Security Analysis of Android Apps

Submitted in partial fulfillment of the requirements of the degree of

Master of Technology (M.Tech)

by

Prashant Maurya

Roll no. **163050074**

Supervisor:

Prof. R.K. Shyamasundar

Department of Computer Science & Engineering

Indian Institute of Technology Bombay

2018

Abstract

The number of malware apps is increasing with increase in number of Android phone users. These malware apps perform malicious activities such as exfiltrating private information. Though several works have been proposed to detect malware on Android but none of them successfully point out the "source" and "sink" API calls due to which security is compromised. Our approach rightly abstracts the possible leaks and identifies the points of leakage. Even though Android uses permission-based access control system, it cannot enforce fine-grained control over data causing leakage. We developed a tool to trace flow of data sensitive to a user. As there is no single entry point in an android application, it makes it difficult to analyze the flow of information through static analysis. We developed a hybrid approach for it that utilizes both dynamic and static analysis approaches. Our tool generates trace file for a running app containing sequence of sensitive API calls. The API methods to be traced are given as input to our tool. We will use DroidBot for automatic execution for apps which does not require login and collect log using logcat. This trace data is utilized in analyzing app's behavior. SOOT framework is used to convert bytecode into intermediate representation called JIMPLE. The analysis is done using RWFM model which is an information flow security model on JIMPLE. We label variables between two API calls and track their label till the end and see if any sensitive labeled variable is going out through any API method call.

Contents

List of Figures

Chapter 1

Introduction

Smart phone usage is increasing significantly. We use it in our professional as well as personal life. Thus it is a source for our private and confidential data. Users increasingly rely on mobile applications for various objectives. Google Android is the most popular mobile platform, hence the reliability of Android applications is becoming increasingly important. Android is the largest installed base of any operating system. Android is worthwhile target for attacks on users' privacy-sensitive data. Possible ways of data leakage are:

1. Certain apps are poorly programmed that leak data, and malicious apps exploit such data intentionally.

2. Apps that are not malicious but contain advertisement libraries might leak data.

When people install a new Android app, it prompts for user's permission before accessing personal information and certain hardware features. Ideally this information is for functionality of the app. For example map app wouldn't be useful if GPS data is not accessible to it. After an app has permission to collect data, it can share users' data with anyone the app's developer wants to, it might give users' data to a third-party companies. Android apps are reporting personal data to third-party tracking companies like Google Analytics, the Facebook Graph API or Crashlytics [13]. Many tools and techniques exist which detect malicious Android app up to certain extent. Most of the tools use either static or dynamic analysis. There are some tools that combine both static and dynamic analysis. Both static and dynamic analysis have their strengths and weaknesses. Dynamic

analysis gather more information than static analysis, therefore combining dynamic and static analysis methods would provide better results. Although many existing works have been proposed to detect malware but none of them successfully points out the API call due to which security is compromised. Most of the works use either dynamic analysis or static analysis or both to collect logs, extract features and further use machine learning approach to classify benign and malicious apps. Our approach is unique as it uses both dynamic and static analysis and further uses a RWFM security model to derive semantics. In the following section we discuss the approaches used to detect malicious app which are broadly divided into two:

- Static analysis, and
- Dynamic analysis

Static Analysis

Static analysis is a technique to detect malicious behavior by analyzing the code. This technique does not require running of an app onto emulator or device hence it requires less time and resource than dynamic analysis. This technique has small overhead, efficient and scalable. However this technique has a major drawback of code obfuscation. The code analyzed may not necessarily be the code that is actually run. Code obfuscation are used for hiding malicious behavior by some developers. Some malwares even perform transformation attacks that changes their code and behaviors during run-time.

Dynamic analysis

Dynamic analysis is aimed at detecting malicious behavior by executing the app on real environment. Dynamic analysis focuses on the run time behaviors and the system metrics of the application. However, dynamic analysis requires executions that will thoroughly cover program behavior. Complementary to static analysis, dynamic analysis is immune to code obfuscation and is able to see the malicious behavior on an actual execution path. The success of dynamic analysis depends on the behavior coverage. It is often very difficult very time consuming and less scalable.

Organization of Dissertation

This report is organized as follows: Chapter 2 gives an overview of Android OS, Android architecture, how different component interact and contains summary of research in this field. Chapter 3 describes SecFlowDroid, a tool for checking security threats. Chapter 4 describes SecFlowDroid+, modified version of SecFlowDroid. Chapter 5 describes example scenario for SecFlowDroid+. Chapter 6 includes experimental analysis and results using SecFlowDroid+. Chapter 7 gives conclusion and future work.

Chapter 2

Background and Related work

Backgroud

In this section we describe the overview of android OS. We first provide a high-level overview of the Android architecture, Android application component and how these components interact using `Intents`. Then we describe how Android uses permission-based access control system for user's private data and hardware access.

Android Architecture

Android operating system is a stack of software components based on Linux OS. It is an open source and Linux-based Operating System. Figure 2.1 [7] shows the Android software stack. Following are the major components of Android platform [7]:

Linux Kernel: This layer is core of android architecture offers service like memory management, power management, and security etc. When multiple Application runs on the Android OS, the main security purpose of the Linux Kernel is to separate applications resources from each other.

- Apps cannot have read permission to any other app's data,
- Apps cannot use any other app's memory space,
- Apps cannot use any other app's CPU resources, and
- Apps cannot use services like telephony, GPS etc. without permission.

Hardware Abstraction Layer (HAL): HAL consists of libraries to provide interface to hardware component such as bluetooth, camera etc. HAL exposes device hardware to higher level API framework. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.

Android Runtime: Android runtime(ART) runs multiple virtual machines on low-memory devices by executing DEX files. Each app run in its own instance of the ART.

