Detecting Vulnerabilities of Broadcast Receivers in Android Applications

by

Di Tian

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Science (MSc)

in

Computer Science

Faculty of Business and Information Technology

University of Ontario Institute of Technology (UOIT)

Oshawa, Ontario, Canada, 2016

© Di Tian 2016

Abstract

As being a representative mobile operating system in the world, Android OS has been part of users' daily life. Unfortunately, the rapid expansion of Android third-markets introduces malware aiming at Android applications at an alarming rate, which poses great threats to its users. Current research about the privacy leakage in Android mostly focuses on Activity, Service and Content Providers. Little attention has been paid to the vulnerability of Broadcast Receiver, which is expected to assist inter-component collaboration and facilitate component reutilization.

In this thesis, we first present a detailed analysis on vulnerabilities of Broadcast Receiver. Then, we design and develop a Broadcast Receiver Vulnerability Detection (BRVD) system that examines such vulnerabilities, using a combination of semantic analysis and taint analysis. Furthermore, we perform experimental evaluation by analyzing 55 applications from Android third-markets using the proposed system; and 132 registered receivers are found with 11 vulnerable receivers being verified.

Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. Xiaodong Lin for the continuous support of my Master study and related research, for his patience, motivation, and immense knowledge. His guidance not only helped me in all the time of research and writing of this thesis, but also gave me a new comprehend about life. I could not have imagined having a better advisor and mentor for my Master study.

Besides my advisor, I would like to thank the rest of my thesis committee, for their insightful comments and encouragement, and also for the time and efforts they have given.

My sincere thanks also goes to my fellow lab-mates in for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last several years. Also I thank all my friends who always inspire me during my study.

Last but not the least, I would like to thank my parents for supporting me spiritually throughout writing this thesis and my life in general.

Contents

1 Introduction **1**

1.1 Background . . . 2
1.2 Objective . . . 3
1.3 Motivation . . . 4
1.4 Methodology . . . 7
1.5 Contributions . . . 7
1.6 Thesis Organization . . . 8

2 Background and Literature Review **10**

2.1 Android Overview . . . 10
2.1.1 Android Framework . . . 10
2.1.2 Application Components . . . 13
2.1.3 The Intent-based Inter-component communication (ICC) . . . 15
2.1.4 Permission Protection Mechanism . . . 16
2.2 Android Threat Models . . . 17
2.3 Analytical Measure . . . 18
2.3.1 Static Analysis . . . 18
2.3.2 Dynamic Analysis . . . 20
2.4 State of the Art . . . 21

3 BroadcastReceiver Analysis **24**

3.1 Semantic Analysis in the BroadcastReceiver Registration . . . 25
3.1.1 Static BroadcastReceiver Registration . . . 26
3.1.2 Dynamic BroadcastReceiver Registration . . . 29
3.2 Taint Analysis in Broadcastreceiver . . . 31
3.2.1 Inter-procedural data-flow Analysis . . . 32
3.2.2 Four Paths (*4P*) Policy . . . 35
3.3 Concluding Remarks . . . 37

4 The BRVD Framework **39**
4.1 Semantic Analysis–SUKE Framework 40
4.1.1 *Abroster* Engine 41
4.1.2 Algorithms in *Abroster* 42
4.2 Taint Analysis–BETA Framework 44
4.2.1 Tools used 45
4.2.2 Taint Problem Setup 47
4.3 *4P-Based* Attacks 52
4.3.1 Intent Spoofing Attack 52
4.3.2 Intent Hijacking Attack 53
4.3.3 Confused Deputy Attack 55
4.3.4 Collusion Attack 56
4.3.5 Self-Evolution Attack 57
4.4 Concluding Remarks 59

5 Experimental Evaluation **61**
5.1 Customized Sample 62
5.2 Real Samples 64
5.2.1 Case Study: Lango Messaging 68
5.2.2 Case Study: BomBom-Free SMS 71
5.3 Concluding Remarks 75

6 Conclusions and Future Work **76**
6.1 Conclusions 76
6.2 Future Work 77

Bibliography **79**

List of Figures

2.1 Android Framework[14] 11
2.2 APK Framework 19

3.1 A Call Back sample in an Application 32
3.2 Flow Chart in a Broadcastreceiver 34
3.3 Four Data-Flow Paths for a Broadcastreceiver 36

4.1 BRVD Framework 39
4.2 SUKE Framework 40
4.3 Intent Spoofing Attack 52
4.4 Intent Hijacking Attack 54
4.5 Confused Deputy Attack 55
4.6 Collusion Attack 56
4.7 Self-Evolution Attack 58

5.1 Filtering Log 63
5.2 BETA's Result 63
5.3 Source of Taint Analysis in vulnerable Receivers 65
5.4 Sinks in Vulnerable Receivers 66
5.5 Lango Messaging Analysis Report 69
5.6 BRVD's Report 70
5.7 Analysis Reports From ApkScan and Eacus 72
5.8 Analysis Reports From BRVD 73

Listings

1.1 A Code Snippet in the Hypothetical Android Application 5
3.1 Static BroadcastReceiver Registration 26
3.2 An Example for the Dynamic Broadcastreceiver Registration 29
4.1 VLC BroadcastReceiver 53
4.2 Malicious BroadcastReceiver 54
4.3 code snippets of Android GoldGPEN Spyware in the Tweetcaster . . 58
5.1 code snippets of Lango Messaging 71
5.2 code snippets of BomBom-Free SMS 74

List of Tables

4.1 Symbol Table 43
4.2 Privacy APIs 47
4.3 Source Methods in X_1 48
4.4 Sink Methods in Y_1 50
4.5 Sink Methods in Y_2 50
4.6 Attacks based on $4P$ policy 60

5.1 The Results of Suspicious Applications Using BRVD. 64
5.2 Testing Results from ApkScan and Eacus 67

List of Abbreviations

AMS	Activity Manager Service
API	Application Programming Interface
APK	Android application package
BRVD	Broadcast Receiver Vulnerability Detection
BSD	Berkely Software Distribution
CIA	Confidentiality, Integrity, and Availability
DVM	Dalvik Virtual Machine
GUI	Graphic User Interface
ICC	Inter-Component Communication
IDF	Inter-Procedural Data-Flow
IPC	Inter-Process Communication
IR	Intermediate Representation
PMS	Package Manager System
SBL	System Broadcast Library
WALA	Watson Libraries for Analysis
XML	Extensible Markup Language

Chapter 1

Introduction

With the rapid development of mobile networks, mobile phones are no longer restricted to telephone calls and text messages. Smartphones have emerged in response to customers' expectation as well as developers. The major smartphone operating systems on the market include Symbian, Apple IOS, and Android OS.

Symbian was reputed to be the world's best-selling operating system before the arrival of the Android OS, whose system framework has become outdated with years of utilization, coupled with innovated advances. Since the Android OS has provided a convenient open-platform, an increasing number of users have been using it for daily life, such as online banking and shopping, when in fact, imminent security threats arise due to fast adoption. An authoritative study, IDC report [8], has shown that the Android smartphone has exceeded by an 11.7% market share with the IOS (IPhone), and has already taken over 84.7% of the world smartphone market by 2014.

Compared with IOS which has the secondary market shares, the Android OS is built upon the idea that the user does have the autonomous right to access to their device as an administrator, and all installed apps are accommodated within its own process sandbox. Furthermore, all privileged resources have been explicitly displayed and must be granted before the installation. The freedom control and reboot of the Android OS is what appeals to users. The drawback of this mode lies within the debate that the general user does not have the capacity to comprehend the connotations of their security choices or chooses not to follow recommended practice, nor does it give reason for a well-intentioned developer who inadvertently makes resources available to malicious individuals who wish to exploit them.

1.1 Background

Android OS is a free and open source platform that allows various of applications to be developed and installed from freedom online markets. In addition to meet the demand of market-oriented economy and users during the era of functional and innovative, companies are more inclined to provide better entertainment features than security solutions. As a consequence, this further opens doors and rises vulnerabilities for hackers to explore attacks on the Android system.

Besides the incommensurate security measures in the product deployment, a chaotic third-party market[45] is accountable for rampant Android malware abuse. Because in the smartphone market the role of Android OS has become increasingly important, the Android market has different proportions of possession. As a result, many Android applications can be released by third-party markets and forums without verification. This ability provides convenience for many developers, without involving a heavy and complicated validation process, while creating a chance for critical attack. For example, the "Ghost Push"[25], as an un-installable virus, accordingly evolves in infection sprees through users installing malicious apps offered by third-party markets, rather than the official Google Play Store. Even worse, the infected smartphone will imperceptibly and passively install other malwares. The above study suggests this virus is infecting 600,000 users per day, which has resulted in an increase in economic losses as well as the number of privacy disclosures.

If used correctly and responsibly, the Android OS is eminently secure. A 2014 Google security review [39] has indicated that fewer than 1% of Android devices have malware. Not surprisingly, under the upsurge of Android malware, the majority of users fail to review the privacy disclosure statement prior to acceptance and utilization of the application [10].

These factors have led to Android devices being easily infected with malware. Although the results are not surprising, the collected data quantifies what has been a generally accepted fact for years: users are either disinterested in or unaware of the computer security risks to which they are exposed, so frequently do not take advantage of the provided security tools. Beyond that basic comprehension is the question of what approach should be taken by software and hardware manufacturers to secure their customers' data, from an economic, legal, or social perspective.

For users, the security of the Android system directly affects the user experience and privacy information, so that a reliable security system is the key to the future for sustainable development and promotion. For developers, a better understanding

of Android security is more conducive to protect user privacy information, which improves the user experience and satisfaction as a virtuous circle for the industry. Therefore, security enhancement in the Android system has become an important research topic.

This thesis focuses on privacy information leakage and defense that target Android users. It introduces and analyzes the Android security mechanism, especially the security issue of the Broadcastreceiver component in the framework. In addition, a Broadcastreceiver detection system, which enhances the security of the Android, is proposed and implemented.

1.2 Objective

The primary goal of this research study is to investigate the security risks affiliated with the utilization of Android phones. Furthermore, proposing a design for a real-time, effective, and integrated defense framework for the Android application is a secondary purpose. This thesis will contribute to identifying classical security threats facing Android phones and the corresponding privacy information. The proposed solution will detect attacks, such as virus infections and malicious phishing, and prompt users to take actions to prevent potential damage; any suspicious behavior that may reveal privacy material to third-parties or unknown ports will be announced to users. Android phones contain easy-to-exploit vulnerabilities as well as sensitive personal data. This collectively offers appealing motives for hackers to target consumers to gain financial incentives. A focused review of the literature [36] discloses that there have been abundant protection strategies offered for securing Android phones and ensuring user privacy. On the contrary, most of those endeavors are small-scale and tackle particular areas of protection, such as the EPICC [29] and Flowdroid [2]. This research is different from past studies. It leverages previously proposed and implemented defense strategies, and presents an enhanced protection framework that will further address Android's weakness and threats, particularly, those security issues in Broadcast Receiver; an important Android application component. Moreover, this study will extend the existing knowledge relevant to the Android security and provide in-depth explanation of how to effectively manage emerging risks and fend off attacks.

1.3 Motivation

Android, as an open-source operating system, has led to a growth in market share, while the security issue of Android research receives a rising in interest in past years. In particular, most current studies[29, 23, 3, 44, 22] focus on the Inter-component communication and permission leakage, whereas some [30, 17] state the specific component security issue related to the Activity, Service, and Content Providers. Instead, few studies discuss the security problem of the Broadcastreceiver component.

The Broadcastreceiver component in the Android system is normally used for listening to the appointed event and returning the correlated response, such as providing the screen locked notification for users when they touch the mobile screen. Indeed, many system events can be listened to, including the low battery warning and the timezone changing. Nevertheless, the Broadcastreceiver component has potential security issues, while it is listening to the registered event. As an event listener, the Broadcastreceiver is the important crossing of information hinge between the caller and callee. At the same time, the transferred data during the hinge risks being modified and hijacked. In other words, a victim receiver may leak the sensitive information that jeopardizes user's privacy. Meanwhile, the receiver can be not only exploited by attackers, but also might be used as a malware for attacking purpose.

Furthermore, the Broadcastreceiver is invisible for users because it runs in the background, which means it strengthens the feasibility of the attack. In addition, Google provides two different registrations for the receiver that can receive the broadcast intent: dynamic registration and static registration, respectively. The original goal of the multiple-registration is to make the Android application development become more flexible, but it brings more risks of the changeable attack. The static registration is registered in the Manifest file and it cannot be easily destroyed because of the macro-monitoring when the Android phone starts. Since the registration is written into the *Manifest* file, the lifecycle of the receiver has been together with the whole specific application's lifecycle, even if that application is an unknown app who can trigger the non-indigenous receiver in other applications. Compared with the static registration, the dynamic registration is more capricious that parasitics on other components as a listener to follow the change of a particular event. For example, a receiver can be part of an Activity component's thriving decay registered with using *registerReceiver()* . This means the receiver is only to surveil the specific activity, and it is hard to be tracked at most of time because of its thanatosis character.

Indeed, there are security issues in both registrations of the broadcast receiver. Meantime, the Android system offers receiver with several ways to restrict the unknown broadcast for safety consideration, such as value of *android:permission* and *android:exported* in the receiver's registration [14]. For example, the *android: permission="perm.P_Limitation"* limits any broadcast receiving except who owns the same permission value in the Intent sender *sendBroadcast(intent, "perm.P_Limitation").* Also, the other securing approaches, such as *LocalBroadcastManager* which provides a private inter-component communication for an individual application, will be introduced in the Chapter 3.

Although there are some restricted approaches to protect the receiver, the emphasis on the protection is apparently not enough. Further, the method of attacks for Broadcastreceiver component has not been investigated by Android security communities. In order to show the security issue of the Broadcastreceiver, a hypothetical example is demonstrated in the listing 1.1, which contains three code snippets in different Java classes.

Listing 1.1: A Code Snippet in the Hypothetical Android Application

```
1 public class getMessage extends ActionBarActivity{
2 ...
3 protected void onCreate(Bundle savedInstanceState){
4  new MyAsyncTask().execute(); //start an AsysncTask
5  Handler seconds = new Handler() {
6   @Override
7   public void handleMessage(Message msg){
8     String me = msg.getData().getString("key");
9     Intent intent = new Intent();
10    intent.setAction("MyBroadcast");
11    intent.putExtra("value", me);
12    sendBroadcast(intent); // send a broadcast
13       }
14        };
15  }
16 }
17 public class MyAsyncTask extends AsyncTask<
18 Void, Void, List<Address>> {
19 ...
20 protected List<Address> doInBackground(Void... arg0){
21 ...
22 list = geocoder.getFromLocation(latitude,longitude,1);
23 ...
24   Message m = new Message();
25       Bundle b = new Bundle();
26       b.putString("key", String.valueOf(list));
27       m.setData(b);
28       Handler handler = null;
29       handler.sendMessage(m); //send a handlerMessage
```

```
30 ...
31  }
32 }
33 public class MyBroadCastReceiver extends BroadcastReceiver{
34   public void onReceive(Context context, Intent intent){
35    Bundle extras = intent.getExtras();
36 ...
37  String message = (String) extras.get("value");//get "value"
38  int a = 5556;
39  String phoneNo = Integer.toString(a);
40  SmsManager smsManager = SmsManager.getDefault();
41  smsManager.sendTextMessage(phoneNo,null,message,null,null);
42  // send a SMS
43   ...
44  }
45 }
```

This is a hypothetical Android chatting application that provides SMS and GPS functions for users. It shows three piece codes from a main activity class, an AsyncTask class, and a broadcast receiver class respectively. Similar to most of apps in the Android market, it aggregates the popular free internet call and the address sharing between friends. Once the user installed the app, it will remotely acquire and send the user's location information back to the visual phone number, without users' concern. This data theft is not uncommon in the application market. Furthermore, the messaging return might be a simple attack in the real world, the data uploading is a much more convenient way based on the hidden link in the child thread. Here we elaborate on this possible attack on this example and discuss the attack mechanism associated with the clear code.

In listing 1.1, method *Bundle* (Line 25) is invoked by the AsyncTask whenever a requester connects to the main activity class *getMessage*. The *execute()* method (Line 4) in the *getMessage* will start an AsyncTask. From now on, the GPS information is collected as a *list* in the *doInBackground()* method (Line 22). The bundled data "b" in line 26 within data will be set into *Handler* method as a message in line 29. Consequently, this message is going to send back to the main activity class. Once the main thread activity obtain the handler message by the objected *Message* method in line 8 with *getData()*, the bundle data "value" is prepared to package into the Intent mechanism (Line 11). The *Intent* then will send a broadcast (Line 12), and pass to the next component which is *MyBroadCastReceiver* class. Finally, the original user's GPS information will be received in line 35 using *intent.getExtras()*, and delivered as a SMS message to the appointed phone number in line 41. To avoid this kind of malicious attack, most of users should install apps from the official Android market

as recommended. Meanwhile, this thesis demonstrates a detection system for this kind of attack to protect users' privacy.

1.4 Methodology

The rapid development of the Android has, simultaneously, attracted hackers and criminals. For the sake of an increasing demand for malware application detection and prevention, a Broadcast Receiver Vulnerability Detection (BRVD) system is proposed as a static analyzing tool in the thesis. Moreover, the purpose of BRVD is to analyze and vet the vulnerability of the independent Broadcastreceiver component in the Android app.

Due to the fact that the Broadcastreceiver component in the Android app is invisible running at background, the BRVD combines semantic analysis and taint analysis together to a complete inspection. In general, the vulnerability of the receiver can be transformed into an equivalent leakage of the sensitive resources problem. Therefore, the data leakage of the receiver could be identified as a data-flow analysis method. Then, the detection of the filtered receiver is transformed into a taint problem[2] that seeks to identify the sensitive information leakage.

To vomit the warning result from the malware for the Android user, there are several steps in our detection system, which are preprocessing, receiver generating, data-flow analyzing, and result assessing. In order to request the application's source code, the Android app needs to be decompiled as a readable file. After filtering the source code based on the semantic analysis in the second step, the generated receiver will be the input of the third step which analyzes the data-flow to assign into different classification. In the end, the organized result is collected and presented.

Compared with other detection systems, the BRVD fills a gap which has hitherto existed in the Android security issue. What is more, the combination of the semantic analysis and taint analysis is based on the automated testing concept. This means a set of broadcast receiver testing is advanced and convenient.

1.5 Contributions

This thesis focuses on the malicious code method in the Android platform and does more exploration and research, especially the Broadcastreceiver component. The main contributions of this research as following:

- Identified a new attack against Broadcastreceiver based on data-flow analysis, as well as the security threats with deep-seated reasons. In addition, as stated in the significance of the security research background, from the in-depth study of the present implementation mechanisms, we analyzed current attacks' methods and defense.

- Formulated a set of malicious behaviors in various data-flows through the Broadcastreceiver component, along with the security design flaws of the inter- and intra- application data transmission mechanism. In contrast with both of the dynamic and static analysis, we summed up advantages and disadvantages of existing Android's theories, and put forward static analysis that combines semantic and taint analysis to detect the vulnerability of the Broadcastreceiver component.

- Designed and developed a Broadcast Receiver Vulnerability Detection (BRVD) system which is based on the semantic and taint analysis. After decompiling the Dalvik bytecode, the pre-defined data structure will be reassembled with the sensitive API call to unify the warning label, and track the privacy data direction to identify the unsafe element of the broadcast receiver. Compared with the similar tools, it has a faster filtering function within compatibility, usability and accurately.

- Collected and designed a System Broadcast Library (SBL) for Broadcastreceivers, which are permission-protected for system receivers' registration. According to the different registration forms in Broadcastreceiver component, we proposed a new detection logic based on registration formats. Concretely, considering the difference between the static data member and the normal text data file in the system action registration, the action library in the BRVD provides two input types of the system broadcast action for the receiver's registration.

