICC. Similar to Epicc, Jitana [87] also analyzes interacting apps. However, instead of combining apps for analysis, it uses static class loader that enables analysis of a large number of interacting apps. Grace et al. [43] perform static analysis in stock Android apps released by different vendors, to check the presence of any information leak. Since vendors modify or introduce their own apps, they might also introduce new vulnerabilities. Their work, however, is limited to stock apps on specific vendor devices. Dexteroid [51] performs

static analysis to reverse engineer life cycle models of Android components and detect privacy data leaks.

Covert [13] uses static analysis and formal verification to verify the *collusive data leaks* and privilege escalation problems that arise from the interaction of multiple apps (known as colluding apps). It infers the security properties from individual apps and reasons about the security of the phone device as a whole; that is, it checks whether apps installed in a device can collude with each other to mount attacks. The technique based on anomaly detection proposed in Chapter 5 differs from Covert because while our proposed approach infers normal and anomalous permission re-delegation (in their case privilege escalation) based on behaviors of similar apps, Covert's formal verification process relies on predefined security policies that describe what is normal and what is anomalous. Though designed for analysis of colluding apps, Covert however does not address Second Order Permission Re-delgation vulnebability, which is presented in Chapter 6. Similar to Covert, AppHolmes [93] also uses static analysis and predefined security policies to detect collusive data leaks. DIALDroid [17] addressed the complexity problem of performing pairwise program analysis of apps to detect collusive data leaks by incorporating optimized, efficient data storage and fast query processing techniques into static analysis of ICC data flows.

In contrast to the above-mentioned approaches, instead of relying on predefined security policies, we either check for unexpected data processing or compare the security properties of the app against its similar apps to detect abnormal/malicious behavior.

To address colluding apps problem, Bugiel et al. [18] proposed a system-centric (rather than application-dependent) solution that conducts policy-driven runtime monitoring of communications between apps at middleware IPC layer and at kernel layer. This requires modification of Android framework.

AAPL [58] performs multiple specialized static analyses such as conditional flow identification and joint flow tracking for more accurate detection of privacy data leaks.

TaintDroid [29] is a tool for performing dynamic taint analysis. It relies on a modified Android installation that tracks tainted data at run-time. The implementation showed minimal size and computational overhead, and was effective in analyzing many real Android apps. A complementary approach is based on static analysis [65], where a type system is implemented to track security levels. A static analyzer applies the type system to byte-code and it detects violations when sensitive information could potentially leave the app through a sink.

8.3 Anomaly Detection based on Similar Apps

There are several previous works [89, 85, 68, 79] that base on grouping of Android apps either for malware detection or obfuscation identification. The approaches used in Mudflow [11] and Chabada [42] are the most closely related to our approaches proposed for detecting anomalous permission re-delegation (Chapter 5) and anomalous information flows (Chapter 7). Mudflow [11] is a tool for malware detection based on sensitive information flow. Similar to our approach, it relies on static taint analysis to detect flows of sensitive data towards sensitive APIs. But the main difference is that Mudflow learns the normal data flow patterns from all available benign apps and compares the patterns of the given app under test against all the learnt patterns, whereas our approach compares them only based on similar apps. In addition, Mudflow only applies static data flow analysis, whereas our anomalous permission re-delegation approach (Chapter 5) additionally applies dynamic analysis to reduce false alarms. Lastly, while Mudflow applies intra-component static analysis, for anomalous information flow detection (Chapter 7), we use inter-component analysis in order to covering flows across components.