Native C/C++ Libraries: Android posses its libraries which are written in C/C++ cannot be accessed directly. With the help of Java API framework, we can access these native libraries through apps. There are numerous libraries like libraries for audio and video formats etc.

Java API Framework: These API's written in Java makes the entire feature-set of the Android OS available to any developer.

Android App component

Android application can be seen as group of different component interacting with each other. To use a component, application has to explicitly request to particular component in intent filter under application's manifest file.In android four kinds of component are present.

Activity: It is one of the fundamental building blocks of an android app. An Activity represents a single screen in an app. Activity is a visible process which is provided on screen i.e. works in the foreground of the handset screen and allowing user to interact with the UI.

Service: A Service is a component that performs operations in the background without a user interface. They handle background processing associated with an application.

Broadcast Receiver: This component allows you to register for system or application events. They handle communication between Android OS and applications.

Figure 2.1: The Android software stack

Content Provider: Content provider is useful when data is to be shared across applications.

Intent

When apps need to communicate with each other it uses message passing method in Android. Intent is a message which contains the information regarding a recipient and data. Intent provide both inter-application and intra-application communication. An Intent is used to provide communication between components in several ways, there are three fundamental use cases:

- **Starting an activity:** You can start a new instance of an Activity by passing an Intent to `startActivity()`. The Intent describes the activity to start and carries any necessary data.
- **Starting a service:** You can start a service with `jobScheduler` which will execute this job on your application.
- **Delivering a broadcast:** A Broadcast Intent is an Intent sent to multiple applications at the same time.

Permission System

To access sensitive user data and certain system features, Android apps must request permission. The purpose of permission is to protect the privacy of an android user. Every android application has a manifest file which contains all the permission requirement which that application need.

Requesting Permissions

Android system uses sandboxing technique where each app runs in a sandbox and need to request for resources and data it requires outside the sandbox. Similar to Linux, android OS isolates the apps using IDs. To access the resources and data which are not available in sandbox needs to be declared in manifest file. Figure 2.2 [1] shows access to sensitive data available through protected APIs. Depending on the sensitivity of the resource and

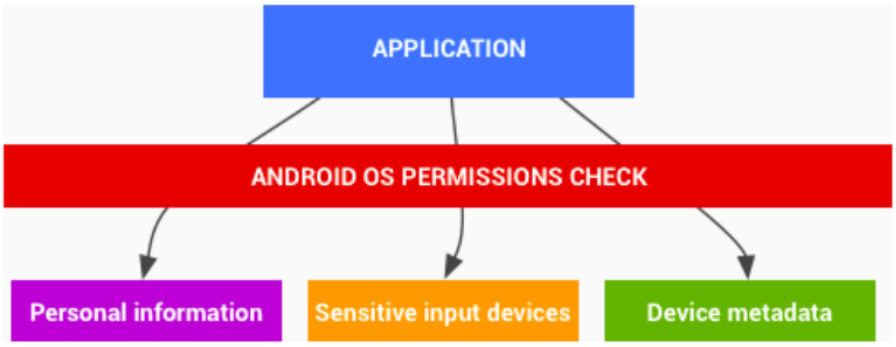

Figure 2.2: Access to sensitive data available through protected APIs

feature, the system might grant the permission automatically or might prompt the user to approve the request [6]. Permissions are divided into several protection levels. The protection level affects whether runtime permission requests are required. There are three protection levels that affect third-party apps:

Normal Permissions: For apps that require data or resources outside the app's sandbox but these data posses a very little risk to user's privacy. These permission are also declared in manifest, though the system automatically grants normal permission. The system doesn't prompt the user to grant normal permissions, and users cannot revoke these permissions.

Signature Permissions: For apps that are signed by same developer, there are some permission requested in manifest are granted automatically by the system at install time.

Dangerous Permissions: For apps that require data or resources outside the app's sandbox and these data could potentially affect the user's sensitive data then system prompt the user to grant permission either at runtime or in case of older version at install time.

Related work

DroidDolphin

DroidDolphin [18] is a dynamic Android malware detection framework using big data and Machine Learning. DroidDolphin makes use of the technologies of GUI-based testing, big data analysis, and machine learning to detect malicious Android applications. For better results it uses both dynamic and static analysis. The DroidDolphin framework contains four phases: Preprocessing, Emulation and Testing, Feature extraction, and Machine Learning. For Preprocessing it uses APIMonitor, to monitor specific API calls that may be called by malware. In emulation and testing phase it uses APE, for triggering GUI events by identifying the buttons and filling forms on the touch screen. Logs are collected for further analysis. Using the logged data from above phase, features are extracted in the next phase using n-gram model to represent the features. In the last phase, SVM machine learning algorithm to build a malware prediction model. The success of such a dynamic analysis approach depends on two issues:

1. Whether we have collected sufficient, useful runtime logs, and
2. How well the machine learning techniques are applied.

Anti-emulation technique hide themselves from dynamic analysis with emulator. The efficiency of dynamic analysis is not good. The code coverage is less. Time consumed is more.

Dynaldroid

Dynaldroid [16] is a System for automated dynamic analysis of Android application. This tool automatically does the dynamic analysis of an application by generating automated test cases for execution of the app. This tool operate in three phases:

1. **Preparation Phase:** It uses Apktool [3] for disassembling the APK. It uses some information extracted to generate a Robotium [9] based test case specific to the application.

2. **Execution Phase:** Robotium is used for generating the events for different types of UI elements(created in preparation phase). Systematic event triggering follows DFS as shown is the following figure 2.3 [16].