1.6 Thesis Organization

This thesis studies the threat and the security mechanism of the Android Broadcastreceiver. On account of the security risk in the Android platform, we proposed a malware detection model which combines the semantic filter and data-flow monitor. In the light of the characteristics from the PC and Android phones, the detection model based on the PC platform can reduce the mobile phone's workload and provide a large amount of the data storage. Also, we apply this model to improve the

detection accuracy of the Android malware. The thesis consists of six chapters, and the chapters are organized as follows:

Chapter 1 It presents a background of the subject. Furthermore, a motivation example for architecture of the Android platform is described detail, along with the purpose and significance of the thesis. In the end of this chapter, the organization of the thesis is presented.

Chapter 2 It introduces the architecture of the Android platform and the threat malware model, as well as the current research associated with the Android security technology and the corresponding open source tool. This platform security design also provides the theoretical reference for our detection system.

Chapter 3 It analyzes a new classification of the Broadcastreceiver data-flow detailedly, including the broadcast sending, receiving and the security analysis of the registration. Meanwhile, the static registered receiver and the dynamic registered receiver are analyzed separately, together with different attack paths of the data-flow respectively.

Chapter 4 It proposes the BRVD's framework and implementation. Precisely, the main function of the each sub-module is introduced, in conjunction with the relevant key algorithm. The model focuses on combining the characteristics of the text registration and the taint analysis in the application, to avoid the impact of the user behavior on the test results. Accordingly, it improves the detection accuracy.

Chapter 5 It depicts an experimental evaluation for BRVD. Based on a customized sample and real Android applications. BRVD analyzes and provides detailed statistical data related to testing results. In addition, compared with two Android security detection tools, there are two vulnerable Android applications as case study for a specific analysis.

Chapter 6 It states a summary and future works for the thesis. Moreover, the advantage and disadvantage of BRVD are indicated. In accordance with the deficiency existing in this research, aspects which should be awaited the further amelioration and research were put forward.

Chapter 2

Background and Literature Review

Wireless technology has revolutionized the way people communicate and live, while smartphones have become indispensable in the private space for personal information. They are increasingly being used to track fitness and health information and access bank accounts. Extensive storage of sensitive details make smartphones more attractive for attackers. Since the first "SMS Trojan" [45] was discovered, the malware on the Android platform has drawn attention. To mitigate attacks and enhance security, new mechanisms for the Android platform have frequently been proposed. In this chapter, we provide a detailed description of Android security mechanisms and review some malware threats with current research solutions.

2.1 Android Overview

Android is an open-source application execution environment that includes an operating system, application framework, and core applications. Android is designed and released originally by Android Inc. to provide a user-friendly, open and mobile-based development environment. This open-source mobile development framework is user-centric, providing a variety of development tools and features. Nevertheless, the open-source nature of Android also poses challenges in securing sensitive user data, such as those third-party applications that usually phish and send some attractive information to Android users.

2.1.1 Android Framework

The Android system uses a multi-layer architecture system. It is divided into four layers from top to bottom: application, application framework, libraries and Android runtime, and Linux kernel. Java language is used in the application and the

application framework layers. The libraries and Android runtime layer is composed of a Java coding environment (Dalvik Virtual Machine) and core libraries with local codes (C and C++ database). The Linux kernel layer incorporates a custom Linux system and the drivers; mainly implemented by the C language. Figure fig 2.1 shows the Android framework [14], which is detailed below.

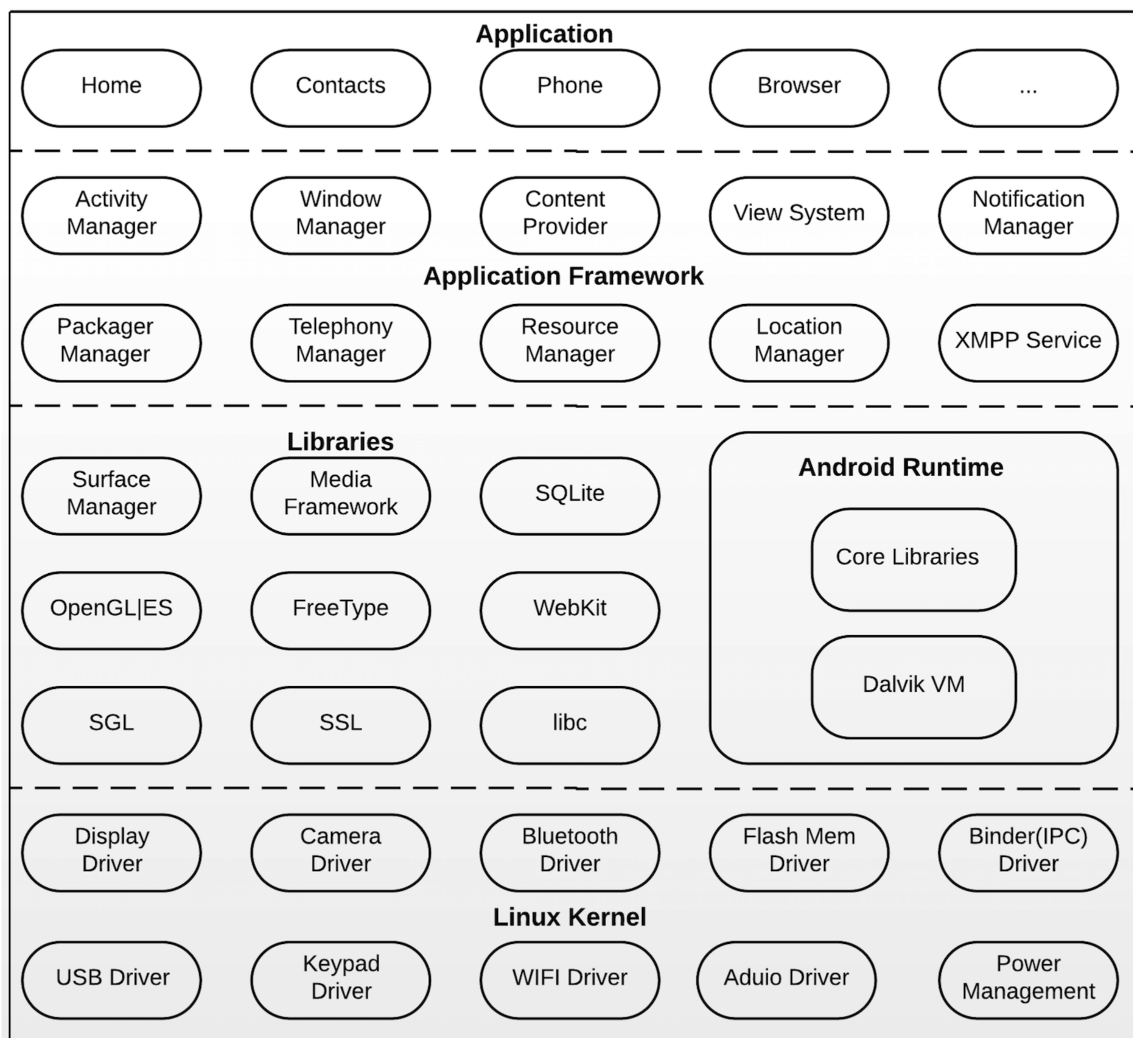


Figure 2.1: Android Framework[14]

1. First Layer-Application

    As a user application layer, it contains basic smartphone's functions for the Android user, such as Email client, SMS programs, browser and contacts etc.

2. Second Layer-Application Framework

    This was designed specifically for the Android platform application devel-

opment, which allows full access to core applications in the API framework. It Consists of services and systems including the view invoking, package manager and content providers etc. All APIs inside are functionally offered for various apps.

- View- An abundant set of scalable views can be used to build applications such as lists, tables, buttons and embedded webpages
- Resource Manager- Accessible un-coded resources such as local string, bitmaps and layout files.
- Package Manager- Has the privilege for enabling and disabling installed applications, including *getAllPermissions()* and querying the specific component
- Message Manager- Allows applications to be displayed in the status bar with custom view.
- Task Manager- Manages the application life cycle and it is used for general navigation

3. Third layer-Library and Android Runtime

   This layer is mainly associated with the process running where the library provides JAVA programming language for the majority of the core library functions. Additionally, each program has an Android Dalvik's java virtual machine for its operating environment.

   - Android includes a C/ C++ library for each part of the Android system.
   - Lbc is a standard C library with Berkely Software Distribution (BSD) implemented for its embedded Linux devices.
   - SGL (multimedia library) is based on PacketVideo's open-source kernel. This library supports playback and record of many popular audio and video formats as well as static pictures such as MPEG4, H.264, PNG and others.
   - Surface Manager controls a multi-application access to the display subsystem and continuously mixing 2D and 3D graphics systems.

   Each Android application creates a DVM to run its own process. Owing to that Dalvik has been rewritten, thus each device is able to run multiple VMs effectively. The DVM relies on the Linux kernel which provides many implication functions, including threads and low-level memory management.

4. Fourth layer-Linux Kernel

   Android's kernel is the model of Linux2.6 kernel. The kernel is the fundamental part of the Android system where its main role is to interact with the computer hardware. In fact, it will implement not only to control hardware programming and interface operations, but also to schedule and access hardware resources. At the same time, the kernel provides a high level execution environment and the hardware virtual interface. The key features include service routine interrupting, scheduler processing, address space memory management, inter-process communication(IPC) and network protocol etc. In order to make the android system more adaptable to the mobile devices, the android system adds some special drives and some of them are not belonging to the Linux standard drives. Normally, they do not need to operate with the real hardware, but assist the system to process smoothly. The main Android drivers are as follow:

   - Ashmem is anonymous shared memory. Through this kernel mechanism, it provides memory allocation scheme for user-space programs to implement functions similar to malloc.
   - Logger driver supplies lightweight log support for the application programs
   - Binder driver is based on the driver of the Open-binder system, along with the support of the inter-components communication in the application. Furthermore, the whole Android system operation relies on the Binder for the process to process communication.
   - Power Management (PM)
   - Low Memory Killer driver will end the process, if the process lacks memory.
   - Android PMEM is a physical memory driver.

2.1.2 Application Components

Android application framework requires developers to conform to Android's architecture while programming an application. There are no *main()* functions nor similar execution entry points in Android applications. On the contrary, applications are composed of component concepts.

These concept of components are the cornerstone in the Android system, and provide for the entry points into the application. However, not every component are entry points for a user's application. Many components are interdependent. Each component plays a specified role and exist as an independent entity. In other words,

every component plays a unique part, collectively they define the overall behavior of an application. In the Android system, there are four main components, which are Activity, Service, Broadcastreceiver and Content Providers.

* Activity Component

  The purpose of Activity is to provide a graphic user interface (GUI) interaction. Furthermore, every Activity is an independent interface which could not only display other GUI widgets, but also monitor and respond to a received event. Generally, an application contains more than one Activity which could either run in the same process or in different processes. For Activities in different processes, they will co-operate with each other to complete application functionality through the Inter-Process Communication (IPC) mechanism.

* Service Component

  Service is the one of the four modules in the Android. The difference between Service and Activity is that the Service is mainly used to deal with the user interface independent of the business logic. Due to the fact that the Service component does not need to directly interact with the user, its tasks are often related to computational processes which can be time consuming. This is applicable to run in the background and prevent blocking the user interface (UI) thread. There are two approaches to start the Service which are explicit and implicit. For the implicit way, it only needs to pass the action by using the Intent object. By comparison, the explicit needs to clarify the specific class name and even the package name. Moreover, the Service component can be activated by other components, such as the Activity component. Meanwhile, Service can run in the main process with other components that trigger it to start, it can also be created as independent sub-process in the background.

* Broadcastreceiver Component

  Due to the function of the inter-component communication in separated processes, the Binder in the Android Kernel is a significant driver for the broadcasting function. Accordingly, the essence of the broadcast is implemented through the Binder. This mechanism is a message publish/subscribe model based on the event-driven. In other words, the recipient needs to be registered before broadcasting information, and then receivers can accept the message from a caller. As a result, this registered receiver in the Android system is abstractly regarded as a Broadcastreceiver component. Simultaneously, the three main

components in the Android system; Service, Activity and Broadcastreceiver, can easily interact with each other via using the Intent mechanism. Besides this, although the broadcast receiver does not have the user interface, it can either through the corresponding Activity and/or Service act in response with a registered event, or be activated by the correlative broadcast.

* Content Providers Component

  By reason of the limitation for the Linux user ID, different applications do not have access directly to each other. Therefore, Content Provider, as a public database, is the one to realize the function of data sharing and transmitting among Android applications. A typical example is to apply Content Provider to access the main address book. Applications with related permissions will obtain the contact information through the Content Provider. In addition, the Content Provider can transfer data based on the IPC to applications by using the Binder. However, due to the inadequacy of the Binder transmission for the large amounts of data, the Android system also provides an Anonymous Shared Memory to achieve the big data sharing between processes. In fact, only the file descriptor is passed between processes. Therefore, on account of the synergy of Binder and Ashmem, the Content Provider can realize the efficient data sharing.

2.1.3 The Intent-based Inter-component communication (ICC)

The Intent is an important connecting bridge among the four components in the Android system, which specifically is a run-time binding mechanism. In particular, the intent is an objective that is instantiated by the Intent Class in the Android and it can bind running components together within data transmission. Apart from this, the Intent text input during the registration has three main parts which are; action, data and category. Over and above it, the main functions of the intent can invoke the *startActivity (Intent)* as an argument to start an activity, be received as an intent broadcast (*sendBroadcast (Intent)*) by the registered components and communicate with the Service component through *startService (Intent)* or *bindService ()*.

Indeed, the Intent mechanism has the explicit intent and the implicit intent. In detail, both of them can be called by Activity, Service and Receiver. On the contrary, there are some differences between them.

- The explicit intent: It has the specific class name or the package name outlined in the Intent registration when it is being called. Thus, the explicit intent always points out the exactly callee direction.

- The implicit intent: It only needs to meet the action name matching requirement when it is being called. Therefore the Intent is exposed to any callers and several application with the same action name can call upon the intent simultaneously which risks sharing data with vulnerable receivers.

Thus, the explicit intent is more secure than the implicit intent, because the implicit intent supplies an omni-directional data transmission without security control while explicit needs the clear class name.

2.1.4 Permission Protection Mechanism

Android is a “Privilege Separation” system. Restricted resources, such as Bluetooth and SMS functions, should have the corresponding registered permission before applications downloading. In turn, an application can process to utilize the appropriate resources after installing. The permission mechanism is a many-to-many mapping relation between Permission and Java Application Programming Interface (API). This means Android applications will call the relational API to complete its function after acquiring the matching approval. The process of the API calling from user-behavior to the specific function realization can be divided into several steps: An application will call the public library API after permission is obtained, then the public library API invokes an interface named RPC stub which will send requests to a system service via using inter-process communication (IPC) bindings format. In the end, the system service progress will complete specific functions. Actually, the permission detection will be running during the period between system services and system processes.

In the Android system, all installed applications must own a digital certificate. Otherwise, the Android system will never accept an application without a digital certificate. In detail, the digital certificate is applied to identify the trust relationship between authors and applications. Besides this, functions in the Android application can be activated by both of user behavior and other applications. For example, Google map location is an external method that can be invoked by many Apps associated with the GPS permission. Consumers take a risk to leakage the location information through installing malicious application that trigger other apps who have the GPS permission.

To defend the malicious switch-on in the inter-application communication from attackers, the permission mechanism provides a customized permission to protect non-system action. The strength of the protection is divided into four levels; Normal, Dangerous, Signature, and SignatureOrSystem permissions.

- Normal: It is granted and approved to activate acquiescently by the Android platform, but this default value is limited inside the application without outsides requesting. Thus, the normal level is low risk permission.

- Dangerous: It notifies the specific accessing function to users before the application installing, and does risk the method calling associated with the user's privacy. Hence, this level has potential danger.

- Signature: It is invoked automatically without the users notification by other applications which have the matching certificate. Once the authentication between the caller and the requester is verified, the system will grant the permission.

- SignatureOrSystem: the function will only grant and share to the other APPs which is in consonance with APKs signature. No matter how many activities in other APPs defined by the same signature, the target permission only matches completely with the same APKs or Android package classs signature. Especially, if two different APPs want to call each other, they should reach the signature matching requirement that is created by the same and unique author.

The associated permission requirement in the Android application is written by the developer, but is permitted by the user. In other words, the user will make the final decision to choose installing or not, whereas the majority of users do not have the capacity to understand the security choice.

The permission mechanism seemingly is literally safe. On the contrary, it is "Overlord Provisions". The reason for the inflexibility of the permission mechanism is that it only provides two options for users; either they have to approve all permissions before installing the favorite application or disapprove and cancel the utilization. In addition, once the user has approved the permission and activates an application, the activity of this application during the lifetime cannot be stopped from the user's operation, such as automatically completing a malicious SMS sending event in the background before uninstalling.

2.2 Android Threat Models

There are many variable applications in the Android third-market. What's more, some of them are even lurking inside the official market. As stated in the user's perspective, the loss from various of attacks and the Confidentiality, Integrity, and Availability (CIA) security model standard, attacks can be divided into five models;

information disclosure, information destroyed, function blocking, fraudulent billing and fraudulent information.

- Information Disclosure: a malware illegally gains users' privacy information by attacking smartphones, including their address book, SMS, and GPS location, which directly affects consumers' life and work.

- Information Destroys: a malware modifies and deletes users' privacy information via infecting smartphones.

- Function Blocking: a malware interferes functions in smartphones through distributing denial of service attacks. The function blocking includes stopping the cellphone system, continuous cellphone restarts, disruptions in the interface display, interfering with the cellphone signal, intercepting phone calls and SMS, illegally terminating and uninstalling the APP.

- Fraudulent Billing: a complete mobile communication network has a complicated billing system, and some of operations are directly related to consumers' fund account. Fraudulent billing can cause the loss of money by faking bill, which might bring an economic loss to its users. This behavior contains unconsciously dialing and sending SMS, passively connecting network and illegally customizing value-added services.

- Fraudulent Information: Using users' contacts to spread fraudulent information, so that false messages would be received by users' friends and families who might have emotional and material loss. The way of deception contains applying malicious advertising plug-ins to advise phishing websites, leveraging users' relationships network to spread fraudulent information.

2.3 Analytical Measure

The detecting approach in the Android security can be concluded into two main fields. The pre-installing analysis which is static behavior-based detection and the dynamic behavior-based detection that is real-time based analysis.

2.3.1 Static Analysis

Static analysis[42] uses the special software tools to analysis software including files and execution code before running this APP. It typically obtains codes from

disassembling software, then analyzes this softwares execution process, and code logic by using the manual analysis. There are several main key technologies, from the code extraction to the method analysis.

Due to the fact that the APK in Android platform is similar to the JAR package in Java program, so extracting and detecting feature codes from the original package is one way of Android security analysis. In turn, we can detect the feature code from three steps; APK (filename.apk) extraction, DEX (classes.dex) decompilation, and XML (AndroidManifest.xml) analysis.

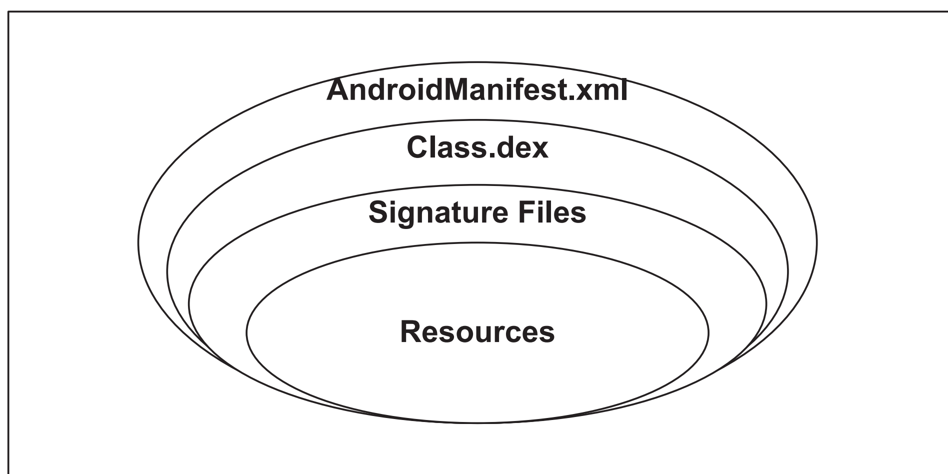


Figure 2.2: APK Framework

* APK Extraction

  Based on the reverse engineering[13], feature codes in a procedure of an application will be extracted, including binary sequences, operating codes, and function callings. Furthermore, feature codes will never be modified. Otherwise, the modified feature codes in the application will be dysfunction, even fail to start. In Figure 2.2., it states an APK framework, as an installed package for the Android application. There are specify files inside :

  - APK features: There are not only source codes in the APK but also non-code resources, which are extrinsic attributes and intrinsic attributes. Specifically, the size, name, and building date of files are belong to the extrinsic attribute, while the information of URL, version and permission list in the package is the intrinsic attribute.