Chabada [42] detects anomalous apps based on their app descriptions. While we apply similar techniques in terms of natural language processing of apps descriptions and clustering of apps, our goals differ. The goal of their approach is to find anomalous apps among the apps in the wild based on the inconsistencies between the advertised apps descriptions and their actual behaviors. By contrast, our approaches specifically target at accurately identifying permission re-delegation vulnerabilities (Chapter 5) and identifying anomalous information flows (Chapter 7) in a given app under test. More specifically, there are two main differences to achieve these different goals. First, Chabada only identifies calls to privileged APIs (call-sites) using static analysis as features to classify benign and anomalous apps. Instead, we consider execution paths from public entry points to privileged APIs, to analyze privileged API calls related to permission re-delegation and data flows from sensitive sources to sensitive sinks. Second, we perform automated test case generation to verify the results of static analysis and to generate a proof-of-concept attack to confirm and document vulnerabilities.

AAPL [58] also employs a technique called "peer voting" to filter out legitimate privacy leaks and reduce false alarms. The results remaining after filtering are then notified to end-users. Like our approach, AAPL's peer voting mechanism is also inspired by the fact that applications with the same or similar functionality should exhibit similar privacy-related behaviors. However, the main difference is that AAPL

gathers "similar" apps by leveraging Google's app recommendation system, which is based on users' practice — "Users Also Viewed" and "Users Also Installed". Hence, the apps returned may not actually be functionally related to the target app. Even though, AAPL additionally employs NLP to perform semantic similarity analysis and filter out noisy apps, it is limited only to the apps returned by Google's recommendation system.

8.4 Security Test Case Generation

Among the first test case generation techniques for Android apps, we can find those related to Graphical User Interface (GUI) [61, 82, 12, 50, 69, 4, 57, 5, 6, 60]. GUI based app testing approaches have been proposed to generate events and event sequences that make the apps crash or to identify abnormalities of apps. Hu et al. [50] presented an approach for testing Android apps' GUI. Random graphical events are generated and patterns are used to detect bugs in the system log, such as app crashes, type exceptions and violations in the activity state machine. Amalfitano et al. developed AndroidRipper [6] and A2T2 [4] to test Android GUI. These tools dynamically analyse the apps to get a list of fireable events in the GUI widgets, they then generate sequences of graphical events and sensor events. Code is instrumented to record crashes and to eventually translate event sequences into JUnit test cases. Mahmood et al. [62] generates test cases for Android apps using random graphical events and input values. The main difference between all these approaches and ours is that they generate test cases mainly to detect abnormalities in apps such as crashes, exceptions, and violations of access permissions; our approach, instead, generates security test cases that expose permission re-delegation vulnerabilities in Android apps.

Mahmood et al. then introduced EvoDroid [63], the first search-based framework for Android testing. EvoDroid uses a model generated based on static data collected from manifest file and layout XMLs to guide the search process. Sapienz [66] is a multi-objective search based testing tool for Android apps. The authors combined random fuzzing, systematic and search-based exploration. Even though both EvoDroid and Sapienz apply search-based techniques to generate inputs for Android testing, their technique is aimed at GUI testing. In our case, however, we use a search-based technique to generate Intent messages (inter-app communication messages) that can be used to trigger Activities, Services and Broadcast Receivers.

The other works address inter-component (inter-app) communication inputs [95, 80, 94, 47, 20, 74, 9, 96, 64]. These tools randomly generate ICC inputs (Intents) focusing mainly in app crashes. In our case, however, we generate Intents that reveal permission

re-delegation vulnerabilities. JarJarBinks tool [64] analyses inter-app messaging with the objective of generating executable scenarios, where the inter-app communication is tested to detect app (or system) crashes. That is, it generates invalid Intents (i.e., that violate intent-filters) that make apps crash. Test cases, however, reveal generic app crashes rather than focusing on security defects. FuzzDroid [75] also generates executable scenarios using a evolutionary algorithm-based fuzzing approach, with the aim of covering a given target location (e.g., privileged API call) to expose malicious behaviors of a given app. EvoSuite [34] is a general purpose test generator based on genetic algorithm, which can also be adapted to generate executable scenarios for Android apps. Zhong et al. [98] generate test case specifications based on the static analysis results to complete a security testing approach.