3. **Analysis Phase:** Traces collected in above phase is used for analysis. Training sets were observed to know the behavior corresponding to the malware samples. Both system call traces and API traces were observed when the malicious action occurred to deduce patterns which could help in identifying the occurrence of malicious activity.

SmartDroid

SmartDroid [19] is an automatic system for revealing UI-based trigger conditions in Android applications. SmartDroid automatically and efficiently detect UI based trigger conditions required to expose sensitive behavior of android apps. Compared to other dynamic analysis tools it takes less time in revealing the sensitive API calls. To reveal UI-based trigger SmartDroid uses both static and dynamic analysis. Static analysis produces Activity switch path which would lead dynamic analysis in triggering the UI elements. During static analysis Activity call graph (ACG) and Function call graph (FCG) shown in figure 2.4 [19] created to extract expected Activity switch paths. Then, use dynamic analysis to traverse each UI element and explore the UI interaction paths towards the sensitive APIs.

Dynalog

Dynalog [2] is an automated dynamic analysis framework for characterizing Android applications. Dynalog framework enables automated dynamic analysis of android applications. Dynalog framework is designed to accept large numbers of android apps, launch them serially in an emulator and log several dynamic behaviors. Dynalog has several components as shown in figure 2.5 [2]:

- **Emulator-based analysis sandbox:** The Dynalog framework utilizes DroidBox tool that can be used to extract some hig h level behaviours and characteristics by running the app on a device or emulator.