- XML files: Each APK contains several XML files. What is more, the *AndroidManifest.xml* is one of the most important file in the Android APK, because most of Android permissions will be declared in this file. Elements in XML files contain sub-elements with related attribute value that can announce and affect all component functions, such as icon, label and permission.
- DEX files: DEX files, constructed with bytecodes, are required by APK files in the Android OS. The DEX file is generated by Dalvik Visual Machine (DVM) from Java source codes.

After extracting the APK file, there are APK features, XML files, and DEX files. On account of the unreadable DEX file, the DEX decompilation is requested to be a readable Java file.

* DEX Decompilation

DEX Decompilation is a specific reverse engineering for the file decompiling. There are many DEX decompiling tools, and they can obtain java source codes from the DEX byte-code. For example, the DEX file in an APK can be transform into normal JAR format files by using such as *dex2jar* [32] or *Baksmali* [34] tool, and then we can check the Java source code in JAR package by using tool *JD-GUI* [33]. As a result, the Android software can be analyzed via implementing the semantic of codes.

2.3.2 Dynamic Analysis

The dynamic analysis is an approach to investigate behaviors of programs' execution in strictly controlled environment (sandbox). Also, the real-time demand of the dynamic analysis needs to be accurate, so that malicious behaviors would not cause a great damage. In android platform, the environment of the dynamic detection generally applies sandboxes or the DVM for simulating applications. There are two main methods in the dynamic analysis; they are status contrast and behavior tracking.

* Status Contrast

It is an analyzing method that is to compare pre- and post-execution of the system status, along with the source data extraction. Nevertheless, the drawback of the method is that it is easily affected by the changing status of superposition, and the detecting result is not accuracy.

* Behavior Tracking

  It can dynamically capture the execution of a program operation and analyze the program behavior. According to the different implementation techniques, the behavior tracking can be divided into two categories: instruction level analysis and lightweight level analysis.

  - The instruction level analysis could not only obtain or modify the status of CPU, memory, and value, but also change the control flow during a program execution. However, the time consuming is longer than other analysis.
  - The lightweight level analysis method can implement some technologies, such as *HOOK* technology[24], to track the run-time based behavior. Owing to the fact that the amount of data resources on the light level is small, so this analysis is faster and easier than other similar methods. Simultaneously, the accuracy is relatively high and has a wide tracking range.

In conclusion, the static behavior-based detection abstracts program characteristics through using reverse engineering, such as binary sequences, operation code sequences, and function call sequences, while the dynamic analysis monitors real-time behaviors after the operation system service or component authorization.

Both of static analysis and dynamic analysis are significant for the Android security. Compared with the dynamic analysis, the static analysis has lower energy consumption and lower risk, but accurate.

2.4 State of the Art

With the development of the Android security detection technology, there are two main platforms that be implemented into the data-flow analysis for Android application research; soot[20] and the Watson Libraries for Analysis (WALA)[12]. The first platform is developed by Sable Research Group of McGill University, supplied an JAVA optimization framework. The other one, WALA, is issued and donated as a open source tool from IBM company that provides a static analysis for java bytecode and javascript. Currently, most of Android detection tool based on the static analysis are built on the soot and WALA.

Thanks to Felt et al.[7], a clearly inter-application communication is defined in Android applications. They examined a communication mechanism in the Android

platform and developed a tool named *ComDroid*. This tool is based on the Dex code that is applied for identifying suspected components in an Android application.

Huang et al.[16] applied a WALA-based tool, *AsDroid*, to detect stealthy behaviors based on the identification of the difference between application's behaviors and user interface text. In particular, they presented the relationship between interface text from those frequently used key words in the Graphical User Interface (GUI) of android applications and the associated user's behavior. For example, when an user is clicking a button "Send Message" in a malicious application, there are any other behaviors inside not only message sending. In other words, according to the GUI design in Android applications, if this function is not matching with the UI text direct meaning, this will be defined as malwares. Furthermore, they implemented AsDroid to test 182 apps and 113 of them show the stealthy behaviors.

PermissionFlow, as a WALA-based tool for detecting Android application vulnerabilities in the bytecode and configuration, has been demonstrated by Sbrlea et al[41]. They provided a solution for the issue of monitoring the permission leaks based on the Android manifest file. Due to the result that 56% of tested 313 Android APPs have been utilized inter-component information flow. Hence, the relationship, also named rule generator in the thesis, between permissions and corresponding behaviors is constructed and saved as an Android library. According to the rule generator, this tool can check every event in an Android application through the Intent mechanism and filter the sensitive permission related to the users' privacy.

Didfail is a system that combined EPICC[29] and *flowdroid*[2] together. In specific, EPICC mainly focuses on the Intent mechanism and detects the information leakage by reason of the Intent bundle data, while the *flowdroid* is a soot-based analyzing tool that provides a precise data-flow analysis. Thus, this system is using the EPICC as a filter tool to select vulnerable components based on the unexpected connection. In addition, the output of the EPICC will be the input of the *flowdroid*. Then the *flowdroid* will track the data-flow simply depending on the broadcast Intents, such as the *intent.sendBroadcast()*, to analysis the vulnerability.

Chen, T et al.[6] proposed a concolic execution combined both dynamic and static analysis, which is applied by the symbolic execution on hosts and the concrete execution on testing objects. They implemented their prototype on the "VxWork" system. Also, the speed for the code coverage is acceptable, because all time consumed works are finished on the resourceful hosts.

Apex[27] is a policy enforcement framework that provides permission options for users who are installing Android applications. In other words, the user is given an

ability to choose and deny permissions from an application from third markets. This framework is based on the static analysis that extends Android original framework. Moreover, the extending framework has a simple graphic user interface for users.

There are many defending tools from researchers who have paid attention to the Android malware and discovered its vulnerabilities. However, the Android platform still needs to be updated and improved. Most of researches only consider about the data transmitting between components, which is the key point for defending data leakage in Android, but they do not recognize the Broadcastreceiver component itself could be a malicious object. In other words, the Broadcastreceiver, for example, may contain some malicious URL links and submit privacy data via Internet. Researchers consider about permission issues, whereas some Android applications do not need permission by leveraging other applications. Meantime, the Broadcastreceiver is one component that can be used for bypassing related permissions. As a result, Broadcastreceiver can be an attacking avenue, but its related security issues have never been studied in the past. Therefore, we will give some insight into security and privacy issues associated with Broadcastreceiver component.

Chapter 3

BroadcastReceiver Analysis

A BroadcastReceiver represents one main component of an essential publish/subscribe messaging platform in Android. It, known as a subscriber, can be used to receive and respond to specific messages (or broadcasts) from the system or other components in the same application or different applications, such as notifying Android users when cellphone power is low as a system event-driven. This is generally achieved through the utilization of intent filters defined in the app manifest file. Nonetheless, the most common uses of the Broadcastreceiver component in Android application development are to act as a listener to receive Broadcast intent from the intra- and inter-applications[7]. They can be used for local listeners among different components in one application, as well as global listeners in inter-applications. Inside a self-contained Broadcastreceiver object, the expected operations to be performed are specified in the action field and filter definition with corresponding permissions.

In Android, a broadcast is delivered through the passing of an Intent object using Broadcast message sending methods, including *sendBroadcast()*, *sendOrderedBroadcast()* and *sendStickyBroadcast()*. Generally, the definition for Android broadcast mechanism is one event or broadcast intent to trigger many broadcast receivers with their intent filters that match the Intent. Based on the publish/subscribe message mechanism, different receiver registration modes will result in different published broadcasts to be received. There are three major categories of broadcasts that can be received:

- Normal broadcast: A normal broadcast is sent to all the eligible receivers at the same time without considering any order between them. Normal broadcasts can be transmitted implementing method *sendBroadcast()*.

- Ordered broadcast: It depends on the priority level, which is claimed by a receiver. After receiving a broadcast intent, the receiver will decide whether to

stop forwarding the received broadcast using method *abortBroadcast()*. If not, the Intent object will pass to the next matching receiver. Ordered broadcasts can be transmitted implementing *sendOrderedBroadcast()*.

- Sticky broadcast: Unlike other broadcasts, sticky broadcasts will be destroyed after executing transmission. It can be saved into Activity Manager Service (AMS) when it can not currently find any corresponding receivers. Once a new Broadcastreceiver has been registered and matched, the AMS in Android system will send this saved broadcast to the target receiver immediately. However, Google has deprecated the sticky broadcast after Android 5.0/API21, but it still can be used for its earlier versions by implementing method *sendStickyBroadcast()*.

Also, according to different utilization modes, two special types of broadcast may be particularly worth considering from perspectives of security. In detail, the system broadcast is triggered by the basic system operation, such as power on, network state changing, and screen off, etc. Instead of the attack threaten, the system-event broadcast can be safely delivered to all permission-protected receivers. Meanwhile, the *LocalBroadcastManager()*, as its name indicates, is the only one method that provides broadcast intents transmission inside an individual application. In other words, the *LocalBroadcastManager()* restricts other application's Intent from the passive activation. Compared with global broadcasts between applications, local broadcast is only limited to a single app, and is highly efficient and secure. Its associative receivers can be created by *LocalBroadcastManager()* as an inner class of a Java class in Android application development.

To analyze the Broadcastreceiver component security, we combined semantic analysis [11] and taint analysis together. On the basis of the subscribe/publish mechanism, the first step of analyzing in the event-driven model is the Broadcastreceiver registration, which also is for the identification of vulnerable receivers in our analysis. Then the vulnerable Broadcastreceiver will be verified from its message publish result as a classification of malicious behaviors.

3.1 Semantic Analysis in the BroadcastReceiver Registration

Compared with the static analysis discussed in chapter 2.3, dynamic analysis is not only time consuming, but also cannot accurately detect the vulnerable receiver

during the running time as long as the malware does not behaves abnormally. For instance, the target application is working well with all permission protected data flow under the detection, whereas dynamic analysis cannot point out the weakness of the receiver because there is no dangerous behaviors being triggered in this application. Hence, in virtue of the malware's elusiveness, the vulnerable receiver in an Android phone has possibility to be called by other malicious software without the relational permission authentication, and dynamic analysis in this situation is really inflexible for searching vulnerable receivers.

Semantic analysis, as one of static analysis, is based on the examination and determination of source codes. It is applied for understanding the semantics of the code, as well as the logical process. Thus, according to the logical order between the Broadcastreceiver registration and behaviors, the vulnerable Broadcastreceiver in testing objects can be detected and filtered. However, the Broadcastreceiver component in Android has two different programming languages, XML and JAVA, which focuses on the establishment of two quite different registrations. Therefore, the semantic analysis of the Broadcastreceiver component in Android applications needs to be divided into two types for analysis; static registration analysis and dynamic registration analysis, which will be discussed in detail in the following subsections.

3.1.1 Static BroadcastReceiver Registration

The Android static broadcastreceiver is registered in the *AndroidManifest.xml* and defined in the *manifest* file written by XML programming language. The Android system implements the Package Manager System (PMS) to arrange the static receiver. The format for the standard static broadcastreceiver is as follows:

Listing 3.1: Static BroadcastReceiver Registration

```
1  <xml> <receiver   android:enabled=["true"|"false"]
2                    android:exported=["true"|"false"]
3                    android:icon="drawable resource"
4                    android:label="String resource"
5                    android:name="String"
6                    android:permission="String"
7                    android:process="String">
8 ...
9             <intent-filter android:priority="-1000-1000">
10                    <action android:name="String">
11            </ intent-filter>
12 ...
13        </receiver>
14 </xml>
```

Among the various features of an Android application, their attributes have a certain structural relationship. Suitably, XML is a structural markup language, and its language structure characteristic attributes can be reflected to the relationship of the properties of an Android application.

As shown in Fig. 3.1., it can be observed that there are some attributes that are used indifferent interrelated tags, such as the *receiver* tag, the *intent-filter* tag, and the *action* tag. Precisely, each of tags has its own attributes for different purposes. The attribute in the *receiver* is generally implemented for the basic receiver's class type and limited the object, while the *android:priority* in the *intent-filter* is designed for controlling the order sequence of the receiving broadcast. Besides this, the *action* element represents the specific type of action name that matches the corresponding broadcast intent.

After successfully registering, the implementing class will extend the BroadcastReceiver's class and invoke the *onReceive()* method to trigger the receiver's reaction. Remarkably, the *receiver* tag is one subset of a *application* tag in *androidManifest.xml* files of Android applications. Especially, this *receiver* tag is not only one sub tag under the *application* tag, there are other sub tags, such as *activity* tags, *service* tags and any other *receiver* tags. Besides this, we found that a receiver may have more than one *android:name* under the sub-tag *action* in the *receiver*, and each of the action names signifies an independent callback function, sharing the same receiver's name with the rest of other actions.

As a consequence, to protect the static receiver in the Android, Google provides some attribute options in the *manifest* file for limiting broadcast Intent's abuse, and the receiver's external exposure, such as the *android:exported* and *android:permission* options under the *receiver* tag. Not surprisingly, some security issues in the static Broadcastreceiver registration have been discussed already. Felt et al.[22, 7] have already identified the threaten of the communication between components, for example, if the value of *android:exported* in the static receiver registration is "true" or null, this receiver might be vulnerable from other Android applications' calling.

Nonetheless, to be the best of our knowledge, existing works do not mention the vulnerability of the *android:priority* attribute under the *intent-filter* tag, nor do they mention the different formats of permission-protected system broadcast in the dynamic and static registration. Most importantly, the *android:priority* attribute under the *intent-filter* tag is the mandatory property for the ordered broadcast receiver, and this attribute value is within the range of the integer number from -1000 to 1000. Specially, the larger integer number with the ordered broadcast receiver, the higher

its priority. In detail, the first receiver to receive the ordered broadcast has the privilege to cancel and modify the rest of ordered broadcasts. Consequently, on account of the malicious cancellation and modification, the vulnerability of the priority attribute must be considered when designing an Android broadcast receive vulnerability detection system.

During the static receiver registration, most of the system broadcast intents' constant value in the action tag are protected by their related permissions, but all of them are registered as string characters in the *androidManifest.xml*. For example, the Android platform provides the permission *"READ_PHONE_STAT"* to limit and protect the relevant system intent action *"android.intent.action.PHONE_STATE"*. Thereinto both of them should be registered separately. In reality, the list of the system actions have been collected in the each version of the platforms under the Android Software Development Kit (SDK). Owing to the characteristics of the lower influence and non-synergism, the non-permission protected system broadcast intents have been considered safe. In detail, the non-permission system broadcast intents actually are not necessary to be concerned as a security issue, such as *"android.intent.action.DIAL"* which only provides dialing function from the system dialer without permission-protected. On the other hand, if this dialing function cooperates with other behaviors, such as uploading, phoning, or messaging to transmit data to outside, this synergism will be concerned as a potential security issue.

Therefore, it is acquiescent that any receivers with system intent action registered are in safety. However, there is one case that needs to be cautious, which is multiple intent actions might be registered concurrently under the same receiver. The reason for our concern is the vulnerable customized intent action might share the receiver with the system intent actions. Subsequently, the multiple-actions receiver has a risk of being invoked by other malicious apps. Thus, no matter how many intent actions are there with the protected permission inside one receiver, if there is one vulnerable registered intent action mixed with other protected ones, this receiver exists a security problem.

Accordingly, there are some key points in the static registration where some concerns need to be addressed:

- To check attribute *android:name* and confirm the receiver is registered and protected with the system broadcast.

- To check attribute *android:exported* and confirm the receiver is publically available to other apps or not.

- To check attribute *android:permission* and confirm the receiver is written and protected with the relevant permission.

- To check attribute *android:priority* in the *filter-intent* and make sure it has the highest number or not.

Owing to the special structural of the XML language, the semantic analysis is more appropriate inasmuch as it solves the receiver leakage thoroughly in the static registration. Also, there are many open source tools available can be explored to extract the data from a XML file. Thus the semantic analysis on the Broadcastreceiver can be efficient and accurate.

As a result, the mapping relationship between vulnerable and safe receivers in the static registration can be accurately defined by the semantic analysis. Furthermore, the potential vulnerable receiver can be efficiently filtered from amount of Android decompiled applications.

3.1.2 Dynamic BroadcastReceiver Registration

The dynamically Broadcastreceiver is in the java source code to realize the broadcast mechanism using *onReceive()*. This mechanism is managed by the Activity Manager Service (AMS) on the second framework layer in the Android architecture. As the dynamic registration is mentioned in the Android developer's guider[14], a dynamic Broadcastreceiver, as a inner class, relies on other components' lifecycle. There is an an Activity class example named *aTest*, as shown in List.3.2. The *aTest* class needs to create an object myBroadcastReceiver and *intentFilter* via instancing BroadcastReceiver and IntentFilter classes in the line 2 and the line 12 respectively. After necessarily adding the associative action name value using method *filter.addAction()* in the line 13, the process of the dynamic receiver registration will be generally done with method *registerReiceiver()* in this Activity component. Compared with the static registration, there is an additional note is to unregister the dynamic receiver with method *unregisterReceiver()*, such as the code snippet in line 19, when the dynamic receiver is registered in the *Activity.onResume()*.

Listing 3.2: An Example for the Dynamic Broadcastreceiver Registration

```
1 public class aTest extends Activity {
2     private BroadcastReceiver myBroadcastReceiver =
3         new BroadcastReceiver( {
4             @Override
5             public void onReceive(...) {
6                 ...
```

```
7                 }
8             });
9         public void onResume() {
10            super.onResume();
11            ....
12            IntentFilter intentFilter = new IntentFilter();
13            intentFilter.addAction("a.test");
14            this.registerReceiver(myBroadcastReceiver, intentFilter);
15        }
16         public void onPause() {
17            super.onPause();
18            ...
19            unregisterReceiver(myBroadcastReceiver);
20        }
21    ...
22 }
```

In this case, the dynamic Broadcastreceiver registration can be hidden in the source code when users are implementing the semantic detection. Meanwhile, to be noticed is that the *registerReiceiver()* has two types of usage; which are *registerReceiver(BroadcastReceiver, IntentFilter)* and *registerReceiver(BroadcastReceiver, IntentFilter, String, android.os.Handler)*. The first approach of the registration is preferred by most developers, due in simple format. The latter with the permission-protected, however, is the best option for the security concern, because the *String* attribute inside represents the permission option.

In comparison with the static *Broadcastreceiver*, the dynamic registration acquiescently has a higher priority in the Android platform. In other words, the dynamic registration is faster to receive a broadcast, when both the dynamic and the static register synchronously for an same event. Meanwhile, most of the dynamic broadcast receivers are embedded in other components, such as binding in the Activity and Service. As a result, once these components are stopped or destroyed by users, the binding receiver will be invalid simultaneously.

Meantime, the documentation of the system *Intent* action for Java classes in the Android SDK contains a list of static data members, and dozens of them are prefixed with *"ACTION_"*. Specifically, the static member in the Intent class does not need to be registered as a String format, but not all Intent actions are defined in the Android SDK as listed on the *Intent* class. Therefore, for the sake of the convenient invocation, the dynamic broadcast receiver registration provides the system broadcast action names for the direct Java input as static data members in the *Intent* class, such as *"ACTION_TIMEZONE_CHANGED"*, which is different from *"android.intent.action.TIMEZONE_CHANGED"* in the static registration as only one

input option. All of them demonstrate the timezone has changed, whereas only the dynamic registration can use both input forms.