CHAPTER 9

Conclusions and Future Directions

In this chapter, we summarize the contributions of this thesis and discuss about possible future research directions.

9.1 Summary of Contribution

Apps are granted permissions in order to access sensitive resources, such as contact or GPS location. Moreover, apps interact with each other in order to complement each other with features. That is, an app that requires a given task (e.g., taking photo) can request other apps that are capable of doing so. In such a way, an app can reuse complex features from other apps without re-implementing them. However, if implemented insecurely, this integration could lead to permission re-delegation vulnerability, putting the end-user data and privacy at risk. Therefore, developers, market owners and security analysts would benefit from means that allows automatic detection of this vulnerability.

In this thesis, we have implemented different program analysis techniques in order to precisely detect permission re-delegation vulnerabilities in Android apps and automatically generate test cases to document them.

The key challenge for automatic detection of permission re-delegation vulnerability is the difficulty to differentiate between intentional integration and vulnerabilities. To address this issue, we investigated different approaches.

We relied on taint analysis to automatically detect this vulnerability. However, instead of reporting every data-flow path leading to permission re-delegation (as majority would be false positives), we focus on permission re-delegation that occur while processing data that an app is not expecting. With the assumption that if an app is expecting input X but receives, instead, input Y that is different from X and still performs a privileged operation instead of rejecting it, we consider the app to be vulnerable. We implemented our approach both in static and dynamic taint analysis.

This approach has then been extended based on the assumption that similar apps would implement a similar behavior. We collected top apps from Google Play and grouped them together by similarity. We then learnt their permission re-delegation behavior. When testing a new app, we compare the behavior of this app against the similar *top* apps. If the app under test has anomalous behavior regarding permission re-delegation, we consider it vulnerable. Furthermore, we use genetic algorithm to generate test cases that show how it can be exploited.

This threat model has been further extended and we defined Second Order Permission Re-delegation. Instead of performing a privileged task themselves (using the Android framework APIs), apps holding the required permission might request system apps to complete a given task. For example, an app that holds the `CALL_PHONE` permission can request the system Phone app to initiate a phone call. When an app requests another privileged system app to perform a privileged task on behalf of an attacking app, we call the scenario Second Order Permission Re-delegation. We discussed the limitations of current multi-app analysis techniques and proposed an automatic detection approach that also relies on the genetic algorithm based test case generation tool we developed before.

Finally, we investigated how to classify apps based on how they deal with sensitive data. We defined a general information flow classification technique to identify anomalies. This approach relies on the assumption that similar apps would have similar information flow behavior. An outlier is then either a vulnerability or a malicious behavior.

All these approaches have been experimentally validated with thousands of real world apps and have been seen to perform better than state-of-the-art approaches in terms of precision. When issue tracking is available, detected vulnerabilities have been reported to their respective developers.

The different techniques proposed in this thesis could be helpful to different stakeholders. First of all, developers could benefit from an automatic tool that shows potential vulnerabilities in their apps. Automatically generated test cases might help

the developers document, fix and test their apps before releasing them to the public. Market owners, such as Google Play, could also benefit from these approach to filter potentially vulnerable apps before publishing them in the store.

Corporate security analysts who, for example, need to investigate whether apps installed in an employee's device are vulnerable or not before installing company apps could also benefit from these tools.

9.2 Future Research Directions

In this thesis we focused on automatic detection of permission re-delegation vulnerabilities and test case generation for Android apps. It would be interesting to investigate if the techniques proposed in this thesis could be adopted to precisely detect other kinds of vulnerabilities. For example, looking at CVE databases, information leak (aka gaining information) is one of the most common Android vulnerabilities. Current approaches addressing this issue usually produce several false positives as in many cases the 'information leak' is legitimate (e.g., a navigation app sharing GPS location). Precise information leak detection with proof-of-concept test cases could be an interesting area to investigate.