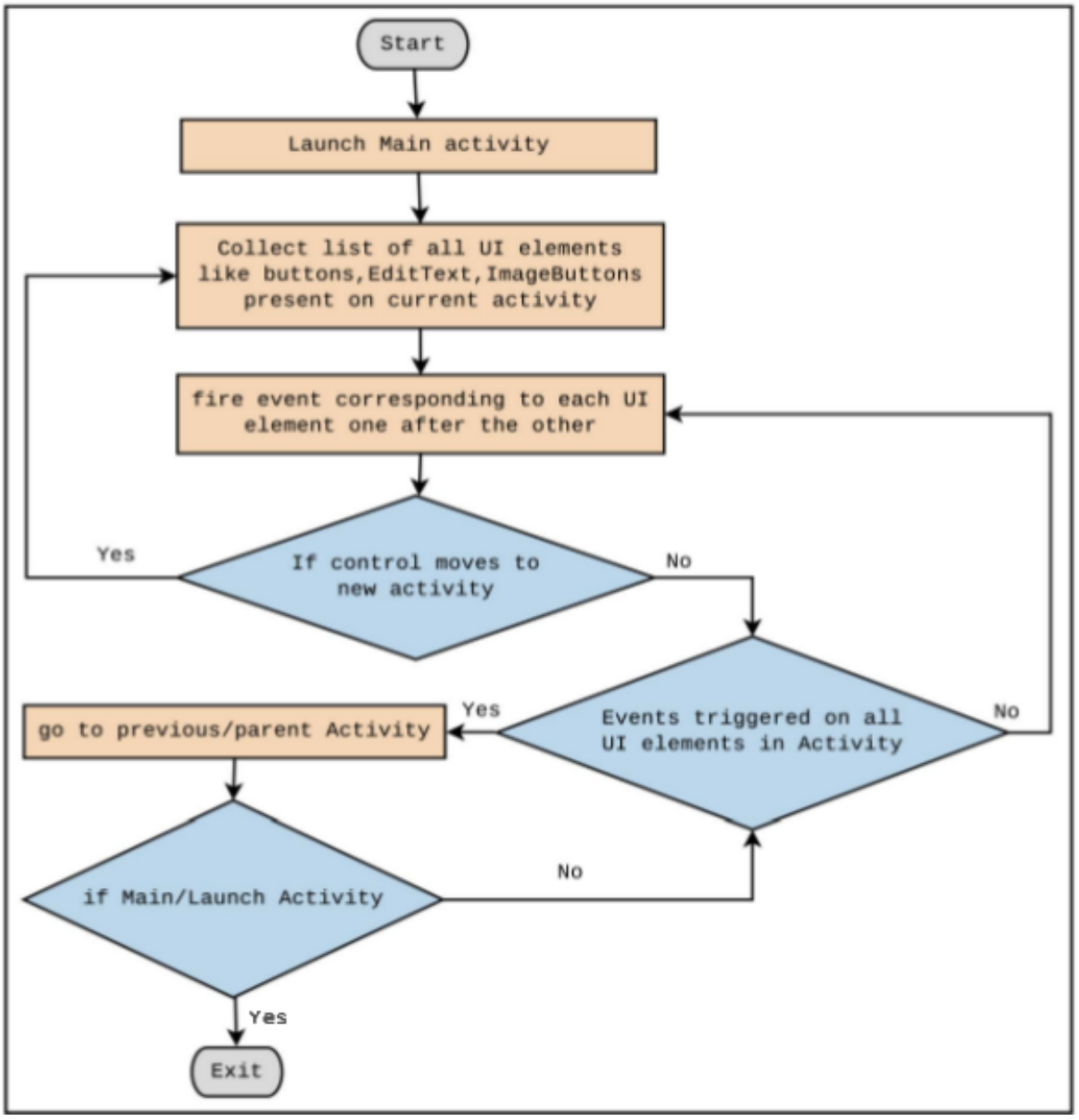

Figure 2.3: Flowchart for automatic application execution in Dynal

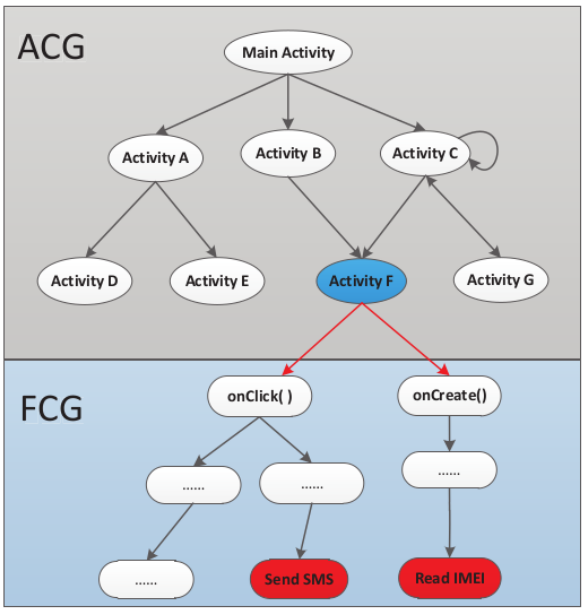

Figure 2.4: Activity call graph and Function call graph

- **APK instrumentation module:** APIMonitor is used to log API calls that are commonly used by malware.
- **Behavior/features logging and extraction:** Dynalog implements capability to extract specific log entries that correspond to monitored behaviors. The extracted log entry is further dissected to get lower level features. These features are used for classification of benign and malicious app.
- **Application trigger/exerciser:** DroidBox [14] includes Monkeyrunner [5] by default. The Monkey tool trigger a set of pseudo random events on the GUI.
- **Log parsing and processing scripts:** Some Machine learning classification method is used to classify benign and malicious app.

Comparison of related work done

Comparison of above tools are done based on following attributes:

- **Techniques:** The techniques used to detect malicious app which is broadly divided into two: Static analysis and Dynamic analysis.

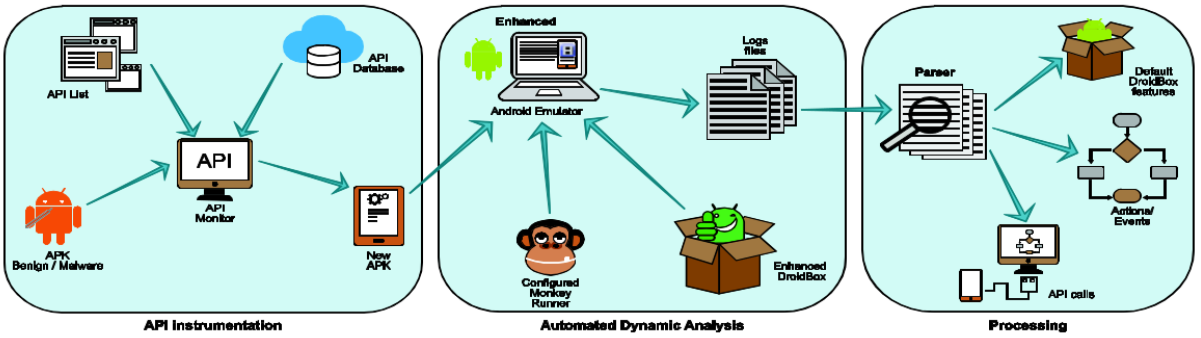

Figure 2.5: Dyanlog architecture

- **Test case generation:** To run the application in structured manner for better behavior coverage automated test cases are generated. Some tools generate test cases before emulation and others may follow some strategy at run time only.
- **Exploration strategy:** Exploration strategy determines which UI element to traverse next.
- **Security Analysis:** What method is used to classify benign app from malicious one.

	DroidDolphin	**DynalDroid**	**SmartDroid**	**Dynalog**
Techniques	Dynamic and Static	Dynamic	Static and Dynamic	Dynamic
Test case generation	AT runtime	Uses Robotium	ACG and FCG	No
Exploration strategy	Modified Monkey	Strategic	DFS	Random(Monkey)
Security Analysis	ML	ML	No	ML
Output	Secure?	Secure?	Logs	Secure?
Scalability	Moderate	Moderate	High	Low

Table 2.1: Comparison of related work done

Tools Used for Automated exploration of App

AndroidViewClient

AndroiViewClient [4] is an extension to monkeyuner. It used for automatic exploration of any android application. It generates re-usable scripts which could be used to run the app again following the exact same path. Following are some key features of AndroiViewClient:

- Fills forms and provides different kind of inputs based on UI element selected
- Create python file to simulate same UI interactions
- Slower UI interactions
- Control goes out of the app
- No app instrumentation
- Gets stuck on some apps
- Works with latest API versions

DroidBot

DroidBot [11] is a lightweight UI-guided test input generator, which explores an Android app. DroidBot works on almost any device and does not require instrumentation. Following are some key features of DroidBot:

1. Good code coverage quickly
2. Does not fill text fields
3. No app instrumentation
4. Works with latest API versions and devices
5. Keeps control within app / Returns control back to app in case of irrelevant exploration
6. produce UI structures and method traces for analysis
7. Runs app in continuous mode

DroidMate

DroidMate [10] is automated exploration tool for android app. It has powerful test generator which generates test by exploring the app at runtime. DroidMate first installs the

app and launches its main activiMy. Further it operates in a loop, exploration strategy takes last exploration action as input which contains information on the displayed GUI views and called API methods. Using this information it decides which next exploration action to conduct. Exploration action can be any of click, long-click, press home, press back, reset, terminate. Following are some key features of DroidMate:

- Works on emulators with API 19 and API 23
- App instrumentation required
- Good code coverage
- Runs app in multiple sessions
- Does not work on some apps

Monkey

Monkey [8] is a program that runs on a device or emulator for execution of app by generating pseudo-random streams of user events such as touches, clicks etc. as well as a number of system-level events. Monkey is generally used for stress-testing of an app.