The value assignment in the dynamic broadcast receiver registration is flexible, for example, there are at least two ways to assign the value to *android:name* in the *action* attribute, which are *addAction()* and *IntentFlter()*. Thus, the dynamic registration is harder to apply the semantic analysis to filter this flexible registration, compared with the static registration in an standard XML tree structure. Nevertheless, the dynamic receiver registration still has some keywords that are semantically in regard to the *IntentFilter* method invoking and the action adding, e.g. *registerReiceiver*, *addAction()*, and *IntentFIter()*, which occur frequently, but do uniquely cover the dynamic receiver registration in the text. In this specific condition, the most frequently occurring keywords will be generated as a dictionary to check the dynamic broadcast receiver registration based on the regular regression, whereas the registration pattern through the regular expression in the testing text is actually closer to a greedy detection because of the changeable value assignment in the dynamic registration.

3.2 Taint Analysis in Broadcastreceiver

Android is a component-based platform. There is no *main* method for an object-oriented Android application. Instead, an Android application consists of four main components, and each of them can have its own entry point. To coordinate various interaction between the platform and applications, there are various callback methods in the Android system, and all of them are through from the platform methods to the specific application function as a view of the sequence of calls. An Android application can be viewed as a sequence of callback functions from the Android platform to the codes or implementations of these callback functions[43], as well as showed in figure 3.1. However, the callback functions in Android platform are different from others, for example Javascript, which allow a callback function to run asynchronously. Instead, Android platform will wait for a callback function to finish before invoking another one.

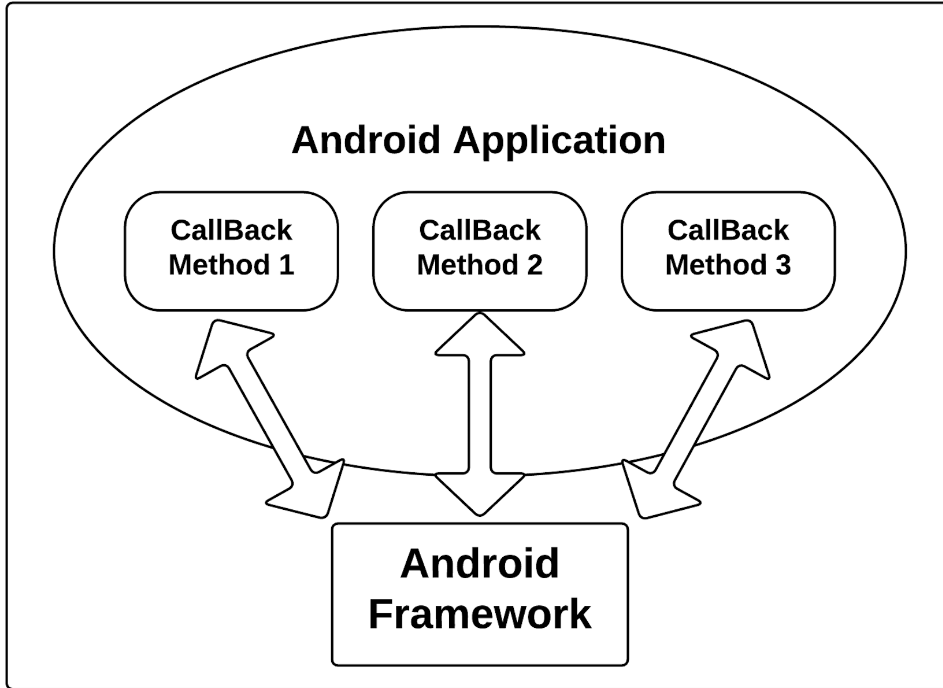


Figure 3.1: A Call Back sample in an Application

Considering inter-component communication (ICC) and callback function in the Android framework, a malware could call Android APIs due to the callback function, such as *onReceive()*. Therefore, the concept of the taint is used to determine whether permissions in Android applications are over-privileged via tacking the marked data. In other words, the over-privileged application is the one that has unnecessary permissions or has more API invoking without relevant permissions. According to the precise control-flow generation, this taint analysis in the detection system will provide a accurate result.

3.2.1 Inter-procedural data-flow Analysis

Based on the above observation, we apply the Inter-procedural Data-Flow (IDF) analysis[40] as a key step to examine the data-flow of the methods of Broadcastreceivers in an Android application. In fact, the precise data-flow graph analysis is also an; inter-procedural, finite, distributive, and subset (IFDS) problem[4], where the dataflow facts considered are finite and the dataflow functions considered are distributive, and as a result, all and only valid execution paths can be discovered in a timely manner. In an IDF graph, nodes correlate to statements, and intra-procedural

edges display the direction inside the whole flow. The standard IDF graph for a procedure P has an exclusive start node Sp and a specialized exit node Ep.

During a procedure P, data can be passed between components in Android platform. Besides this, each of start and exit nodes can be regarded as interactions with two other nodes: a call-site node Cp from the Android platform triggering P and a return-site node Rp from the platform corresponding to Cp. In consequence, there is an inter-procedural edge for the full path, $Cp \rightarrow Sp$, $Ep \rightarrow Rp$. In other words, a DFG path P is a full callback procedure representation in Android platform, and it is activated by the call-site node Cp, and starts from Sp; the corresponding exit node Ep will send the data out via the return-site node Rp. Specifically, the node Sp is the entry of the *main* method while the inter-procedural edges are matched. As one of callback methods in the Android platform, the *Broadcastreceiver* carries the interaction between an application and the platform. In either dynamic or static registration, the receiver is managed by the Android platform to override the method *BroadcastReceiver.onReceive()* when an event occurs.

In detail, a complete life-cycle process of the *Broadcastreceiver* callback is initiated from the platform method *OnReceive()* to the end of the destroy method *OnDestroy()*, which is around ten seconds or less without considering the Broadcast Queue in the background time. On the other hand, if the execution of the *onReceive()* method does not finish within a short period of time allowed in Android, the Android platform will alert users and reply the unresponsive information. Owing to the fact that the object of the broadcast receiver will be destroyed immediately after executing *onReceive()* method, the main implementation of the broadcast receiver is always applied to the event listening, triggering and notifying.

Fig 3.2 is an example of an implementation of a Broadcastreceiver. Class *main* defines Broadcast Receiver: an application component responsible for requesting callback event and creating other events. After requesting a broadcast intent, the receiver will invoke *onReceive (Context context, Intent intent)*. Method *onReceive()* is an example of a life-cycle callback method; it is triggered by the Android platform when the instantiated receiver has been registered already.

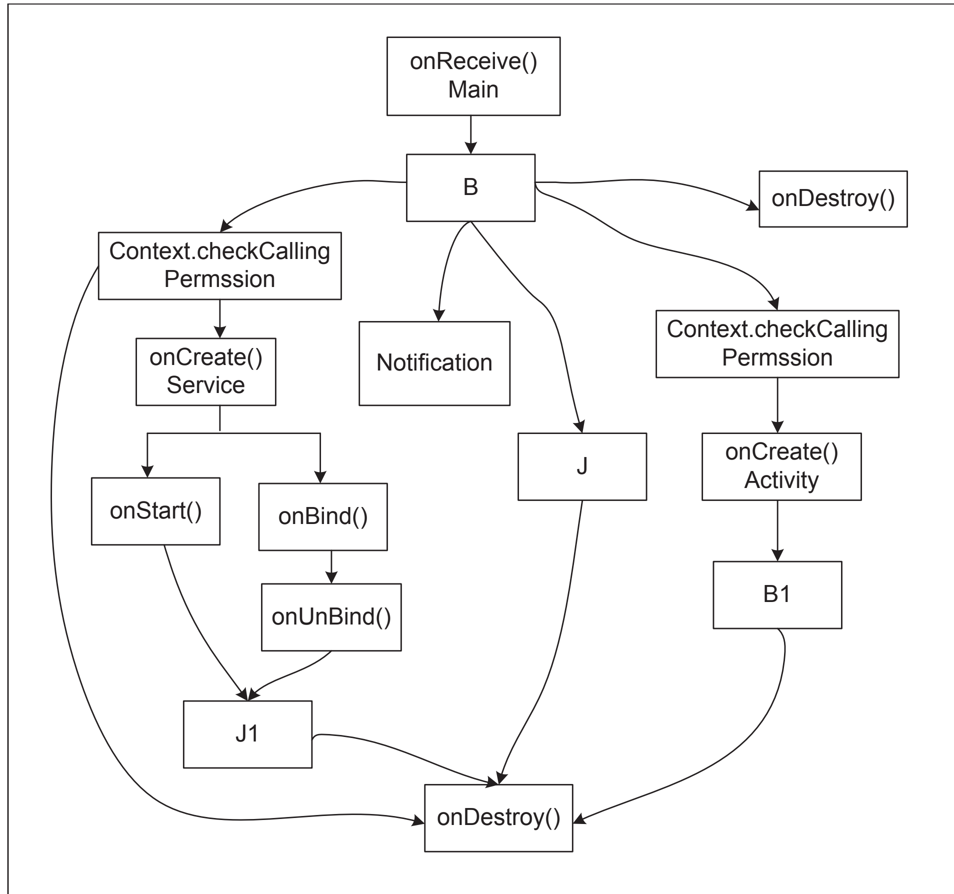


Figure 3.2: Flow Chart in a Broadcastreceiver

The structure of the receiver is defined with the receiver's name, and the action's name with additional permissions, through either dynamic *registerReceiver()* method in the Java class or static registration in the *AndroidManifest.xml*. Furthermore, the *context.checkCallingPermission()* is another callback method to check whether the receiver has the relevant permission to trigger other components or not, such as Activity and Service. Also, in this example, B is a switch node to provide multiple execution paths: start an *Activity*, start a *Service*, trigger some APIs (J), give a notification and be destroyed directly. Among these paths, both *J1* and *B1* are the nodes to represent unknown event triggering, functioning as to activate another new *Activity* or a new broadcast. Regardless the number of events of being triggered during the life-cycle of a broadcast receiver, all of them eventually are being destroyed.

3.2.2 Four Paths (*4P*) Policy

The purpose of the IDF analysis is to conclude the set of all valid paths through the receiver. Some of these abstraction paths are actually implemented. As its essence, by all means, the collection of all the valid paths needs to be abstracted and explored, where the concept of the precise detection equals to "meet-over-all-path"[18].

The inter-procedural data-flow analysis investigates the relevance between the callers and callees. Thanks to the fact that context-sensitive analysis cannot distinguish the distinct call from the caller, the set of the path collection provides a solution to generate a complicated Android control flow involved multiple-classes in source codes.

Here, we take the Broadcastreceiver as an independent module, and the data is going to flow through it from the input to the output. According to the tabulation method of functional approach, if one or more calls to a procedure P have the same data-flow value X to the entry of the P, the identical data reaches the exit of the procedure, denoted by Y. As a result, the (X, Y) will be a pair of the input-output values, where X stands for the data to the entry of the Broadcastreceiver component and Y is the corresponding data flow value reaching the exit of the Broadcastreceiver. Also, X has two basic divisions; the external source data in the Broadcastreceiver is collected from elsewhere (X_1), and the internal source data in the Broadcastreceiver is generated from the receiver itself (X_2). In addition, Y has two main directions; to send the data out directly (Y_1), for example, send out a SMS message, and to send the data to other components (Y_2), for example, start an Activity. However, the Y_2's data direction might contain multi-components activation, such as passing data to a new activity and then activate another service to send the data outside. In such cases, data remains unsafe because it still can be leak out eventually after passing through the receiver to other vulnerable components. According to the value set of the (X, Y), there are four paths based on the data-flow of the inter-component communication (ICC) mechanism, as well as displayed in figure 3.3.

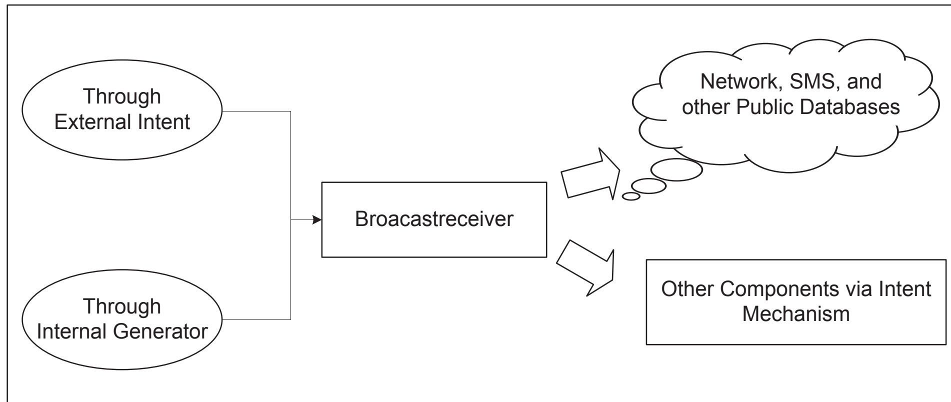


Figure 3.3: Four Data-Flow Paths for a Broadcastreceiver

Due to the characteristic of the privilege separation in the Android platform, if a sensitive event has been triggered without the relevant permission, we can assume this behavior is a vulnerability from a unknown trigger. Hence, the target broadcast receivers we are interested are those that we have found vulnerable having regarding to improperly setting permissions (or filters) on the broadcast receivers and intents. Then we perform IDF analysis on those vulnerable broadcast receivers, and identify all the valid paths through the receivers, each path corresponding to a potential attack. It is worth pointing out that the path of the Y_2 direction is considered as a threat, although the data flow is not traced and the data-flow-based tainted analysis is not performed beyond the vulnerable receiver. In doing so, we do not have to analyze the whole application, which could be time consuming and even infeasible if the size of the application is large, and instead, are focused on unprotected receivers. It significantly improves efficiency in broadcast receiver's vulnerability analysis and simplifies vulnerability assessment on Broadcastreceiver components. Therefore, the four abstraction paths; P_1, P_2, P_3, and P_4 as following will be our concern:

P_1 $(X_1 \rightarrow Y_1)$

The source data will be transmitted by the Intent from other components to the vulnerable receiver, and then send to the outside of the receiver via triggering the related and permission-protected Android APIs, such as uploading data to the Internet and messaging data to other devices.

P_2 $(X_1 \rightarrow Y_2)$

The source data will be transmitted by the Intent from components to the vulnerable receiver, and then be transferred to the rest of other components.

P_3 *($X_2 \to Y_1$)*

The source data will be generated by an vulnerable receiver internally, and this receiver will be triggered by other component. In the end, the data will be intentionally sent outside directly.

P_4 *($X_2 \to Y_2$)*

The source data will be generated by an vulnerable receiver internally, and this receiver will be triggered by other component. Consequently, the data will be transmitted to the other components through the Intent.

The *4P* policy in the taint analysis only focuses on the Broadcastreceiver component, and classifies different behaviors from the data-flow. It builds a relationship between receivers' data collection and assignment.

Also, the abstraction paths can be used to classify the attack behavior from actively and passively based on the input X. From the paths' classification, the $P_1$and P_2 are under the input X_1, while the P_3 and P_4 start from the X_2. Obviously, the path begins from the X_1 is to send the data actively to the receiver. Videlicet, the P_1 and P_2 are belonged to the actively attack behavior. In contrast, the attack path of the P_3 and P_4 that comes from the X_2 is passively waiting for other components' triggered. Therefore, the data-flow for the $X_1 \to Y_{1,2}$ is active and the $X_2 \to Y_{1,2}$ is passive.

3.3 Concluding Remarks

As a conclusion, to detect vulnerabilities of receivers, we applied two analysis techniques which are semantic analysis and taint analysis. After semantic analysis in the Broadcastreceiver registration, suspicious vulnerable receivers are identified in an Android application. Afterwards, according to different types of threats and attacks against receivers, these suspicious receivers have to go through another thorough analysis, and are further evaluated to validate vulnerabilities using different taint analysis criteria.

On the one hand, it is very challenging to confirm the caller of the receiver or the source which triggers the receiver. If a potentially vulnerable receiver in an Android application is passively involved in an attack, for example, being a callee or called by another malware, the potential risks of the receiver can be identified by using taint analysis according to our *4P* policy. Consequently, we can make sure this receiver is influential in the local application or not. If this receiver does not actively participate

in communication in any forms, it is not a threat. In other words, we can conclude that the receiver is not a risk or invulnerable. However, if the receiver plays the role of passing data, it will be assumed as a high-risk receiver as a callee. On the other hand, if the receiver that is flagged as being suspicious is a caller, for example, to actively collect data, we can assume it is vulnerable since it is very likely that it is a malicious spyware that is designed purposely.

Most of existing security tools do not analyze the receiver as a callee that could be a pertinence attack. The proposed two-step analysis presents a new approach to finding passively vulnerable receivers. Particularly, the definition of active and passive receiver in the *4P* policy is to identify the explicit callers and implicit callees in the receiver. As a result, the *4P* policy has ability to find a hidden vulnerable receiver, which makes vulnerability detection more accurate.

Chapter 4

The BRVD Framework

In this chapter, we will present a Broadcast Receiver Vulnerability Detection (BRVD) that we design to detect vulnerable receivers in Android applications. There are two main parts in this system including semantic analysis module, named SUKE, and taint analysis module, named BETA. As shown in the figure 4.1, the input is an Android APK file, while the SUKE module provides a function combined both decompiling and filtering. The APK file will be decompiled into original source files consisted of XML files and Java files, and vulnerable registered receivers, as a detecting result of the SUKE, will be filtered from these files based on semantic analysis. Then, the BETA module will give data-flow results for each vulnerable receiver based on taint analysis from the output of the SUKE. Finally, we will further analyze the data-flow results, and identity the vulnerabilities in this application.

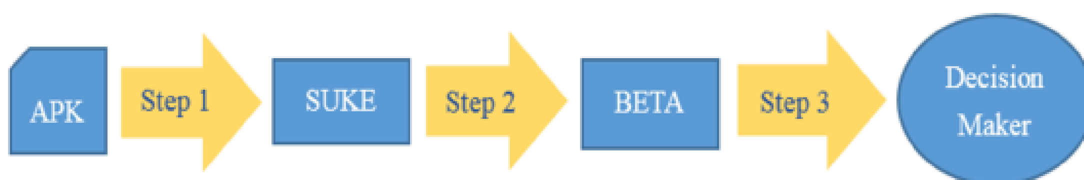


Figure 4.1: BRVD Framework

Specifically, the semantic analysis module SUKE is used to decompile APK files applying reverse engineering tools, and collect all the registered Broadcast Receivers in source files. Through the core engine *Abroster* in the SUKE we design and develop, the suspicious vulnerable receivers can be filtered from all registered receivers in the end. The *Abroster* is a Java-based tool that is utilized to receive and analyze

XML files and Java files. It can find all the vulnerable registered receivers based on semantic analysis. As a result, the SUKE will output data log, including the number of receivers, the value of receiver attributes and interrelated file names, etc. Then, the data log will be transmitted to the BETA module for a further inductive classification. It has two main results: insignificant threat and real threat. Additionally, the real vulnerability will be divided into four different possible attack paths based on the view of *4P* policy we introduced in the Chapter 3.2.2.

By confirming each broadcast receiver from the SUKE result, the BETA module will generate a data-flow graph to analyze suspicious receivers based on taint analysis, which will be detailedly introduced in next sections. Eventually, the final evaluation result of the testing APK will be displayed on the decision maker from the accurate data-flow analysis.

4.1 Semantic Analysis–SUKE Framework

To analysis the real Android Market application whose source code is not usually for the customer, the SUKE module, as a tool package, supports the input of APK form. Because an APK file with Dalvik bytecodes, which is introduced in section 2.3.1, is hard to be directly analyzed, the SUKE module provides reverse engineering tools to decompile APK files into XML files and Java files.