We adopted search based approaches for test case generation. While this approach is proven to be effective, it is, however, limited when input values with complex structure need to be generated. Combining search based approach with symbolic execution (similar to [35]) could be an interesting direction to explore.

The test case generation technique that we proposed for inter-component communication could also be adopted for a different context. For example, it could be combined with GUI input generators such as, Monkey [39] or Dynodroid [60], in order to improve code coverage and error detection rate. Investigating how this integration would improve current approaches could be another interesting area to explore.

Though we implemented our approaches in the Android context, we believe they could also be adopted to other multi-component systems. For example, in *Microservice* architecture, an application is split into multiple components called microservice processes (aka containers). Access to each microservice, however, would require authentication and authorization. Splitting an application into microservices makes the application more vulnerable as the different microservices might need to be exposed to the public (e.g., via network). It would be interesting if the approaches presented in this thesis could be adopted to find vulnerabilities such as capability leaks in microservices.

Another interesting area to explore could be investigation of the usability of the proposed approaches. This could be achieved by performing user study with developers to see if the tools help them discover and fix vulnerabilities. The anomalous information flow detection web application could also be made public to see if the tool is useful to end-users.

References

[1] Al-Subaihin, A. A., Sarro, F., Black, S., Capra, L., Harman, M., Jia, Y., and Zhang, Y. (2016). Clustering mobile apps based on mined textual features. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 38:1–38:10, New York, NY, USA. ACM.

[2] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016a). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA. ACM.

[3] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016b). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA. ACM.

[4] Amalfitano, D., Fasolino, A., and Tramontana, P. (2011). A GUI crawling-based technique for Android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252 –261.

[5] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Imparato, G. (2012a). A toolset for gui testing of android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 650–653. IEEE.

[6] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. (2012b). Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA. ACM.

[7] Arp, D., Spreitzenbarth, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket.

[8] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA. ACM.

[9] Aslam, H. A. S., Zeller, A., and Hammer, C. (2014). *Inter-Application Communication Testing of Android Applications Using Intent Fuzzing.* PhD thesis, Universität des Saarlandes Saarbrücken.

[10] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM.

[11] Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., and Bodden, E. (2015). Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436.

[12] Azim, T. and Neamtiu, I. (2013). Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM.

[13] Bagheri, H., Sadeghi, A., Garcia, J., and Malek, S. (2015). Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, (9):866–886.

[14] Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022.

[15] Borgaonkar, R. (2012). Dirty use of USSD codes in cellular networks. In *Ekoparty Security Conference*.

[16] Borgaonkar, R. (last accessed August 2015). Demo: Dirty use of USSD codes, https://www.youtube.com/watch?v=q2-0b04hphs.

[17] Bosu, A., Liu, F., Yao, D. D., and Wang, G. (2017). Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM.

[18] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. (2012). Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19. Citeseer.

[19] Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90.

[20] Cao, C., Gao, N., Liu, P., and Xiang, J. (2015). Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 361–370. ACM.

[21] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011). Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA. ACM.

[22] Clause, J., Li, W., and Orso, A. (2007). Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA. ACM.

[23] ComputerWorld (2014). Android bug lets apps make rogue phone calls, https://www.computerworld.com/article/2489707/malware-vulnerabilities/android-bug-lets-apps-make-rogue-phone-calls.html.

[24] Dai, T., Li, X., Hassanshahi, B., Yap, R. H., and Liang, Z. (2017). Roppdroid: Robust permission re-delegation prevention in android inter-component communication. *Computers & Security*, 68:98–111.

[25] Demissie, B. F., Ceccato, M., and Shar, L. K. (2018). Anflo: Detecting anomalous sensitive information flows in android apps. In *Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. ACM.

[26] Demissie, B. F., Ghio, D., Ceccato, M., and Avancini, A. (2016). Identifying android inter app communication vulnerabilities using static and dynamic analysis. In *Proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 255–266. ACM.

[27] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38.