Comparisons

Tool	**Fills form**	**App Instrumentation**	**Strategy**	**Supported Android versions**	**Code Coverage**
AndroidViewClient	Yes	No	Random View and Random Event	All versions	Medium
DroidBot	No	No	Model	All versions	Good
DroidMate	No	Yes	Model	Api 19 and 23	Good
Monkey	No	No	Random	All versions	Less

Table 2.2: Comparison of tools used for automated exploration of app

Chapter 3

SecFlowDroid: A tool for checking security threats

SecFlowDroid [1] follows a hybrid approach and utilizes information from both, the traces generated using APIMonitor and source code(APK). It uses this information to detect possible leaks and also have ability to detect where exact leakage is happening. SecFlowDroid not only finds out possible leak but also tracks the point where it is leaking by using taint analysis and information flow First it dynamically executes the given application in APIMonitor which generates a sequence of protected APIs that are invoked during the manual exploration of the app. Logs are collected which are to be used for static analysis. Then, it extracts JIMPLE code of those methods which has invoked protected APIs using SOOT-Android tool. IT analyzes these JIMPLE files and tracks information flow, including implicit flows, between API invocation using RWFM model and report misuse if found. SecFlowDroid consists of following steps:

1. Dynamic analysis using APIMonitor.
2. Extracting JIMPLE using SOOT-infoflow-Android.
3. Labeling of Source code using RWFM model.

3.1 Dynamic analysis using APIMonitor

APIMonitor has configuration file which has all the protected API calls that have to logged to monitor specific API calls that may be called by malware. APIMonitor repackage the apk with monitored codes by extracting the APK file into `smali` format, insert the monitor code, sign the APK with its own signature. With the repackaged APK, we would be able to trace the calls to these API calls in this application during the runtime, through `logcat`. Let API calls be $X_1, X_2, ..., X_n$, where each X_i is in the form of {ApplicationClassName, ApplicationFunctionName, APIClassName, APIFunctionName}. Logs are collected based on manual interaction with the application on the emulator/device.

3.2 Extracting JIMPLE using SOOT-Android

From logs collected in previous phase we have class name and function name of invoked API calls in app. Now we only need to analyze this part of the source code. We need to convert the APK into JIMPLE format using soot-infoflow-android framework only for these classes. This JIMPLE code is analyzed to check the data flow between every consecutive API (X i , X j) calls using RWFM model and report if any sensitive data is leaked.as shown in the figure 5.1. For example, suppose X_1 is getLocation() API call(source) and X_2 is global output API call such sentMessage()(sink). In JIMPLE code, it is written as:
a = getLocation();
b = a;
c = sentMessage(b);
In order to find out whether b contains sensitive information or not, Information flow model comes into picture, it tells the sensitivity of the variable b irrespective the complexity of the JIMPLE code.

3.3 Labeling of source code using RWFM model

JIMPLE statements are labeled according to RWFM rule. Read the JIMPLE class files according to the API calls given in trace file. For each API calls in trace file, do following:

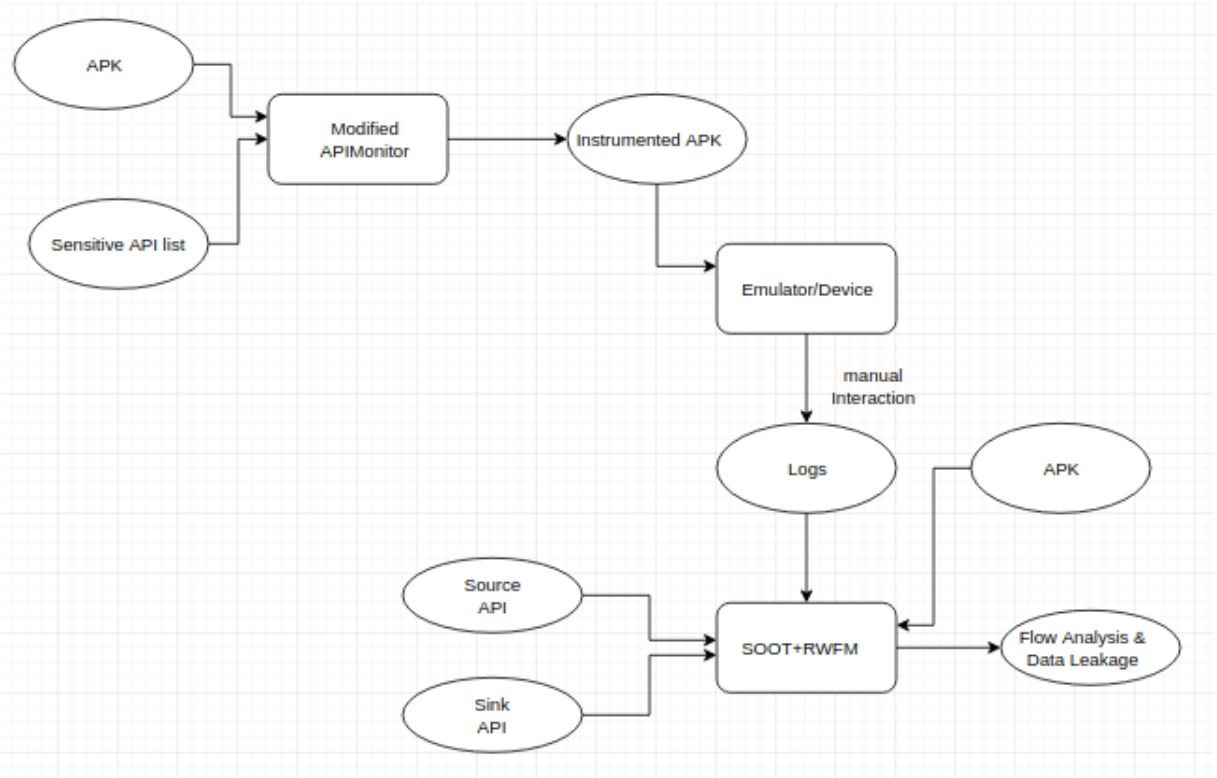

Figure 3.1: Analysis of an app using SecFlowDroid

- Label the variables from starting of the file to API call according to RWFM implementation.
- Label the variable calling or participating with API call as private.
- Trace the data flow from source to sink and report if any misuse happen according RWFM rules.