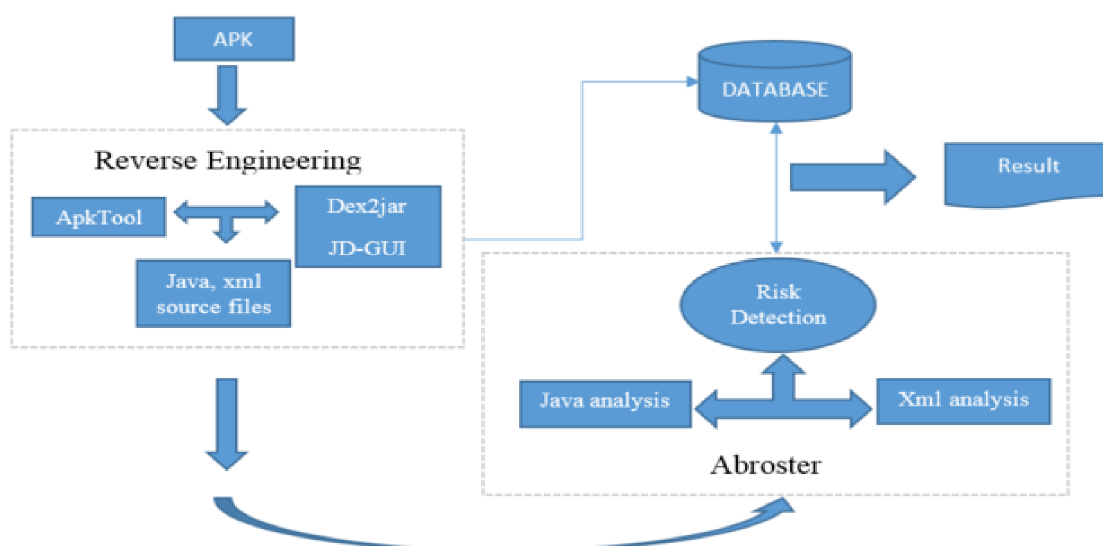


Figure 4.2: SUKE Framework

There are three main components in the SUKE module as showed in Figure 4.2; Reverse Engineering as a preprocess, *Abroster* as a core engine and Database as a data storage which is applied to save the decompiled source files in the SUKE module.

* Reverse Engineering

  Reverse Engineering is a preprocess in the SUKE that uses three open source tools as a decompling platform to automatically generate source codes from APK files. Thereinto, Dex2jar[32] and JD-GUI[33] are applied for Java decompiling, while ApkTool[31] is for XML process.

  - Dex2jar, as a famous open source tool, is used to decompile Dalvik bytecodes in an APK file into a Java jar file directly, and JD-GUI is an open tool to read jar files.
  - ApkTool is a proper parser for the XML programming language decompilation. It is implemented to extract XML file, especially the *manifest.xml* file that contains the setting of all component permissions in Android applications.

* *Abroster*

  Abroster is a reusable filtering engine we designed for SUKE, consisted of XML analysis, Java analysis and Risk detection. It is based on the semantic analysis to find keywords from both XML files and Java files, and will process a testing result to the risk detection. This engine will be fully introduced in next subsections.

The result of SUKE contains source codes of decompiled APKs from the Database and vulnerbale receivers' information from *Abroster*. Currently, all described components in the SUKE are implemented automatically, except the result transmission between SUKE and BETA. This transmitting data is the result of *Abroster* that is performed manually.

4.1.1 *Abroster* Engine

Abroster is a reliable engine in the SUKE module that processes XML files and Java files. Also, it is a Java-based tool that has a good compatibility for Broadcast Receivers' filtering. This means it can be designed as a extension tool in any Android security detection systems for users, developers and researchers in the the Android ecosystem.

In consideration of different input types, *Abroster* is designed for two interfaces to separately process XML and Java source files. Due to the difference of the registration for dynamic and static receivers, this engine also applies two filtering mechanisms based on semantic analysis to find registered receivers in different approaches. Thus, the two main parts in the *Abroster* are XML analysis and Java analysis as following:

- XML analysis is one language detection mechanism in the *Abroster*, which can read the input XML file using *DOM*[15]. Because of the tree structure in the XML language, DOM provides a function to query the *receiver* tag to extract child nodes and relevant attributes' value in the static Broadcastreceiver registration.

- Java analysis is a detection mechanism for Java programming language in *Abroster*. It is used to examine Java source files from the decompiled Android APK, implementing *Regex* [1] to find the unique text format in the dynamic Broadcastreceiver registration, such as *addAction()* and *IntentFilter()*. According to the key text format searching, the number of receivers and their attributes in Java files will be generated as a result.

In view of the different system broadcast registration formats, which was introduced in Section 3.1.2, we also designed a System Broadcast Library(SBL) for *Abroster* to test different format inputs. For example, both of the *android.intent.action.BATTERY_CHANGED* and *intent.ACTION_BATTERY_CHANGED* represent the same system event for the Android phones' battery state. However, the first expression is the only one format for static registration when a receiver wants to listen the battery state, but dynamic registration can use both of them. *Abroster* considers these situations for the different receiver's registration. Besides this consideration, new system broadcasts can be easily added in *Abroster's* SBL.

As a third part of the engine in the SUKE, the Risk Detection receives the processing result from two mechanism analysis, and gives an integrated result in the end of *Abroster* process. This final result records the specific description of vulnerable receivers, including receivers' name, attributes and class files' locations.

4.1.2 Algorithms in *Abroster*

Identifying and filtering sources of XML and Java files is challenging at the same time after reverse engineering. To avoid this, we provide two correlative algorithms in *Abroster* based on semantic analysis for XML analysis and Java analysis.

Table 4.1: Symbol Table

Symbol	Definition
W	Warning Level
E	Value of *android:exported*
P	Value of *android:permission*
R	Value of *android:priority*
A	Value of *android:name*
S_{Action}	A Set of System Broadcasts in the SBL

Assuming the value of W is the warning level, while S refers a set of the system broadcasts in the SBL. Furthermore, the value E of the *android:exported* attribute is under the *receiver* tag, as well as the *android:permission* P. In addition, R and A are hypothesized as value of *android:priority* and *android:name* respectively, which are under the tag of the *intent-filter*. Specifically, the A is the attribute of the *action* element, while R is located in the *intent-filter*. The specific reference is in the Table 4.1, and the Algorithm of XML Analysis is shown as follows:

Algorithm 1: XML Analysis

Input : XML Source Files
Output: Vulnerable Receivers

1: Int $W = 0$;
2: Boolean E;
3: **if** $A \in S_{Action}$ **then**
4:   **return** W;
5: **else if** $E = False$ **then**
6:   **return** W;
7: **else**
8:   **while** $E = \emptyset \parallel E = Ture$ **do**
9:     $R \in (-1000, 1000)$
10:     **if** $R = \emptyset$ && $P = \emptyset$ **then**
11:       **return** W;
12:     **else if** $R \neq \emptyset$ && $P = \emptyset$ **then**
13:       **if** $R > 998$ **then**
14:         **return** W;
15:       **else**
16:         **return** $W++$;
17:       **end if**
18:     **else if** $R \neq \emptyset$ && $P \neq \emptyset$ **then**
19:       **return** W;
20:     **else if** $R = \emptyset$ && $P = \emptyset$ **then**
21:       **return** $W++$;
22:     **end if**
23:   **end while**
24: **end if**

Depending on the type of the input, *Abroster* will automatically choose the related file on the corresponding algorithm. Most of Receivers registered with system broadcasts have relevant protected permissions, and all of them can be regard as safe in the Android platform. Therefore, *Abroster* will firstly extract the value of *android:name* from a decomplied APK file to compare with the existing system broadcast in the

SBL: If the value of *android:name* is not included in the SBL in the static receiver registration from XML files, *Abroster* will apply *iterator()* in the XML format to find the value *android:priority*, *android:permission* and *android:exported* respectively. On the other hand, if the value of *android:name* is not matching from the SBL in the dynamic receiver registration from Java files, *Abroster* will implement the greedy matching for calculating all the number of registered receivers. Hence, the Algorithm of Java Analysis is displayed as following:

Algorithm 2: Java Analysis

Input : Java Source Files
Output: Vulnerable Receivers

1: $NumberOfReceivers = GetAllRegisteredReceiver()$
2: $NumberOfSystemActions = GetAllSystemAction()$
3: **if** $NumberOfReceivers > NumberOfSystemActions$ **then**
4: **return** $W++$;
5: **else if** $NumberOfReceivers = NumberOfSystemActions$ **then**
6: **return** W;
7: **else if** $NumberOfReceivers < NumberOfSystemActions$ **then**
8: Erro;
9: **end if**

As a conclusion, *Abroster* in SUKE module is used to receive decompiled source files from the reverse engineering component, and generates suspicious receivers as a testing result. This testing result of *Abroster* combined with decompiled source files will be the final result of SUKE module. Meanwhile, the output of the SUKE module will become the input of the BETA module.

4.2 Taint Analysis–BETA Framework

In order to figure out the specific types of attacks and perform the data verification, we designed BETA module in the BRVD to receive and analyze the data flow based on the suspicious *Broadcastreceiver*. After filtering general apk files, the rest of apks will be assumed as threatening apps. However, the SUKE module is not accurate for the attack detection. For example, the receiver would not influence any other methods inside the application, such as to simply monitor the system power and release the notification, but it is still considered as a vulnerable origin by the SUKE module. We will not regard this receiver as a concerning threat for the whole apk. In other words, the scenery we should be concerned with is that the target receiver would not influence

other normal components' running when it is being attacked by malicious apps. We named it as an insignificant receiver and the security warning result from the SUKE is a false positive. Nevertheless, the discovery of the vulnerable insignificant receiver also should create an awareness of the security concern that needs to be enhanced. It is meaningful for the Android ecosystem to build a better, and a more healthy environment.

Therefore, the existence of the BETA is to distinguish the threat receiver in the rest of vulnerable apks, and put them into different classifications. In the BETA, we regard any methods which could create the data flow into the vulnerable *Broadcastreceiver* as a "source", while the method which is triggered by its receiver and to send the data out will be defined as a "sink".

An algorithm for the accuracy of inter-procedural data-flow analysis should meet the requirement of the "*meet-over-all-valid-paths*". This means the path is valid if it accurately shows the procedure of the data-flow. Here, we take the *Broadcastreceiver* as an independent module and the data is going to flow through it from the import to the out-port. In our taint analysis, the source as data input, and the sink as data output, keep activating during this whole running process of the module. Since the taint analysis is the core concept in the BETA, we have provided data-flow analysis to the current attack based on the Four Routing policies P_{1-4} in Chapter 3.

4.2.1 Tools used

The input of Beta consumes an Android application package (*.apk*). It retrieves the package and translates the Dalvik byte-code into an Intermediate Representation (IR). The front end of Beta starts the IR generation process by parsing the input byte-code file, while the back end analyses the IR based on the taint analysis. Beta applies an open source Dalvik byte-code parser named Soot[20], part of a well-known java optimized tool for Android apps. Soot has powerful interfaces to process code from Android byte-code and analyze the inter-procedural data-flow. Specifically, a Soot-based taint analysis tool, *Flowdroid*[2], is our current key engine for the BETA module. In general, BETA adopted from two main modules, *Flowdroid* provides the static taint analysis for Android application and SUSI[38] brings a fairly comprehensive list for the Android's sources and sinks.

Flowdroid is built on the Soot platform and Heros[4] that gives taint analysis for Android applications. It is provided by the European Center for Security and Privacy by Design (EC SPRIDE). We choose the *Flowdroid* for following reasons:

- Provided a well-defined semantic proximity between Dalvik bytecode and the IR by Soot community. It has three IR formats; Bat, Jimple, and shrimple, based on which the back end analyzers carry out their tasks.

- Implemented complete callback functions and each of components' lifecycles in the Android application. Therefore, the broadcast receiver as an independent component will be our research target in the BETA detection system.

- Achieved 93% recall as a higher precision Soot-based tool currently, comparing with other similar data-flow analysis tools, such as JOANA[26] which is WALA-based tool. Especially for the lack of *main* method in the Android application, the *Flowdroid* is the one will automatically generate a dummy *main* method and supplies a completely method analysis.

- Offered to be configured to output the data-flow path for every source-to-sink connection separately. Accordingly, the multi-paths may display at the same time, and it caters for our data-flow routing P_{1-4} definition.

In our view, the tool selection is according to the requirement of fast and accuracy, so the *Flowdroid* is the one that meets all these conditions. More detailed information related *Flowdroid* and the download links can be found in its official website: *http://sseblog.ec-spride.de/tools/flowdroid/*.

In addition, Beta realizes the data-flow path and models discussed before. It detects four different types of the attack based on the Routing P_{1-4}, with a set of specific 34 sources and sinks that match the data-flow defined by paths. The whole library of set was constructed by SUSI to cover a relatively wide range of Android's sources and sinks, but we concern on the specific part that only affect the Broadcastreceiver component. Hence, the biggest reason for why we choose SUSI is that it provides a powerful category list of sources and sinks. For example, there is a source category we defined "*Broadcast Receiving*" grouping all methods relevant to the *getBroacast()*. According to the categorized sources and sinks, we can perform a demand-driven vetting of *Broadcastreceiver* component in Android apps.

The overall principle for the data-flow analysis is to check if there is a potential flow between source and sink. Therefore, SUSI not only gives us a highly accurate list of found sources and sinks, but also a completely list of categories. Besides this, as a machine-learning tool, SUSI can be extended the definite source and sink to vet the individual component from apps, so that the Beta can capture the data-flow from an

independent Broadcastreceiver component. More information concerning SUSI can be found in its official website, *http://sseblog.ec-spride.de/tools/susi/*.

4.2.2 Taint Problem Setup

In Android applications, there are many APIs that extract and collect users' information. To protect the user's privacy data in the Android phone from the malicious application's calling, most of main privacy APIs will be tracked and monitored.

Privacy	APIs
Calendar	*java.util.Calendar*
Browser History and Bookmarks	*android.provider.Browser*
User Account	*android.accounts.AccountManager*
Microphone	*android.media.AudioRecord*
System Log	*android.util.Log* *android.os.Handler*
SMS	*android.os.Handler* *android.telephony.SmsManager*
Network State	*android.net.wifi.WifiInfo* *android.bluetooth.BluetoothAdapter*
Application Information	*android.app.PendingIntent* *android.content.pm.PackageManager*
Location Information	*android.location.Location* *android.location.LocationManager* *android.telephony.gsm.GsmCellLocation*
URL	*java.net.URL* *java.net.URLConnection* *org.apache.http.util.EntityUtils* *org.apache.http.HttpResponse*
User Profiles (Including User Contacts)	*java.io.Writer* *java.io.Reader* *android.database.Cursor* *android.content.ContentResolver* *android.content.SharedPreferences* *android.database.sqlite.SQLiteDatabase*

Table 4.2: Privacy APIs

The privacy API through the suspectable broadcast receiver will be tainted in the BETA. Meanwhile, each API in the list contains at least one method. For instance, the *getLocation()* method in the *android.location.Location* is one of the ways to query

users' location information, and there are other methods that can do the same function. As a result, the number of specific methods are much more than the number of associated APIs. Hence, to avoid the implementation details with the specific method list out of the scope in this thesis, and also generate a wider kinds of privacy APIs, we list main source APIs as Source Methods (*Sm*) in table 4.2, consisted around 100 tainted methods in total, and the Sm has been updated in the BETA library.

In the current version, BETA in the BRVD marks the input-output value set of (X, Y). Specifically, X_1 and X_2 are two main sources in the set of X, while Y has Y_1 and Y_2 two different output methods. However, all of paths in the BETA of BRVD are according to the $X \rightarrow Y$ direction. Pursuant to different carriers of *Sm*, X_1 and X_2 are regarded as two source carriers in the taint analysis, but different methods' utilization.

During the period of the X_1 carrier, after filtering in the SUKE module, any source data Sm will be transmitted to the *Broadcastreceiver* component via the Intent mechanism, will be assumed as tainted sources in the Beta module. As a result, we collect a set of intent methods X_1 and categorize them as a category "*Broadcast Intent Receiving*" grouping all sources related to data transmission between receiver and other components. To do so, the carrier X_1 will be a tunnel of the *Sm* as tainted sources, which contains *PendingIntent()* method, *Intent* content sources, and *Bundle* sources as displayed in the Table 4.3.

APIs	Descriptions
android.os.Bundle	Input data via Bundle methods
android.content.Intent	Input data via Intent Mechanism
android.app.PendingIntent	Input data via *PendingIntent()*

Table 4.3: Source Methods in X_1

In detail, although all of them are under different APIs, they are using a same approach for the data communication because the broadcast intent is an information carrier. In other words, the broadcast intent is the input of the receiver which can access and trigger the receiver activating. To obtain the broadcast intent from the receiver, the *getBroadcast()* and its related content showed in the Table 4.3 will be the main detecting part in the BETA.

Additionally, the *Bundle* in the Android is used for data transformation between components. Besides carrying Sm, it could also hold varies of data parameter, such as *String* and *Int*, when an component crushed. For example, the screen rotation will

lead to the Activity destroyed, later another instance of the activity is created. So if there is a receiver is registered under the activity, the bundle data will be held until the broadcast receiver is re-create again. For this temporary data saving point, the *android.os.Bundle* is considered as tainted data in X_1 category as well.

On the other hand, according to the types of carrier category, the X_2 is a category Source Generating categorizing all the Sm from the *Broadcastreceiver* component. Compare with the X_1, the X_2 is using receiver as its data carrier originally without outside data injecting. In order to detect the data leakage in the X_2 path, we perform all of privacy data methods Sm as sources that will be triggered in the relevant receiver. Namely, for the sake of identifying the Sm method from being called directly by the receiver, the list of Sm has been connected to the X_2 as sources. As a result, to initialize the taint analyzing from the X_2 as an entry point, in addition to receiving the broadcast Intent from other components' triggering, the *Sm* should also be called by the vulnerable receiver class at the same time.

Therefore, the Beta provides a visual routing to prove the data-flow where the privacy method Sm has been called eventually. In our vetting system, the two different carrier X_1 and X_2 are the source of the taint analysis.

Since the source X_1 and X_2 are tracked, the set of sink Y is the method that brings the data out of the receiver. It has two different directions within data; using functional APIs directly (Y_1), and activating other components (Y_2). e.g. the first direction (Y_1) in the Y could be the data leakage straightforwardly of the SMS sending method. Whats more, the other direction (Y_2) in the set of Y could utilize *startActivity(intent)* to start another activity with the *intent.putExtra()* data package.

In the Y_1 sink method, we consider all of the directly data exists in the receiver as sinks. Particularly, the path is from the node X_1 or X_2 to the node Y_1. By the reason of the life-cycle limitation of the *Broadcastreceiver* component, only two main exits of data-flow can be implemented by the receivers activation which are Internet, SMS, and file output.

Sinks of Y_1	
File in-out-put	*java.io.OutputStream* *java.io.FileOutputStream*
Message	*android.os.Handler* *android.telephony.SmsManager*
Internet	*java.net.URL* *java.net.URLConnection* *org.apache.http.client.HttpClient* *org.apache.http.impl.client.DefaultHttpClient*

Table 4.4: Sink Methods in Y_1

From the table 4.4, it states a list of main output APIs for related functions. Identifying the inter- and intra-procedural components data-flow on the Android platform requires a call graph with proper definition of the reach-ability analysis. Once we confirm the sink of the *Sm*, an edge is drawn from the node X to the node Y. Consequently, an amount of edges through the receiver between scalar variables can be drawn with the help of the *Flowdroid* implied from the source-sink.