[28] Elish, K. O., Yao, D., and Ryder, B. G. (2015). On the need of precise inter-app icc classification for detecting android malware collusions. In *IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*.

[29] Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th Usenix Symposium on Operating Systems Design and Implementation*.

[30] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. (2011). A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA. USENIX Association.

[31] ESET (2017). Eset-nod32 antivirus https://www.eset.com/.

[32] F-Droid (on going repo). Android market, https://f-droid.org/.

[33] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. (2011). Permission re-delegation: Attacks and defenses. In *20th Usenix Security Symposium*.

[34] Fraser, G. and Arcuri, A. (2011). Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM.

[35] Gallingani, D., Gjomemo, R., Venkatakrishnan, V., and Zanero, S. (2015). Practical exploit generation for intent message vulnerabilities in android. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 155–157. ACM.

[36] Gartner (last accessed November 2018). Worldwide sales of smartphones returned to growth in first quarter of 2018, https://www.gartner.com/newsroom/id/3876865.

[37] Google (2016). Compact langauge detector, https://github.com/cld2owners/cld2.

[38] Google (2018a). App security improvement program , https://developer.android.com/google/play/asi.

[39] Google (2018b). The monkey ui android testing tool, http://developer.android.com/tools/help/monkey.html.

[40] Google (2019). Android security tips, https://developer.android.com/training/articles /security-tips#ipc.

[41] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. (2015). Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110.

[42] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. (2014). Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM.

[43] Grace, M. C., Zhou, Y., Wang, Z., and Jiang, X. (2012). Systematic detection of capability leaks in stock Android smartphones. In *NDSS*. The Internet Society.

[44] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.

[45] Harman, M. and O'Hearn, P. (2018). From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation*, SCAM 2018, pages 1–23.

[46] Hattersley, L. (2012). Samsung Galaxy S III secret USSD reset code discovered, http://www.macworld.co.uk/news/apple/samsung-galaxy-s-iii-secret-ussd-reset-code-discovered-3400408/.

[47] Hay, R., Tripp, O., and Pistoia, M. (2015). Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM.

[48] Hodge, V. J. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126.

[49] Holland, J. H. (1975). Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence. *Ann Arbor, MI: University of Michigan Press.*

[50] Hu, C. and Neamtiu, I. (2011). Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA. ACM.

[51] Junaid, M., Liu, D., and Kung, D. (2016). Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *computers & Security*, 59:92–117.

[52] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer.

[53] Klieber, W., Flynn, L., Bhosale, A., Jia, L., and Bauer, L. (2014). Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA. ACM.

[54] Laurikkala, J., Juhola, M., Kentala, E., Lavrac, N., Miksch, S., and Kavsek, B. (2000). Informal identification of outliers in medical data. In *Fifth International Workshop on Intelligent Data Analysis in Medicine and Pharmacology*, volume 1, pages 20–24.

[55] Lee, Y. K., Bang, J. y., Safi, G., Shahbazian, A., Zhao, Y., and Medvidovic, N. (2017). A sealant for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 312–323, Piscataway, NJ, USA. IEEE Press.

[56] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and Mcdaniel, P. (2015). IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, pages 280–291.

[57] Liu, C.-H., Lu, C.-Y., Cheng, S.-J., Chang, K.-Y., Hsiao, Y.-C., and Chu, W.-M. (2014). Capture-replay testing for android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pages 1129–1132. IEEE.

[58] Lu, K., Li, Z., Kemerlis, V. P., Wu, Z., Lu, L., Zheng, C., Qian, Z., Lee, W., and Jiang, G. (2015). Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*.

[59] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. (2012). Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA. ACM.

[60] Machiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM.

[61] Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., and Stavrou, A. (2012a). A whitebox approach for automated security testing of android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 22–28. IEEE press.

[62] Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., and Stavrou, A. (2012b). A whitebox approach for automated security testing of Android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*, pages 22–28.

[63] Mahmood, R., Mirzaei, N., and Malek, S. (2014). 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.