APIMonitor tool has been deprecated and does not work with `play store` apps now. So i needed some other alternative to trace sensitive API calls with the `class and function` name it was called in. There are tools available which traces API calls but does not log their `class and function` name. This log file will be input to the SOOT framework. In the next chapter we have described how the API calls are being traced.

Chapter 4

SecFlowDroid+

In this chapter we will see how the API calls are traced. For tracing API call we have developed an API tracing tool for Android based mobile device. The Android middleware provides abstractions for conveniently using device functions like sending SMS messages directly from applications written in Java using API calls without having to directly interact with native code libraries on the system level. Logging the sequence of these API calls helps us in tracking sensitive data. Our tool gives sequence of API calls with their class name and function name it was called in. For apps which do not require login can be explored using DroidBot. SecFlowDroid+ consists of following steps:

1. Tracing API calls
2. Extracting JIMPLE using SOOT-Android
3. Labeling of Source code using RWFM model

The last two steps are same as SecFlowDroid described in previous chapter.

4.1 Tracing API calls

Our tool uses program instrumentation mechanism for monitoring certain API calls at runtime. Our tool repackage the APK with monitored codes by extracting the APK file into `smali` format, insert the monitor code, sign the APK with its own signature. With the repackaged APK, we would be able to trace the calls to these API calls in this application

during the runtime, through `logcat` command. Logs are collected based on automated interaction with the application on the emulator using DroidBot. Instrumentation involves following steps:

4.1.1 Getting APK and Sensitive API

APK

`Google Play` supports `multiple APK` feature which allows developer to upload multiple APKs for his app to target devices with different device configuration. `Google Play` filters from these APKs to decide which device receive which APK. Thus we need same device to download the APK and test it for better results. For my project I have used `Samsung S6` to download APK and to test it. To download I follow these steps:

```
$ adb shell pm list packages  //list all packages on device
$ adb shell pm path <package name>  //returns path of given package name
$ adb pull <path>  //pull APK from device
```

Sensitive API

Permissions are organized into groups and each permission allows several permissions shown in figure 4.1 to be defined in app manifest file. For example the `Phone` group includes following:

1. READ_PHONE_STATE
2. READ_PHONE_NUMBERS
3. CALL_PHONE
4. ANSWER_PHONE_CALLS
5. READ_CALL_LOG
6. WRITE_CALL_LOG
7. ADD_VOICEMAIL

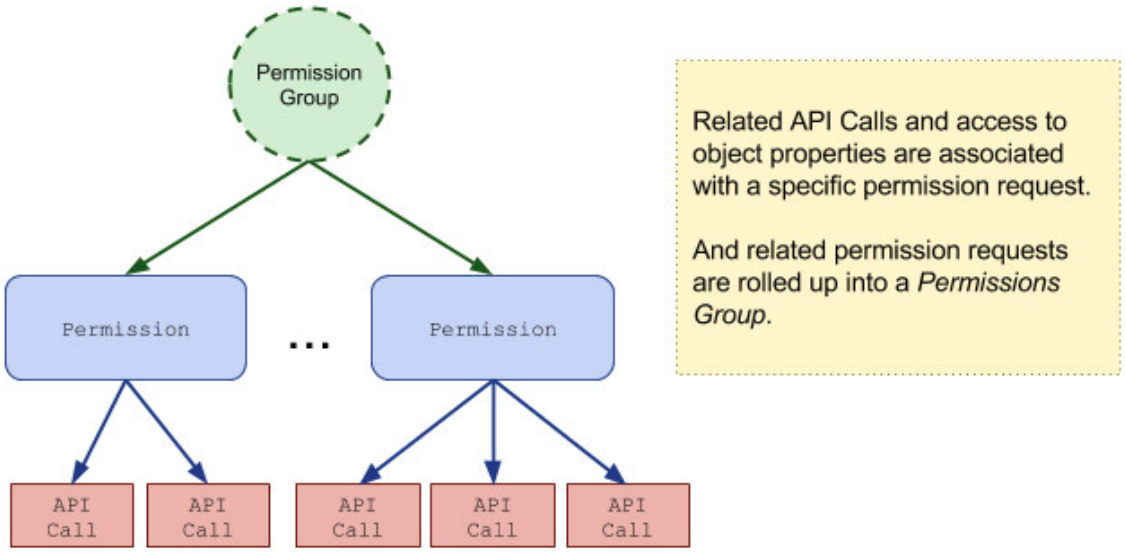

Figure 4.1: Permission group mapping with API

8. USE_SIP
9. PROCESS_OUTGOING_CALLS

The permission groups are defined in this way so that users can understand which permission access what data. Following permission groups are defined in Android:

1. Calender
2. Camera
3. Phone
4. SMS
5. Location
6. Contacts
7. Microphone
8. Storage
9. Sensors

Google Android does not give the mapping of permissions to API calls i.e. what are the API calls allowed on getting certain permission. There are three major studies which has attempted to recover this information SuSi [17], PScout [15] and axplorer [12]. I have used Pscout result for getting sensitive APIs.