Similarly, the sink of Y_2 is the other data-flow exist of the broadcast receiver model. In contrast of the Y_1, Y_2 assumes all the data that pass out of the receiver will be considered as a threaten. For example, the target receiver is triggered, and sends an Intent query to start a new thread for a new Service within data transmission. It is not a simple one-way direction which is $X \rightarrow Y_1$ as we mentioned before, but a multiple-ways for $X \rightarrow Y_2 \rightarrow Y_2 ... \rightarrow Y_1$. This means the data transmission from the receiver may through more than one component till it find a exist Y_1 to pass to the outside of the phone. However, we only pick the first component activation as the data input for the Y_2. Based on the inter- and intra- component data communication, the set of Y_2 will be methods from the broadcast receiver implemented any data transportation via Intent mechanism.

Sinks of Y_2	
android.os.Bundle	Data holding unit via Bundle method
android.content.ContentResolver	Output data to Content Providers
android.content.BroadcastReceiver	Output data to Broadcast Receivers
android.content.Context	Output data via other components, including Activity and Service.

Table 4.5: Sink Methods in Y_2

Specifically, all methods in the list showed in the table 4.5 are functions to trigger other components running. In the list, *android.content.BroadcastReceiver*, *android.content.ContentResolver* and *android.content.Context* are the key to generate an Intent to start a new lifecycle of a component that comes from the Activity, Service, Broadcastreceiver and Content Providers. For example, Content Providers not only can be affected by API *android.content.ContentResolver*, but also *android.content.SharedPreferences$Editor*. Therefore, a Content Provider component, as a data storage in Android phones, can be accessed from a receiver in accordance with different methods. In case of the data leakage in the database, the content provider is a big security concern as a data sink in taint analysis.

However, a data flow F from the source X to the specific sink Y_2 might not be a completely flow in a whole function process after passing the receiver, because Y_2 could be a temporary transfer station and will pass the data to a next new component. Thus, the F could be multiple-points transportation flow until to the final destination Y_1, which carries data outside of the user's phone. It could be $X \rightarrow Y_2 \rightarrow Y_2... \rightarrow Y_1$ that considers $X \rightarrow Y_2$ as a beginning. In detail, X is the data input of the receiver, while Y_2 is the data output of the receiver. Considering the source and sink correspond to the data entrance and exit of the vulnerable broadcast receiver respectively, the data-flow F will be assumed as a short $X \rightarrow Y_2$ path, because the beginning path of the F has already been threaten by the vulnerable receiver. For the rest of the data-flow direction $Y_2 \rightarrow Y_2... \rightarrow Y_1$, we do not omit the continue tracking in this situation from concluded two reasons:

- The data tracking in the BETA is based on a single application detection. If the data communication is among several applications, the final exist Y_1 is almost impossible to be found according to the isolated application vetting method.

- Since the receiver has been confirmed as a suspected vulnerability from our first SUKE module, no matter where the final direction of the data-flow is going, the vulnerable receiver risks a potential attack from other malicious applications. In other words, the unprotected Y_2 method in the receiver can be possibility attacked and called by malwares.

As a consequence, with the $X \rightarrow Y$ flow F is generated, the BETA picks two sets of nodes, Fx and Fy, where Fx contains pre-defined sources (X_1 and X_2), and Fy contains pre-defined sinks (Y_1 and Y_2). The resulting F is built on different four data-flow edges (*4P* policy in chapter 3) from the node Fx to the node Fy. According to the different Fx and Fy, although there are many attack approaches around the

broadcast receiver, they can never get rid of the four routing principles no matter how the attack method is changed.

4.3 *4P-Based* Attacks

In this section, we elaborate on four popular attacks against Broadcast Receivers, particularly, from the view of *4P* policy. Also, we discover a new attack according to our *4P* policy.

4.3.1 Intent Spoofing Attack

The Intent Spoofing attack is a famous active attack. In the Intent mechanism, it contains Activity Intent, Service Intent, and Broadcast Intent. However, during the BETA analysis, we only concern on the security issue of the Broadcast Intent in the Android application. In other words, the BETA mainly focuses on the Spoofing attack that comes from the Broadcast receiver.

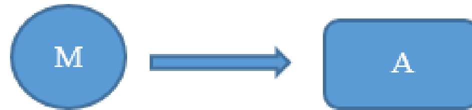


Figure 4.3: Intent Spoofing Attack

In the fig 4.3, the M as a malware is trying to call the broadcast receiver in the application A, when the receiver in the A is vulnerable and open for any applications' calling. Actually, in accordance with the receiver's registration, there are some attributes inside for protecting and controlling, such as export and priority number setting. In Google handbook[14], the default value of the *android:exported* setting in the receiver's registration is *True*. Accordingly, once the value setting of *android:exported* is null or *True*, the relevant receiver is opened as a public component for the global calling in the Android platform. Thusly, the application that has the vulnerable receiver will take the potential risk from other applications' malicious triggering.

As a consequence, considering the attribute under the receiver tag during the registration can be set as open-ended, the unprotected and public receiver is vulnerable under the Intent Spoofing attack. As an actively attack, the key of the Intent spoofing

attack utilizes the vulnerability of the receiver registration to threaten the correlated application.

Listing 4.1: VLC BroadcastReceiver

```
1 <xml> <receiver android:name=".RemoteControlClientReceiver">
2         <intent-filter>
3       <action android:name="android.intent.action.MEDIA_BUTTON"/>
4       <action android:name="org.videolan.vlc.remote.PlayPause"/>
5         <intent-filter/>
6        </receiver>
7 <xml>
```

From the listing 4.1, the above code snippet is one broadcast receiver registration that comes from the Android media application VLC[35]. In this open source application, the first action with the MEDIA_BUTTON is a system-protected action name, whereas the other customized one *org.videolan.vlc.remote.PlayPause* is the weakness and open to the global calling in the Android platform. In this example, the VLC public receiver provides an action name that is unprotected without the *android:exported* attribute. In other words, the receiver *RemoteControlClientReceiver* could be called by an unknown source with the default *android:exported* value.

Consequently, the process of the Intent Spoofing attack is that the malicious Intent from other components is going to attack the vulnerable receiver, and through the target to use its corresponding function methods for the unexpected behavior. Pursuant to data-flow F, it is from the node F_{X_1} to the node F_Y which belongs to P_1 and P_2 based on the P_{1-4} principle. Equally, the Intent Spoofing attack is to send the data to the target receiver in order to accomplish the malicious activity. Furthermore, the different Y_1 and Y_2 in the set of F_Y decide the direction way of the Intent Spoofing attack. Therefore, this attack can be divided into P_1 and P_2.

4.3.2 Intent Hijacking Attack

The way of Intent hijacking is a well-known passive attack, including Activity Hijacking, Service Hijacking and Broadcast Theft. As we can see from the Fig 4.4., the malicious receiver M is sitting in the middle of the component A and B. As stated in the current study, the Intent hijacking M can use the Intent mechanism to modify, listen or even stop the communication between A and B. However, in the BETA module, we only focus on the broadcast theft.

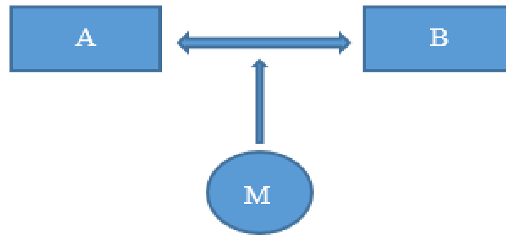


Figure 4.4: Intent Hijacking Attack

Specifically, when the implicit broadcast intent will be sent by the application A, the malicious receiver in the M and the common receiver in the B will be triggered and obtained the access to all the data via declaring intent-filters with the same name of actions, data and categories.

Both malicious receiver M and normal receiver B need to wait for component A's triggering while the user is interacting. Indeed, the passively Intent attack is similar to the phishing attack. In doing so, once the M mix the spurious with the genuine receiver B, the A might activate both M and B at the same time by the implicit *Intent*.

Listing 4.2: Malicious BroadcastReceiver

```
1 <xml> <receiver android:name=".myReceiver">
2         <intent-filter>
3       <action android:name="org.videolan.vlc.remote.PlayPause"/>
4         <intent-filter/>
5        </receiver>
6 <xml>
```

As mentioned above from the VLC media application example, if the hijacking receiver M (Listing 4.2.) has the same action name with the receiver B, when the A is sending the implicit intent for a normal requesting, both *.RemoteControlClientReceiver* and *.myReceiver* will be responding.

According to the data-flow of the Broadcast Theft, the malicious receiver is called by the other health components via Intent, and do the threaten behavior. In this general attack, it contains both data injecting from Intent and data collecting from the malicious receiver. Based on the four paths in the BETA module, the Broadcast Theft will be defined into whole R's paths depending on the data-flow. On the other hand, although the broadcast theft is similar to the Intent Spoofing attack, while one is passively attack and the other one is actively attack, the Broadcast Theft is more

complicated than the Intent Spoofing. Compared with Intent Spoofing, the Broadcast Theft attack mainly can be divided into data obtaining and data creating. For the reason of two data acquisition, the flow $Fx \rightarrow Fy$ is generated accordingly by the BETA based on various set of *(Fx,Fy)*.

4.3.3 Confused Deputy Attack

The Confused Deputy attack is an advanced Intent Spoofing attack. Nevertheless, it is a special attack that is bypass the permission to call the accompanying API.

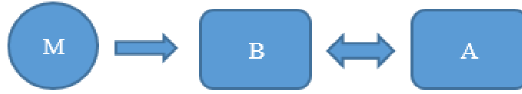


Figure 4.5: Confused Deputy Attack

In the fig 4.5., an application A could be relative security compared with other applications. All the receivers in the application A is secure, but only one data communication connected with an application B. Also, the communication tunnel between A and B is secured. However, if the receiver that triggers A's activity in the B is not setting in proper, the malicious M will call the B's vulnerable receiver to control A's behavior.

In a Confused Deputy attack (some people called permission re-delegation [9]), Service and Broadcast Receiver are the main parts which are invited to ally with stealthy permission re-delegation behaviors because they are not visible user interface at the most time. To invoke the target application, the malicious application does not need to declare the corresponding permission inside, it still can attack and access the target through controlling other connected applications.

The data-flow F in the Confused Deputy attack is similar to the Intent spoofing, but the output of the receiver is just a start for another behaviors activating. According to the P_{1_4} policy, the set of Y_2 sink is confirmed in this attack as a stable data output. As a matter of course, the edge of the flow F should be $X \rightarrow Y_2$, while X is a set contains X_1 and X_2. Therefore, the P_2 $(X_1 \rightarrow Y_2)$ and $P_4(X_2 \rightarrow Y_2)$ in the BETA module will be the path to analyze the data-flow of the Confused Deputy attack.

4.3.4 Collusion Attack

The collusion attack is neither passively nor actively, but cooperation. In the Fig 4.6., each of application A and B is completely security in the Android platform, even with the secure permission inside confuse users to download. In contrary, application A and B will be combined as a malware M eventually. The essential condition for this attack is to use the same digital signature in the APK file to recognize and cooperate with each of other functions.

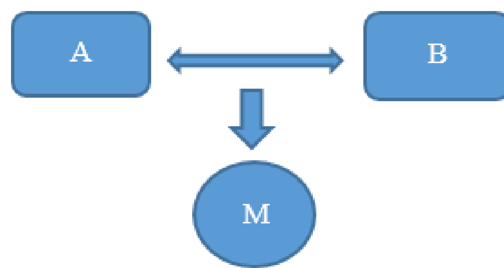


Figure 4.6: Collusion Attack

For example, a normal application A has the camera function and the other application B does the internet uploading separately. When the user is clicking the photo shooting button, the A will send a broadcast via the Intent in the background, such as the sticky broadcast. Moreover, the broadcast can be designed under the author's signature protection, which means only the signature matching receiver can receive the corresponding broadcast and couple to the appointed function. On the contrary, the hiding broadcast from the A will be vanished naturally if there is no corresponding receiver in the application B. Therefore, the application B is going to accept the related broadcast when both A and B are installing in the same phone. Once the vulnerable receiver in the B found A's purposeful broadcast, the data will flow to outside through B's receiver.

In the collusion attack for the data-flow analysis, there is an application A which is for data collecting and transmitting, while the other B is assigned to do the data receiving and sending. The truth for the malware detection is that the data collecting is really hard for detecting, because there is no clear sink Y for the source data in an isolated application. In other words, the data collecting and transmitting in the application A not only is difficult for tracking, but also is fuzzy for distinguishing malware from the healthy application.

Hence, the BETA only focuses on the receiver which is doing the data receiving (X_1) and sending (Y), instead of detecting both caller and callee. From the source-sink analysis, a flow F could be easily vomited from $X_1 \rightarrow Y$, while the set Y contains Y_1 and Y_2. Namely, the P_1 and P_2 is the path for the Collusion Attack. From a certain perspective based on the *4P* policy, Collusion Attack as a classical attack in the Android phone could be regarded as Spoofing attack under the same attack path.

However, Both Intent Spoofing attack and Broadcast Theft are summed up as single-side attack. This means the single-side is mainly based on the receiver's vulnerabilities, and its attack relationship is simple only between malware and applications. On the other hand, the multi-sides attack, such as collusion attack, are not simple as single-side attack. Malware inside is not an easy observed, and all of them are not malware if they do not do some malevolent behaviors. Literally, most of multi-sides attack are written and designed by the same author. Scilicet, the receiver inside is purposefully designed as a secure component on the appearance when the user is downloading these applications. Unfortunately, there is a special situation which is non-detectable in the SUKE filtering module. The application is well-planned and the receiver is permission-protected with the customized *android:permission* setting value. In other words, the broadcast between applications is secured by the permission. To do so, we cannot detect the "secured" receiver and regard it as a malware. In this situation, we need to propose this case as our future work for a further development.

As mentioned before, the $X_1 \rightarrow Y$ flow is embedded into the BETA medule. Similar path detection to the Intent Spoofing attack, as an active attack, the collusion attack belongs to the P_1 and P_2, except for the special case. The classical attack is re-defined by our path policy. According to different paths, we also can forecast some new attack methods.

4.3.5 Self-Evolution Attack

The Self-Evolution attack is a special attack we forecasted based on the exist attacks. As we known for the updating version in Android third party, if any applications provide the new updating version numbers, the target application will notify the user about the updating information when the Android phone is connecting to the internet.

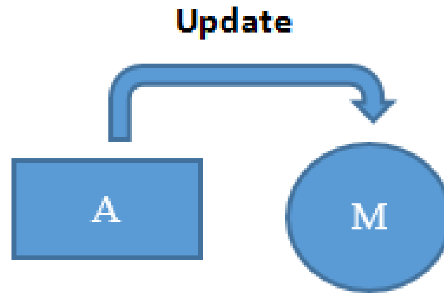


Figure 4.7: Self-Evolution Attack

From the illusion of the Fig 4.7., the application A is a secure Android app, whereas it might be a malware M after version updating by the app's author or other third party operators. The Self-Evolution attacking approach could be repacking some popular Android applications, such as "Tweetcaster" application mentioned by Joany et at[5]. They modified the several components in the "Tweetcaster" and applied a Broadcastreceiver to trigger an updating once a specific call is received.

Listing 4.3: code snippets of Android GoldGPEN Spyware in the Tweetcaster

```
1 public void onReceive(Context context, Intent intent){
2  getTeleService(context); //Access to the TelephonyManager
3  Bundle b = intent.getExtras();
4  String incommingNumber = b.getString("incoming_number");
       //Get Incoming Number
5  ...
6  // Trigger an updating action according to the Caller
7  if(incommingNumber.equals("5556")){
8  AnswerAndRejectCall();
9  Intent launchHome = new Intent(Intent.ACTION_MAIN);
10 launchHome.addCategory(Intent.CATEGORY_HOME);
11 launchHome.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
12 context.startActivity(launchHome);
13 DownloadFromUrl("CMDs.txt",context);
14 File dir = Environment.getExternalStorageDirectory();
15 File cmds_file = new File(dir, "CMDs.txt");
16 ...
17  while((line = buffreader.readLine()) != null){
18 line = line.trim();
19 list = TextUtils.split(line, "/");
20       if(list[0].compareTo("UPDATE")==0){
21       Intent update_intent = new Intent();
22       update_intent.setClassName(
             "telindus.AndroidGoldGPENSpywareFirst",
             "telindus.AndroidGoldGPENSpywareFirst.ProcessUpdate");
23     update_intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

```
24         context.startActivity(update_intent);
25           }
26         }
27     }
28 }
```

In the listing 4.3, a receiver in the "Tweetcaster" obtains incoming numbers from users in line 4 and downloads appointed text based on the specific number in line 7 and line 13. Then, this receiver will read the installed text to update and point a specific spyware in line 22 starting a new Activity in line 24. This is an interesting remote control attack sample based on the Broadcastreceiver. The vulnerable receiver is designed to wait an specific income number in this case. However, there are more approaches than this number "trick", such as to get the user's browser history to match the specific website and download some components when user is connecting Internet.

Furthermore, it is easily applying the same digital signature and providing malicious components into an updating patch. Normally, the user does not realize the difference between the old version and new version, except some new functions displayed graphically. So the non-visible *Broadcastreceiver* and *Service* will be potential suspected components for the malicious behavior after updating.

For the application A, we cannot detect it for secure problems because this could be a healthy application originally on the list of the online commercial market. On the contrary, the malware M is easily to be detected, but at that time the old version application already has been installed in the user's mobile phone.

Consequently, the flow F in the Self-Evolution attack can be any paths (P_{1-4}) depend on the patch inside. to defence this attack, we have to do periodic inspection after softwares installation, and not install unauthorized Android applications.

4.4 Concluding Remarks

In this Chapter, a Broadcast Receiver Vulnerability Detection(BRVD) system is proposed from several areas, including system design, architecture introduction and key algorithms utilization.

BRVD is written by Java and XML language combined Android SDK, Eclipse and other open-source tools. It is divided into SUKE and BETA modules, which are based on semantic analysis and taint analysis respectively. In specific, we design a core engine *Abroster* in the SUKE to analyze input files that are decompiled from

APK files, and set up the source-sink problem based on *4P* for the BETA's data-flow verification.

In addition, we introduce four classical attacks based on data-flow, along with a new attacking method against Broadcast Receivers. All of them are under the detection range of the BRVD system. In the regular data leakage attack, we nominate the *4P* policy for the data routing analysis. From the attack introduction we proposed above, the *4P* principle can be matched for the each attack as a table below:

Attacks	Mode	Paths
Intent Spoofing	Active	$P_1 \& P_2$
Broadcast Theft	Passive	P_{1-4}
Collusion Attack	Active	$P_1 \& P_2$
Confused Deputy	Active	$P_2 \& P_4$
Self-Evolution	Both	P_{1-4}

Table 4.6: Attacks based on *4P* policy

From the table 4.6, all five attacks are listed in the table with their corresponding attack modes. In our principle, we divide the attacks into active and passive modes, depending on various attack behaviors. Only Self-Evolution contains both modes because of its flexible patch modification. Notwithstanding, no matter the way how the attack is, the path category matches each of them based on the taint problem. Next, we will introduce the experimentation for the data-flow analysis.

Chapter 5

Experimental Evaluation

The Broadcast Receiver Vulnerability Detection (BRVD) is written in JAVA based on the Kepler eclipse. The total amount of code is more than 10K lines, excluding third party libraries. It consists of two main modules, namely SUKE and BETA. The two modules in the BRVD are atuomated testing based on static analysis. In addition, the *apktool*[31], *jd-gui*[33], and *dex2jar*[32] are needed for the reverse engineering, as well as the Android platform SDK.

The building process of the BRVD took us plenty of meaningful efforts due to a shortness of similar work and code. Notwithstanding, most of efforts were spent on reproducing existing open tools and testing the related algorithms inside. Certainly, it is not an intent to claim these efforts as contributions in our thesis. BRVD currently supports both Windows and Linux system. After packaging, BRVD can even be embeded in the Android platform as an app tool. However, BRVD is a prototype with single component vetting function, not including *Activity*, *Service*, and *Content Provider*.

As a front end of the BRVD, SUKE is mainly implemented with Regular expression. It consumes JAVA and XML source files separately. The key function of the SUKE is to decompile and filter the source file from the apk package. From the rest of suspected receivers, SUKE provides a strong and precise analysis function. Furthermore, it allows different analysis to choose depends on the source code.

In principle, the privacy data that risks to be tracked should be marked as tainted sources. In fact, the Intent mechanism in the Android platform is the only way for the components' communication among with *Activity*, *Service* and *Broadcastreceiver* in the Android platform. For this reason, the Intent method with the data transmission are essential for our BETA module. Subsequently, from the above description in the chapter 3, the Broadcast Intent is the key to activate the associated receiver, no matter the activation is active or passive. Inspired by the the SOOT platform

support, the problem of checking data-flows could be known as a graph reach-ability test. With the callback function of the broadcast receiver, the receiver can be taken for an isolated vetting unit between various of components in an Android application. Therefore, the BETA provides a test for the connectivity of source-sink pairs on a receiver. A source-sink pair would be displayed on the BETA that demonstrates the existence of a data-flow from the source to the sink.

In this chapter, we will apply our BRVD on Android applications including both one customized sample and real applications in the market. The goal of the experiment is to validate the effectiveness of the proposed detection system. The system is built on a 64-bit Windows operating system with Intel(R) Core(TM) i7-4770 CPU 3.40GHz and 16.0 GB RAM.

5.1 Customized Sample

The customized sample is written for testing as a hypothetical application. It states several sensitive permissions, including SMS, Network and GPS. The advantage of implementing the target is that most of key embedded points provided by the interface method can be frequently called. Compared with the inconspicuous malicious behavior in the market application, the customized sample highlights the attack method for the specific component, and a testing result has already been expected.

According to the motivation example in the Listing 1.1 in Chapter 1, the malicious behavior inside is similar to the $X_1 \rightarrow Y_1$ attack mode based on the policy $4P$. Note that the original GPS data transmission is divided as three parts from the split started by *doInBackGround*, and reach to the main activity by *HandleMessage*, then pass to the split started by Intent in the *BroadcastReceiver* class. By the reason of the asynchronous invocations of entry points, the sensitive source in the real app is hard for detection, such as the GPS obtaining within the *doInBackground()* method (Line 20) in the asynchronous thread is really difficult to be tested. Notwithstanding, the Broadcast Receiver Vulnerability Detection (BRVD) is mainly concerned on the input-output value of the independent receiver. We accordingly count the number of receiver classes in every application, and filter each of the vulnerable receiver. Ignoring all the resources collection in the Android application, our SUKE module find the vulnerable receiver class which is weakness for the public invocation in the target application. From the filtering log, it points out the specific receiver name as well as the corresponding file location in Figure 5.1.