[64] Maji, A., Arshad, F., Bagchi, S., and Rellermeyer, J. (2012). An empirical study of the robustness of inter-component communication in Android. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1 –12.

[65] Mann, C. and Starostin, A. (2012). A framework for static detection of privacy leaks in Android applications. In *27th Symposium on Applied Computing (SAC): Computer Security Track*, pages 1457–1462.

[66] Mao, K., Harman, M., and Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM.

[67] McCallum, A. K. (2002). Mallet: A machine learning for language toolkit.

[68] Milosevic, N., Dehghantanha, A., and Choo, K.-K. R. (2017). Machine learning aided android malware classification. *Computers & Electrical Engineering*, 61:266–274.

[69] Mirzaei, N., Malek, S., Păsăreanu, C. S., Esfahani, N., and Mahmood, R. (2012). Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5.

[70] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. (2015a). Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*.

[71] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. (2015b). Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press.

[72] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. (2013). Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA. USENIX Association.

[73] Porter, M. F. (1997). Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[74] Rasthofer, S., Arzt, S., Triller, S., and Pradel, M. (2017a). Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 300–311. IEEE.

[75] Rasthofer, S., Arzt, S., Triller, S., and Pradel, M. (2017b). Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 300–311. IEEE Press.

[76] Ravitch, T., Creswick, E. R., Tomb, A., Foltzer, A., Elliott, T., and Casburn, L. (2014). Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 4. ACM.

[77] Reeve, D. (2012). Remote ussd attack, http://also.dylanreeve.com/2012/09/remote-ussd-attack/.

[78] Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61. ACM.

[79] Saracino, A., Sgandurra, D., Dini, G., and Martinelli, F. (2018). Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97.

[80] Sasnauskas, R. and Regehr, J. (2014). Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM.

[81] Sharir, M. and Pnueli, A. (1981). *Program Flow Analysis: Theory and Applications*, chapter Two approaches to interprocedural data flow analysis, pages 189–233. Prentice Hall.

[82] Song, H., Ryoo, S., and Kim, J. H. (2011). An integrated test automation framework for testing on heterogeneous mobile platforms. In *2011 First ACIS International Symposium on Software and Network Engineering*, pages 141–145. IEEE.

[83] Soot (2018). Soot - a java optimization framework, https://github.com/sable/soot.

[84] Statista (last accessed November 2018). Android - statistics and facts, https://www.statista.com/topics/876/android/.

[85] Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., and Cavallaro, L. (2017). Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM.

[86] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA. ACM.

[87] Tsutano, Y., Bachala, S., Srisa-an, W., Rothermel, G., and Dinh, J. (2017). An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 324–334, Piscataway, NJ, USA. IEEE Press.

[88] University, S. (2016). Stanford corenlp, http://stanfordnlp.github.io/corenlp/.

[89] Wang, W., Li, Y., Wang, X., Liu, J., and Zhang, X. (2018). Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. *Future Generation Computer Systems*, 78:987–994.

[90] Wegener, J., Baresel, A., and Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854.

[91] Wei, F., Roy, S., Ou, X., and Robby (2014). AmAndroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA. ACM.

[92] Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[93] Xu, M., Ma, Y., Liu, X., Lin, F. X., and Liu, Y. (2017). Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web*, pages 143–152.

[94] Yang, K., Zhuge, J., Wang, Y., Zhou, L., and Duan, H. (2014). Intentfuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 531–536. ACM.

[95] Ye, H., Cheng, S., Zhang, L., and Jiang, F. (2013). Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM.

[96] Zhang, A., He, Y., and Jiang, Y. (2016). Crashfuzzer: Detecting input processing related crash bugs in android applications. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–8. IEEE.

[97] Zhang, M. and Yin, H. (2014). Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications.

[98] Zhong, J., Huang, J., and Liang, B. (2012). Android permission re-delegation detection and test case generation. In *2012 International Conference on Computer Science and Service System*, pages 871–874.