4.1.2 Unpacking and Disassembling

APK is an archive that contains all the necessary files for successful installation and functioning of an app. The source code present in APK are in `dex` file format which is completely unreadable. We have used Apktool [12] for unpacking the application and disassembling its Dalvik byte code. Apktool uses `baksmali` to disassemble the `class.dex` files into `smali`. We have described how to insert the monitoring code in the next section.

4.1.3 Inserting Monitoring Code

Modifying and inserting the `smali` files is a challenging task. For a given list of sensitive API(default-list) to log, our tool creates a class file (Pro.smali) in `smali` which contains as many functions as there are API in the default-list. Every API in the default-list has a corresponding function in the Pro.smali file. Every functions in Pro.smali has same role of logging `class name, function name` and the API called with a `TAG` to facilitate logging only these things. While instrumenting the app for each `smali` file in the app we check whether any API in default-list is called. If we find one, we call the corresponding function in Pro.smali. This function call will log the class name and the function name the API was called in. I have written code in python to automate the editing of `smali` files and a bash script to do complete instrumentation of app. Following code is inserted into `smali` file:

```
invoke-static {{}}, {package_name}/Pro;->{method_name}()V
```

Here `package_name` and `method_name` is replaced by corresponding `package name` of the APK and `method name`. This statement calls the corresponding function in Pro.smali class.

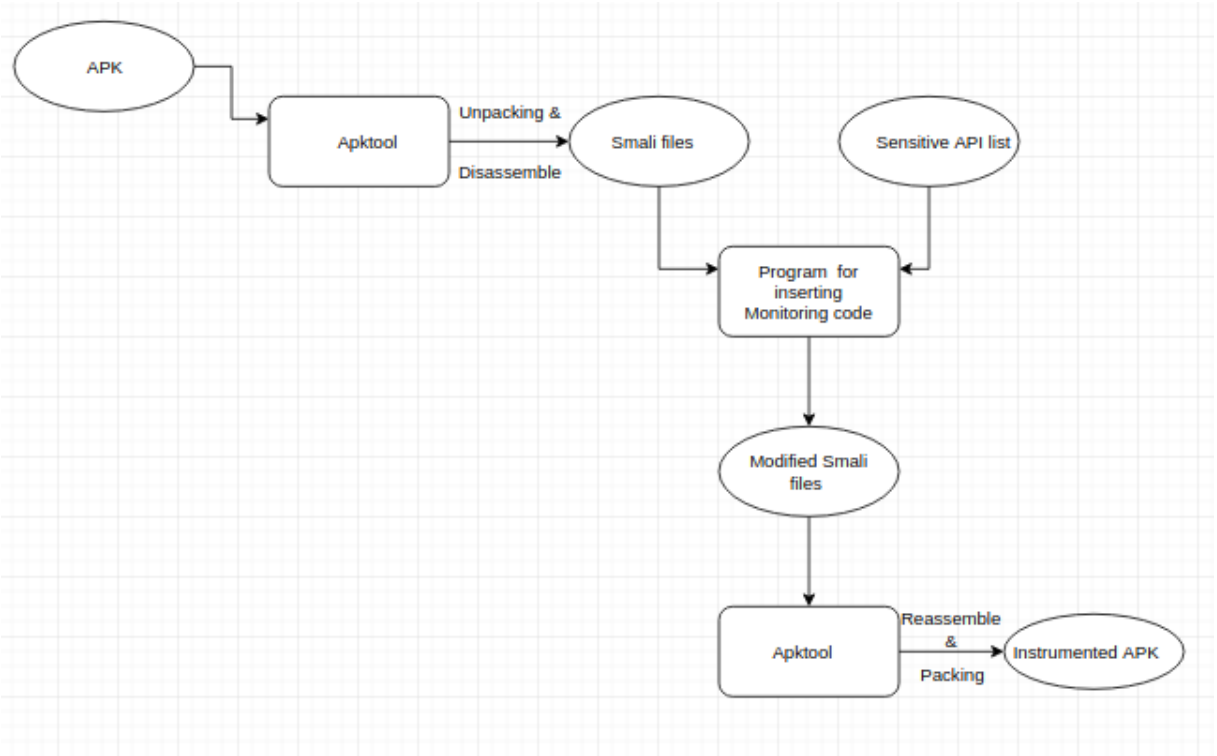

Figure 4.2: Block diagram for instrumenting APK

4.1.4 Assembling and Packing

Assembling Dalvik byte code and packing the hooked application is done using Apktool. After this step we will have the instrumented APK. We still need to sign this APK before we can run it on an Android device or emulator.

4.1.5 Signing the Hooked Application

An APK must be signed before installation with a certificate. When any user signs the app, the certificate facilitates in further updates of application. The main purpose of certifying an app is to distinguish the developers. Also, these certifications allow the system to grant or deny access to signature permissions. We have used standard `keytool` for creating signature and `jarsigner` for signing any APK.

4.1.6 Exploration of Apps

Given an APK and list of API methods to log, this mechanism creates an instrumented APK. Running this APK on an actual device/emulator will log sequence of API calls triggered by this app at runtime. For apps which do not require login can be explored automatically using DroidBot. For apps which require login can be explored manually. Manual interaction also lets you cover app's behavior which has more probability of data leakage.

Chapter 5

Example Scenario for SecFlowDroid+

In this section we will test the tool for a toy app(HelloWorld.apk). We built this app for testing of the SecFlowDroid+. HelloWorld app has two classes to implement two functionality. First one is MyActivity.class, it takes the phone's location and sends a sms to a mobile number. Second one is MyReceiver.class, it sends any number you dial on your smart phone to a mobile number. This app requires following permissions to perform above functionality.