```
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/AndroidManifest.xml
Important Status: true
Action name: MyBroadcast
Android name: MyBroadCastReceiver
Export:
Priority:
Permission:
Matched results:
------------------------------------
total dynamic registration:0
There is no dynamic receiver registration
```

Figure 5.1: Filtering Log

Because *MyBroadCastReceiver* is unprotected and open to public component invocation, the specific application is marked with the explicit receiver class in the SUKE. In accordance with the BRVD, the BETA is implemented for P_{1-4} path classification after filtering.

```
Found a flow to sink virtualinvoke
$r9.<android.telephony.SmsManager: void sendTextMessage
(java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.Pendi
ngIntent)>
($r8,null,$r7,null,null) on line 27,
from the following sources:
  -$r2 :=@parameter1: android.content.Intent
(in<com.di.user.broadcasttest.MyBroadCastReceiver:  void onReceive
(android.content.Content,android.content.Intent)>)
Maximum memory consumption: 1220.334248 MB
Analysis has run for 8.206706944 seconds
```

Figure 5.2: BETA's Result

In Figure 5.2, the BETA applied *Flowdroid* to analysis the vulnerable receiver class in the hypothetical application, which spends 8.2 seconds around for the data analyzing. There are some data leakage behaviors in the customized sample, and the AsyncTask running in the background consumes large amount of system resources. From the BETA's result, there is one line built already from the source to the sink, which is from the source *void onReceive()* to the sink *void sendTextMessage()*. In detail, the source contains the content from the Intent input. In addition, the sink and source in Figure 5.2 also are displayed. Based on the *4P* policy, the Intent input to the SMS output is the P_1 path. As a result, the receiver is not only malicious to sent the privacy data outside, but also it is easy to be called by other components.

In summary, this example indicates that a receiver is vulnerable to be hijacked when it is exported to the public without limiting its attributes, at the same time, it can also be used for the malicious implementation. Compared to the results of expected and real test in the sample receiver, the BRVD is able to accurately reflect the characteristic of the behavior, including data pumping and information disclosing.

Moreover, the testing result of the customized sample demonstrates that applying data-flow paths to model the data leakage is straightforward and simple. Therefore, the detecting system, BRVD, aiming at analyzing common characteristics of malicious code owns four attacking paths imposed by *4P* policy.

5.2 Real Samples

We applied the BRVD on the real Android application from online markets. From the SUKE filtering, we statically analyze Java and XML source code of 55 market applications and there are 132 registered receivers in total. Then according to the BETA module, we find out 11 receivers in 8 applications that are related to vulnerabilities of Broadcastreceiver component. These 13% of applications are specific vulnerable by improperly utilizing *startActivity()*, *sendBroadcast()*, and *startService()* based on the *4P* policy. All of them, which are displayed in the table 5.1, have malicious behavior involving the common operation. In addition, there are 3 applications have "memory explosion" problem in the testing computer, due to the large APK size that is bigger than 4 MB.

Table 5.1: The Results of Suspicious Applications Using BRVD.

APKS	Paths	Vulnerable Receivers(N)	Time(s)
Bitdefender Antivirus	P_1& P_2	1	41.23
Lango Messaging	P_1	2	78.45
Contact Storage V4.4	P_1& P_2	2	35.85
SecDownload Provider	P_1	1	7.10
PPP Widget (Discontinued)	P_1 &P_2	1	9.28
Bluetooth Walkie Talkie_1.1.2	P_2	2	10.34
Weather Radar	P_2	1	18.79
BomBom-Free SMS	P_2	1	56.11

Table 5.1. shows a number of commercial APKs from both Chinese and North American third-markets. In detail, Column "Paths" displays the potential attacking paths based on the *4P* policy, along with the calculating time under the "Time" column. Typically, a detected online application has at least one registered receiver located in amount of source codes, so that the total number of receivers cannot be manually filtered in source files, as well as the number of vulnerable receivers. This complexity has been simplified in column "Vulnerable Receivers" as a final result of detection.

The number of vulnerable receivers is an indication of the variability of behavior for a callback function (e,g., *onReceive()* in the motivation example). Our analysis is considered the subset of Android, such as Activity, Service, and other Broadcastreceivers. To maintain soundness with chains of target Broadcastreceiver components in the Android platform from the same or different applications, we taint the data transmission by $X_{Source} \rightarrow Y_{Sink}$ (defined in the Section 4.2.2) in a Broadcastreceiver. In addition, these Android components should be considered to individually maintain the invariant information. In other words, the privacy data could send to an vulnerable receiver from a permission-protected component, or from a malicious receiver to a secure component: if any permission-protected data D flows into a component, then this component must enforce the related permission via its manifest using *Context.checkCallingPermission()*. However, none of these steps displays in the flow, but the input-output data between Broadcastreceiver and other components. This is the reason why we implemented an acceptable taint analysis engine such as *flowdroid*.

Our analysis aims to model the variability more precisely in an vulnerable receiver, by accounting for the percentage of the two main sources in the X_{Source} and the various exists in the Y_{Sink}. After collecting the Source and Sink data from 11 vulnerable receivers on the above table, we list X_{Source} and Y_{Sink} percentage in the Figure 5.3 and Figure 5.4 respectively.

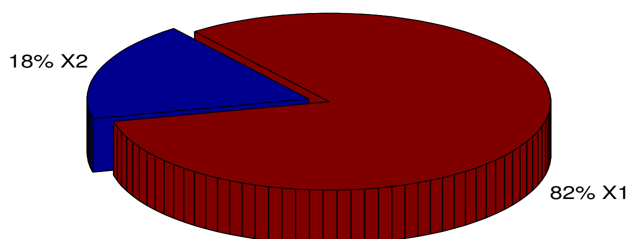


Figure 5.3: Source of Taint Analysis in vulnerable Receivers

Figure 5.3 shows the percentage of different two sources (X_1 and X_2) in the vulnerable receiver. According to the *4P* policy in the Chapter 3, the X_1 represents the data resource which is outside of the receiver from the Intent mechanism, while

the X_2 refers to the source data generated within the receiver. From the experiment information, the proportion of X_1, which occupies 82% , is far greater than the X_2. In other words, the utilization of the Intent mechanism is the main attack approach in the Broadcastreceiver component for the data transmission.

Most of testing samples in the Table 5.1 contains triggering events from the vulnerable Broadcastreceiver, which starts a new life-cycle to other components, such as lifecycle for a Service or an Activity. Meanwhile, all sinks in the end of the data-flow are correlated with SMS sending, Internet connecting, and data querying. Through the analysis of these mainstream malicious code samples, the current form of the Android malware is to implicitly call the framework layer API, and undisguisedly abuse a variety of permission beyond the user's comprehending.

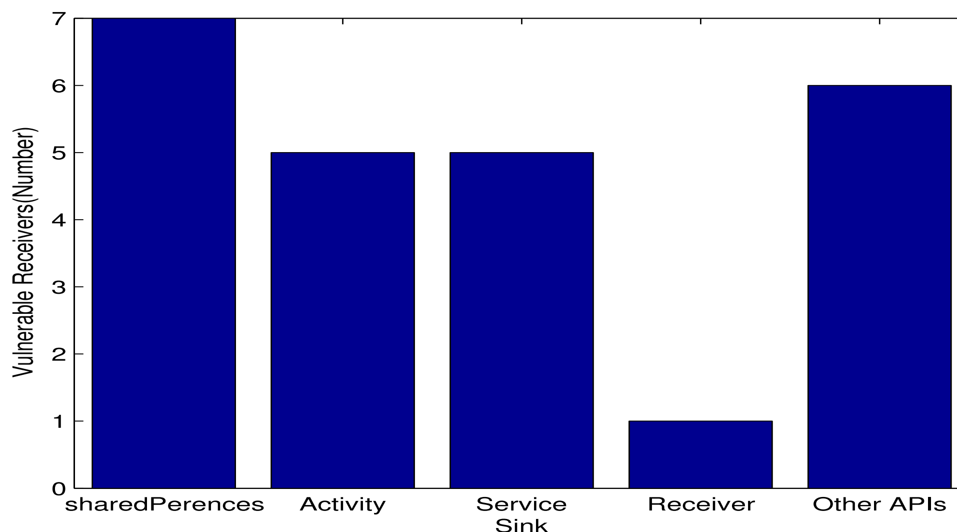


Figure 5.4: Sinks in Vulnerable Receivers

Therefore, to pursue the precise result of the taint analysis, Figure 5.4 presents the data output of the data-flow in the vulnerable receiver. The vulnerable receiver has several key Sinks for the data transmission. In specifically, the sharePerences, as a database, is the most frequently queried component by vulnerable receivers, accounted by 7 of 11 suspected samples in our test. Activity and Service components are the next easily to be triggered as a secondary risk module, such as *startActivityForResult()* and *startService()*. Due to the short lifecycle of the Broadcastreceiver, starting an another Intent broadcast to deliver information is the lowest risk, such as *sendBroadcast()*.

Furthermore, the column "Other APIs" is the number of various APIs that is directly triggered by the vulnerable receiver to send data outside. In the test, most of activated APIs in the vulnerable receiver are applied the Android logging mechanism, such as using *android.util.log* to write the log information through the vulnerable receiver for debug. In fact, the information in the log file is unsafe before the Android 4.1 version [37], and some of sensitive information, such as WLAN mac address, will be easily obtained and accessed by *Runtime.getRuntime().exec("logcat")* after passing the related permission.

Table 5.2: Testing Results from ApkScan and Eacus

	ApkScan		Eacus	
	Results	Risk Rating	Results	Risk Rating
Bitdefender Antivirus	✓	*Suspicious*	×	*NotFound*
Lango Messaging	✓	*Confirmed*	✓	*High*
Contact Storage	×	*Normal*	✓	*Medium*
SecDownload Provider	✓	*Suspicious*	✓	*Medium*
PPP Widget(Discontinued)	✓	*Suspicious*	✓	*Medium*
Bluetooth Walkie Talkie_1.1.2	×	*Normal*	✓	*Low*
Weather Radar	✓	*Confirmed*	✓	*Low*
BomBom-Free SMS	×	*Normal*	×	*NotFound*

Note: *Risk*(✓); *No Risk*(×)

Currently, there are few numbers of Android security tools that specifically detect Broadcastreceiver component. To verify the testing result of the BRVD for real sample behaviors, we compare our proposed BRVD with two similar third-party tools, NVISO ApkScan[28] and Eacus[19]. There are 8 suspicious Android applications in Table 5.2 that have been detected in BRVD. All of them contain vulnerabilities as detailedly showed in Table 5.1.

In contrast, ApkScan is an online Android tool that scans *.apk* files, especially Service components in Android applications. It detects five of eight suspicious applications. Because of the short lifecycle of the Broadcastreiceiver, Service components are always connected with Broadcastreiceiver and assisted in doing some time consuming works in the Android application development. Furthermore, the Service component is invisible running in the background which is similar to the Broadcastreceiver mechanism. Hence, the ApkScan's testing result is the one that has significance for the correlation of BRVD's results. In addition, Eacus is a lite framework analysis tool

for Android applications. It can directly analyze an application based on Android permissions through its related behaviors. As a completely permission analyzing tool, Eacus's examining data, which finds 6 suspicious applications, can be a comparison for BRVD's testing results.

According to the similar working principle, we choose ApkScan and Eacus to test all BRVD's results from Table 5.1. The compared results are showed in Table 5.2, which can be observed that we not only compare with the BRVD's testing results, but also list the risk rating from each of comparing tools. In Table 5.2, the ticks for each applications are the threaten results found through two examining tools, and crosses are the testing results that they assume as non-risk. Compared with BRVD, there are some Android applications that cannot be found by ApkScan and Eacus, such as BomBom-Free SMS.

Next, we will look at two suspicious Android applications from the BRVD's testing results in the table 5.1, which are Lango Messaging and BomBom-Free SMS. One is confirmed as a risk Android application by testing security tools, whereas the other one is regarded as a no risk application. Also, these two applications will be examined in detail as study cases for understanding the Broadcast Hijacking attack and Intent Spoofing attack.

5.2.1 Case Study: Lango Messaging

Lango Messaging is a messaging Android application similar to famous chatting applications Skype and Line, but it has more emotion icons for users.

By applying ApkScan and Eacus, the testing results of Lango Messaging confirm that it is a malicious Android application as showed in Figure 5.5. ApkScan gives a general result for the risk of phone number leakage, while Eacus has an advance analysis for the lango messaging, which displays the specific behaviors of these suspicious components based on the related permissions, including Service and Activity.

In Figure 5.6, BRVD gives a warning result for the Lango Messaging application. There are five suspicious receivers that have been detected in SUKE, while BETA lists a data flow from different receivers to a method *startActivityForResult()*. Specifically, the five suspicious receivers are in the BETA's result. In other words, those suspicious receivers have data transmission behaviors to trigger a specific Activity component during their activation period. Besides the detecting result, Li et at.[21] introduces *startActivityForResult()* as an vulnerable method for the data callback function. As a result, the privacy data through the suspicious receiver takes a high risk in leakage to get the data back from a specific Activity. For example, a new Activity can obtain

users' phone number and return the information to an vulnerable receiver by applying *startActivityForResult()*.

Risk rating
Suspicious acitivity detected

General information

File name	zlango.apk
Other known file names	None
Origin	Manually uploaded by anonymous user [2015-12-15 02:24:01]
MD5 hash	646a8fcb267a8465b26bba6b66f74930
SHA256 hash	53e2f6650bccc30b244b85b061a1244c3055ab39a7d578734fbb711eb7c9ea9c
File size	12550.6 KB
Worker	NVISO_API_KALI_01

Virus Total scan results

NANO-Antivirus Trojan.Android.Agent.cwzgot

Information leakage

Network information leakage

Destination	50.63.202.63:80
Tag	TAINT_PHONE_NUMBER
Data (ASCII)	POST /sy/update HTTP/1.1 deviceName: Full Android on Emulator_unknown msisdn: 15555215554 Content
Data (RAW)	504f5354202f73792f75706461746520485454502f312e310d0a6465766963654e616d653a2046756c6c20416e64726f696420 6f6e20456d756c61746f725f756e6b6e6f776e0d0a6d736973646e3a20313535353532313535340d0a436f6e74656e74
Operation	send

(a) NVISO ApkScan's Result

Eacus - MobiSec Lab

AppName: Zlango Messaging
Package: com.zlango.zms
MD5: 646a8fcb267a8465b26bba6b66f74930
SHA-1: d9f95f7ef9f9c9f5d797194720dd5351972bd643
Version: 3.15.15
Signature: CN=zlango
PublicKey: Sun RSA public key, 1024 bits modulus:
1015058393429932664169799263839695468407715240317228083153178748912494524000563680374835814181648676380593868428524789413410164443300290298045846103893229245341908916952339375285774104297453607221981023296523762258552764059143798796672009327125158413215121828303289715107524953271030390810643166010484507 16313
public exponent: 65537
sig_version: on processing
sig_serialnum: on processing

Summary
This task takes: 42.129s
Conclusion: Coming soon
Link to:

Basic Analysis
Re-pack Check: No matching item found in Certification DB(total 12266 certifications)
ADs Check: Flurry(total 76 advertisement signatures)
Obfuscator Check: Not Found(total 9 signatures)
Virus DB Check: Not Found(total 24 signatures
Embedded APK Check: Not Found

Plugin Analysis
FakeID Check: Not Found
Root Check: Offline
Smishing Check: Offline
Masterkey Check: Offline

Advance Analysis
Behavior Check: #1 java engine takes 427 ms.
Risk Level: HIGH (total 84 checkPoints and 18 rules)
[Receiver] On receiving sms message, the app intercept sms message
[Receiver] On receiving sms message, the app read sms message
[Receiver] On receiving sms message, the app manipulate contacts
[Activity] On startup, the app read device id
[Activity] On startup, the app read location message
[Activity] On startup, the app manipulate contacts
[Receiver] On receiving sms message, the app manipulate accounts
[Receiver] After device booted, the app read sms message
[Receiver] On receiving sms message, the app read info of running tasks
[Receiver] On receiving sms message, the app read info of top activity
[Receiver] After device booted, the app manipulate accounts

#2 c++ engine for server, build version: 2015090207 takes 39771 ms.
Risk Level: HIGH
[Service] On startup, the app read info of top activity
[Service] On startup, the app manipulate contacts
[Activity] On startup, the app manipulate contacts
[Receiver] After device booted, the app manipulate accounts
[Service] On startup, the app manipulate accounts
[Service] On startup, the app intercept broadcast
[Receiver] After device booted, the app read sms message
[Activity] On startup, the app read device id
[Activity] On startup, the app read location message
[Service] On startup, the app read sms message
[Service] On startup, the app read info of running tasks

Heuristic Check: Check pass
Privilege Check: Does not find code for below sensitive operation. Over-privilege or hidden malicious operation. CHECK it please.(takes 536ms)
android.permission.CALL_PHONE
android.permission.MANAGE_ACCOUNTS
android.permission.USE_CREDENTIALS
android.permission.WRITE_SECURE_SETTINGS

(b) Eacus's result

Figure 5.5: Lango Messaging Analysis Report

```
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/AndroidManifest.xml
Important Status: true
Action name: com.zlango.zms.transaction.MESSAGE_SENT
Android name: .transaction.ZmsSentReceiver
Export:
Priority:
Permission:
Matched results:
-------------------------------------
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/AndroidManifest.xml
Important Status: true
Action name: android.intent.action.TRANSACTION_COMPLETED_ACTION
Android name: com.zlango.mms.transaction.RetreivedDataReceiver
Export:
Priority:
Permission:
Matched results:
-------------------------------------
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/AndroidManifest.xml
Important Status: true
Action name: com.google.android.c2dm.intent.RETRY com.zlango.zms
Android name: com.google.android.c2dm.C2DMBroadcastReceiver
Export:
Priority:
Permission:
Matched results:
-------------------------------------
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/AndroidManifest.xml
Important Status: true
Action name: com.android.vending.billing.IN_APP_NOTIFY com.android.vending.billing.RESPONSE_CODE
com.android.vending.billing.PURCHASE_STATE_CHANGED
Android name: .gbilling.BillingReceiver
Export:
Priority:
Permission:
Matched results:
-------------------------------------
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/AndroidManifest.xml
Important Status: true
Action name: com.zlango.zms.RETRY_SEND_IP_ACTION com.zlango.zms.RETRY_FETCH_IP_ACTION
com.zlango.zms.DO_POLLING
Android name: com.zlango.mms.data.IPMessageManager
Export:
Priority:
Permission:
Matched results:
```

(a) SUKE's Result

```
Found a flow to sink virtualinvoke
$r0.<com.zlango.zms.app.ComposeMessageActivity: Void startActivityForResult
(android.content.Intent,int)>($r4, 17),
from the following sources:
 - $r2 := @parameter1: androId.content.Intent (in
<com.zlango.zms.transaction.ZmsReceiver: void onReceive
(android.content.Context,android.content.Intent)>)

- $r1 := @parameter0: android.content.Context (in
<com.zlango.mms.transaction.PushReceiver: void onReceive
(android.content.Context,android.content.Intent)>)

- $r2 := @parameter1: androId.content.Intent (in
<com.zlango.mms.transaction.SmsReceiver: void onReceive
(android.content.Context,android.content.Intent)>)

- $r2 := @parameter1: android.content.Intent (in
<com.zlango.mms.transaction.RetreivedDataReceiver: void onReceive
(android.content.Context,android.content.Intent)>)

- $r1 := @parameter0: android.content.Context (in
<com.zlango.mms.transaction.RetreivedDataReceiver: void onReceive
(android.content.Context,android.content.Intent)>)

- $r1 := @parameter0: android.conteNt.Context (in
<com.zlango.zms.transaction.DailyTasksReceiver: void onReceive
(android.content.Context,android.content.Intent)>)

- $r1 := @parameter0: android.content.Context (in
<com.zlango.zms.transaction.ZmsSentReceiver: void onReceive
(android.content.Context,android.content.Intent)>)
```

(b) BETA's result

Figure 5.6: BRVD's Report

Also, there is a code snippet of Lango Messaging in List 5.1. It is a registered receiver named "ZmsSentReceiver" that is vulnerable in the Lango Messaging. From line 14 and line 18, it is this receiver's static registration in the manifest.xml which displays without any protection, such as *android:exported* and *android:permission* we discussed in the Chapter 3. In particular, the receiver has messages obtaining function in line 3 and keeps them into a specific local file in line 8. Meanwhile, the Box ID is constrained with number 2, which is easily hijacked under an vulnerable receiver.

Listing 5.1: code snippets of Lango Messaging

```
1  public void onReceive(Context paramContext,Intent paramIntent){
2   if((getResultCode()==-1)&&("com.zlango.zms.transaction.
       MESSAGE_SENT".equals(paramIntent.getAction()))){
3   Uri localUri =paramIntent.getData();
4   try{
5  MessageItem localMessageItem=MessageItemManager.getInstance().
      get(localUri);
6   if(localMessageItem.getBoxId() !=2)
7  if("sms".equals(localMessageItem.getTransportType())){
8  if(Telephony.Sms.moveMessageToFolder(paramContext,localUri,2))
9    localMessageItem.setBoxId(2);
10     }
11    }
12   }
13 }
14  <receiver android:name=".transaction.ZmsSentReceiver">
15   <intent-filter>
16 <action android:name="com.zlango.zms.transaction.MESSAGE_SENT"/>
17   </intent-filter>
18 </receiver>
```

According to the *4P* policy, this is the path P_3 that is from an internal data generator *getData()* to an unprotected local file *BoxId(2)*. For those attackers, they can query the specific Uri through triggering the vulnerable "ZmsSentReceiver" receiver, which is a highly risk.

5.2.2 Case Study: BomBom-Free SMS

BomBom-Free SMS is a popular real-time chatting application in the Android online store. It is SMS free not only for the BomBom user, but also for none BomBom users by implementing Cellular Network.

There are two reports in Figure 5.7, (a) and (b), display testing results from Android security tools which are NVISO ApkScan and Eacus. Compared with Lango Messaging, both of them state a non-risk result for BomBom-Free SMS, but BRVD provides a warning result.

Risk rating
no malicous activity detected

General information

File name	**BomBom_-_Free_SMS_to_all_Phone_3.0_.8_apk-dl_.com_.apk**
Other known file names	None
Origin	Manually uploaded by anonymous user [2015-12-15 02:00:15]
MD5 hash	9b6ae166e9a504c3089b2e8be7db0090
SHA256 hash	7c85d901a2fa2fdd5026f94b28641977cc3ef7a9808cec883b77ec40229ce3f4
File size	2056.92 KB
Worker	NVISO_API_KALI_01

Information leakage

Network information leakage

No network information leakage detected.

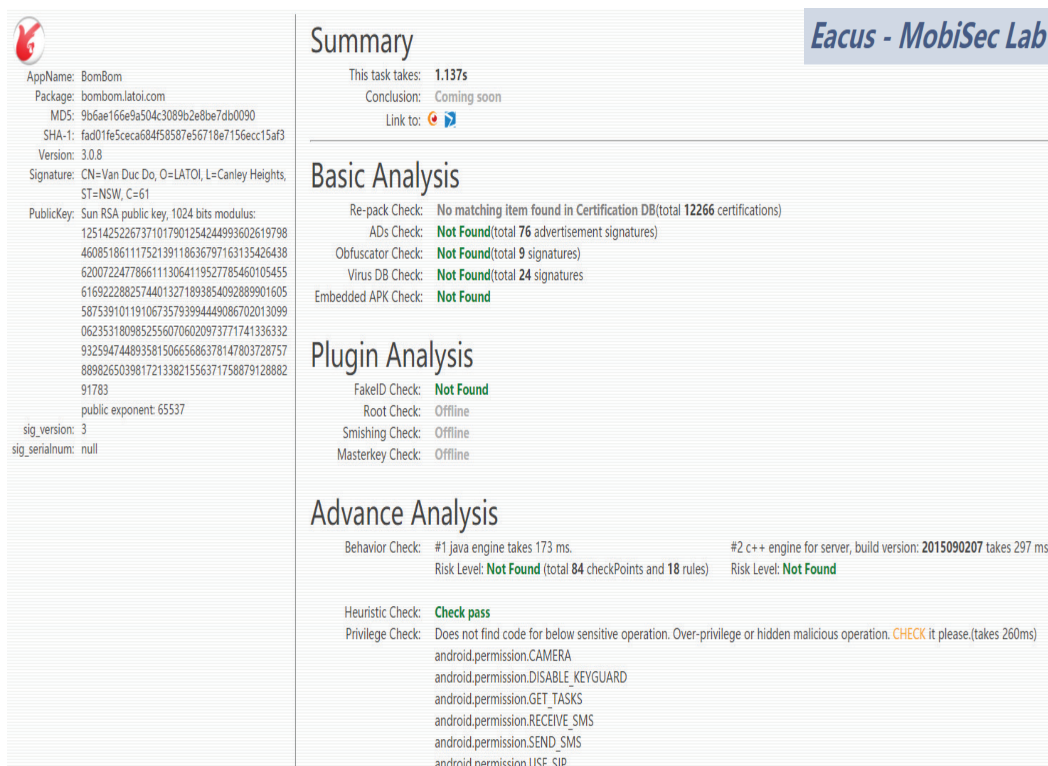
SMS information leakage

No SMS information leakage detected.

File information leakage

No file information leakage detected.

(a) NVISO ApkScan's Result



(b) Eacus's result

Figure 5.7: Analysis Reports From ApkScan and Eacus

```
Receiver Source File: file:///C:/Users/User/Desktop/filter/in/
AndroidManifest.xml
Important Status: true
Action name: bombom.latoi.com.VoiceReceiver
Android name: VoiceReceiver
Export:
Priority:
Permission:
Matched results:
-----------------------------------
Total dynamic registration: 6
Total static registration: 4
```

(a) SUKE's Result

```
Found a flow to sink virtualinvoke
$r1.<android.content.Context:
android.content.ComponentNamestartService

(android.content.Intent)>($r2) on line 15,
from the following sources:
      - $r1 := @parameter0: android.content.Context (
      in <bombom.latoi.com.VoiceReceiver: void onReceive
       (android.content.Context,android.content.Intent)>)
Maximum memory consumption: 1407.084912 MB
Analysis has run for 18.046281937 seconds
```

(b) BETA's Result

Figure 5.8: Analysis Reports From BRVD

In Figure 5.8, SUKE and BETA in the BRVD, respectively examine and give experimental results for BomBom-Free SMS. In detail, SUKE finds a receiver named "VoiceReceiver", which has no *android:exported* value. Meanwhile, BETA verifies this suspicious receiver based on the data-flow. The sink in the receiver is to start a Service within data, while the source is included internally. According to the *4P* policy, the path is $X_{1,2} \rightarrow Y_2$, along with P_2 and P_4. Next, we will analyze the specific attack method in this application based on the source generation.

Listing 5.2: code snippets of BomBom-Free SMS

```
1 public class VoiceReceiver extends BroadcastReceiver{
2 ...
3 public void onReceive(Context paramContext, Intent paramIntent){
4 ...
5   Object localObject2 = (example)paramContext;
6   localBuilder = new NotificationCompat.
        Builder(paramContext).setSmallIcon(2130837504).
        setContentTitle("Incoming Call:"+paramIntent.
        getPeerProfile().getDisplayName()).setContentText("...");
7   localObject2 = new Intent(paramContext, CallActivity.class);
8   ((Intent)localObject2).putExtra( "INCALL_USER_NAME",
        paramIntent.getPeerProfile().getDisplayName());
9 localObject2.startService(paramContext,WlanLockService.class)
10 localBuilder.setContentIntent(PendingIntent.getActivity(
      paramContext, 0, (Intent)localObject2, 134217728));
11  ((NotificationManager)paramContext.getSystemService(
       "notification")).notify(7819,localBuilder.build());
12  return;
13  }
14 ...
15 }
16 <receiver   android:label="Voice Receiver"
      android:name="bombom.latoi.com.VoiceReceiver"/>
```

In the listing 5.2, it shows a receiver named "VoiceReceiver" that records caller identification information and notifies the user with missed calls. This receiver has a common notification function which is similar to most of chatting applications, but an interesting vulnerable data-flow. The flow is an simple point-point structure which starts from *getDisplayName()* in line 6 to start a Service in line 9 and give a notification in line 11. Moreover, there is a conditional multiple arithmetic relationship between objects *localBuilder* and *localObject2*. Here, the *localObject2* is used for packaging and storing caller information by implementing *PendendingIntent()*, while the *localBuilder* is a carrier that contains *localObject2* for the data delivering. In Android, a *PendendingIntent()* is a token that allows foreign applications to use local application's permissions to execute a predefined piece of code[14].

Besides the data collection behavior in the BomBom-Free, as showed in line 16, the receiver "VoiceReceiver" is registered without *"android:export"* and *"android:permission"*. This means the receiver is callable from any other applications using the same value "*bombom.latoi.com.VoiceReceiver*", which can be under the Intent Spoofing attacks.

BomBom-Free SMS is simple and easy for any malevolent developers to disassemble and check the appropriate *receiver* attribute for Intent and the Broadcastreceiver class name. Therefore, the information stolen in the BomBom can be performed by any applications, which means the risk is high in this case.

5.3 Concluding Remarks

This chapter demonstrates the experimental environment, including test platform and host configuration information. According to the *4P* policy that introduced in Chapter 3, we provide experimental results, along with the whole experiment process. To implement the Broadcast Receiver Vulnerability Detection (BRVD), we apply two different samples, which are a hypothetical sample from our design and real samples from third markets. Both samples are effective, while the hypothetical sample is more targeted based on the vulnerable Broadcastreceiver. However, to test the authenticity of BRDV, the experiment of real samples is reliable.

First, the customized sample is used to describe the malicious behavior in Broadcastreceiver, which can clearly display attack paths. Then, we give real samples to test the performance of BRVD, compared with two similar security tools. Based on the specific cases from those experimental results, the veracity of BRVD for detecting the vulnerable Broadcastreceiver is proved.

Chapter 6

Conclusions and Future Work

The open source character in Android system combined with high-tech wireless network environment have lead to domination of Android phones in smartphone's market in last few years. Simultaneously, there are innumerable malwares that's against Android operating system. The previous introduction undoubtedly illustrated the various security issues related to the utilization of Android phones that both the industry and academia have realized and acknowledged.

6.1 Conclusions

This thesis analyzes the current Android development, threats and associated solutions. Aiming at security problems caused by the Android Broadcastreceiver component, we provide a static analysis to detect the malicious behavior in the Broadcastreceiver of Android applications. This vetting approach implements both semantic analysis and taint analysis to focus on an individual Broadcastreceiver component which mentioned rarely few before, rather than general detection. In detail, semantic analysis utilizes code characters to Broacastreceiver, while taint analysis is primarily through instruction analysis to identify and mark the outside data, as tainted data, which will be tracked using arithmetic and data movement instructions to construct explicit information dissemination. As a result, if the tainted data is applied for an unexpecting flow, such as jumping to other object addresses or format string, an unusual behavior alter will be announced. To reduce the false alarm rate and arrive a precise level in the detection result, control flow graph and auxiliary stack under the control-flow analysis are required for analyzing implicit information flow caused by the spread of tainted data through branch nodes.

According to the proposed method, a Broadcast Receiver Vulnerability Detection (BRVD) system for tracking the privacy data in the Broadcastreceiver component is

presented at end. It applies a new flow-based data path policy for malicious behaviors, consisted of two main modules; SUKE and BETA. To implement this system, vulnerable Broacastreceivers from Android applications should be marked and filtered in the SUKE module. Then the filtered receiver will be tagged with related data-flow in the BETA module. Based on different paths of data-flow, the malicious behavior can be confirmed. In addition, a log record function for the data tracking is applied in the method, which enhances the real-time updating.

The performance of the vetting method in the BRVD is thoroughly investigated compared with normal static analysis, our method is more detail and efficient for an independent Broadcastreceiver component in the Android framework. Furthermore, this system can be utilized before an Android application installation, which prevents potential crises in advanced. Meanwhile, in order to verify the effectiveness and performance of the detection system, this thesis also designed relevant tests for Android applications. There are two parts in the test; one is a custom sample based on the malicious behavioral characteristics, including GPS information collecting and SMS messaging, the other one is the real sample obtained from Android online markets. In contrast, due to the imperceptibility of malicious behaviors in real samples, the custom sample has more outstanding muilt-behavioral characteristics.

6.2 Future Work

Compared with dynamic analysis, the BRVD combined with both taint analysis and semantic analysis is high accurate and low risk, along with low energy. Due to kinds of outstanding characteristics, this technology will not only be applicable to the Android application, but also it has potential to be implemented to web applications and even IOS applications.

However, the disadvantage of the BRVD is the performance overheads. In order to satisfy users' increasingly rich life, there are many Android applications that have a large size with various functional services. In our real tests, some of the applications are even around gigabytes, which is time consuming. Although we explored some potential improvements building on prior work, the computing time is still out of our expectation. To get an accurate result in a short time is a challenge for us. Hence, how to make BRVD as a more efficient detection system is a main problem that needs to be solved.

Besides to the time consuming problem, the memory exploration is considered as another issue in the current research. Some of Android application may consume more

memory from the detection terminal. In addition to update the hardware standard, the computing algorithm needs to be strengthened by lowering amounts of threads construction in the background. Therefore, how to solve and simplify the generation of the number of data-flow edges but keep accuracy is another issue in the future.

And last, but hardly least, with the development of the mobile technology, those drawbacks will be further improved. All of the works are significant to have a body of research that addresses the security defense aspects and supports the state of security practice in the Android platform.

Bibliography

[1] Regular-Expression. `http://www.regular-expressions.info/`. [Online].

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 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*, pages 259–269. ACM, 2014.

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

[4] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8. ACM, 2012.

[5] Joany Boutet and Lori Homsher. Malicious android applications: Risks and exploitation. *SANS Institute*, 22, 2010.

[6] Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu. Test generation for embedded executables via concolic execution in a real environment. *Reliability, IEEE Transactions on*, 64(1):284–296, 2015.

[7] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[8] International Data Corporation. 2014Q2. `http://www.idc.com/getdoc.jsp?containerId=prUS25037214`, 2014. [Online; accessed 14-Aug-2014].

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[10] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[11] Eduardo B Fernandez and Xiaohong Yuan. Semantic analysis patterns. In *Proceedings of Conceptual Modeling-ER 2000*, pages 183–195. Springer, 2000.

[12] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/`.

[13] Dominik Franke, Corinna Elsemann, Stefan Kowalewski, and Carsten Weise. Reverse engineering of mobile application lifecycles. In *Proceedings of Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 283–292. IEEE, 2011.

[14] Google. Android Developer's Guide. `http://developer.android.com/guide/topics/manifest/receiver-element.html`. [Online].

[15] W3C Web Applications Working Group. Document Object Model (DOM). `http://www.w3.org/DOM/`. [Online].

[16] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[17] Yajin Zhou Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[18] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[19] MobiSec Lab. Eacus. `http://www.mobiseclab.org/eacus.jsp`.

[20] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Proceedings of Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[21] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *http://arxiv.org/abs/1404.7431/*, 2014.

[22] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[23] Amiya K Maji, Fahad Arshad, Saurabh Bagchi, Jan S Rellermeyer, et al. An empirical study of the robustness of inter-component communication in android. In *Proceedings of Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[24] Luo Xu Min and Qing Hua Cao. Runtime-based behavior dynamic analysis system for android malware detection. In *Advanced Materials Research*, volume 756, pages 2220–2225. Trans Tech Publ, 2013.

[25] Cheetah Mobile. 'Ghost Push': An Un-Installable Android Virus Infecting 600,000+ Users Per Day. `http://www.cmcm.com/blog/en/security/2015-09-18/799.html`. [Online; accessed 18-Sep-2015].

[26] Martin Mohr, Jürgen Graf, and Martin Hecker. Jodroid: Adding android support to a static information flow control tool. In *Proceedings of the 8th Working Conference on Programming Languages*, 2015.

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

[28] NVISO. ApkScan. `https://apkscan.nviso.be/`. [Online].

[29] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of USENIX Security 2013*, 2013.

[30] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[31] Opensource. android apktool google code repository. `https://code.google.com/hosting/moved?project=android-apktool`. [Online].

[32] Opensource. dex2jar github code repository. `https://github.com/pxb1988/dex2jar`. [Online].

[33] Opensource. Java Decompiler. `http://jd.benow.ca/`. [Online].

[34] Opensource. smali github code repository. `https://github.com/JesusFreke/smali`. [Online].

[35] Opensource. VLC media player. `http://www.videolan.org/vlc/index.html`. [Online].

[36] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[37] Siegfried Rasthofer. The Android Logging Service – A Dangerous Feature for User Privacy? `http://sseblog.ec-spride.de/2013/05/privacy-threatened-by-logging/`. [Online; accessed 2013/05/17].

[38] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of 2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[39] Google Report. Android Security 2014 Year in Review. `https://static.googleusercontent.com/media/source.android.com/en//devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf`, 2014. [Online].

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[41] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10–1, 2013.

[42] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Osman Kiraz, Kamer Yüksel, Seyit Camtepe, Sahin Albayrak, et al. Static analysis of executables for collaborative malware detection on android. In *Proceedings of Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE, 2009.

[43] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2015.

[44] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[45] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2012.