1. Location
2. Phone
3. Message

Unpacking and Disassembling We will use Apktool to unpack APK and disassemble it.

```
$ apktool d -r HelloWolrd.apk
```

Now we have `smali` files for the corresponding classes in HelloWorld.apk

Inserting Monitoring Code We want to log sequence of API calls only. So we will search of sensitive API calls only and insert the monitoring code to log `class name`, `function name` and the API being called.

```
$ python createPro.py
$ python insertMonitoringCode.py
```

Assembling and Packing Following command will assemble Dalvik byte code and package the hooked app to create its instrumented APK.

```
$ apktool b HelloWolrd
```

Signing the Hooked Application: To run this app on an emulator or device I need to sign this app. I use my own signature to sign this app which I created using `keytool`.

```
$ jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore
my-releasekey.keystore HelloWorld/dist/HelloWorld.apk sec_analysis
```

Exploration of App We install this app on a device and test it. We covered both the functionality of the app.

```
$ adb install -g HelloWorld/dist/HellowWorld.apk
$ adb logcat -c
$ adb logcat -s Prashant
```

Following logs shown in figure 5.2 were collected.

Static Analysis: We give the apk, "source" and "sink" file as input and obtain following result after doing the static analysis.

SecFlowDroid+ starts from the first function in the log and searches for the sensitive api in that function. It labels all the variable whether it is sensitive. With reference to the following log, variable $r6 is sensitive as it contains sensitive data which is accessed by a api call which was listed in "source" api. See the label given to it is owner=[com.example.android. helloworld], readers=[public, com.example.android. helloworld], writers=[com.example.android. helloworld] where as label given to non sensitive api is owner=[com.example.android. helloworld], readers=[public,

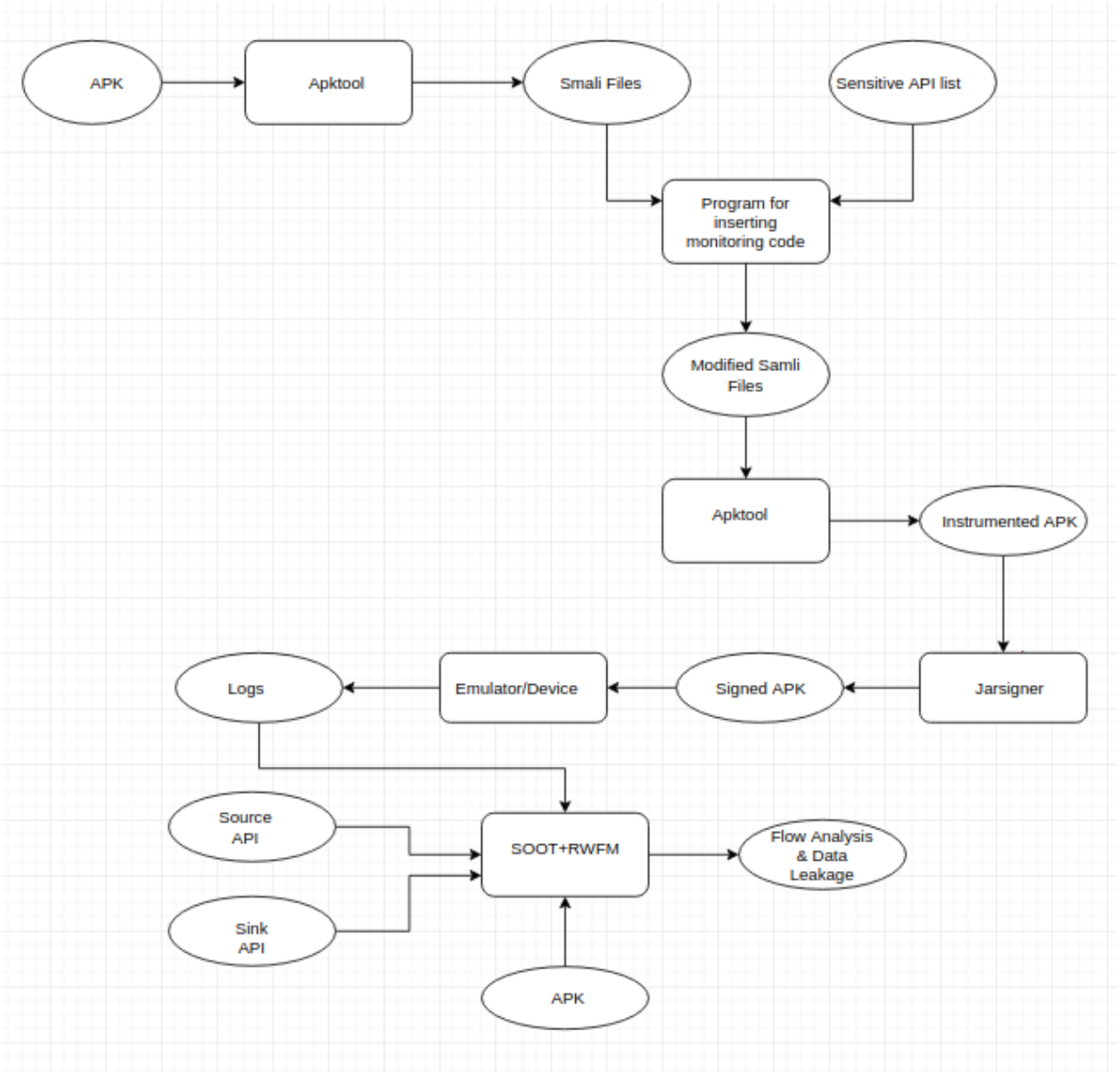

Figure 5.1: Block diagram of SecFlowDroid+

```
: Lcom/example/android/helloworld/MainActivity;->onCreate();->Landroid/location/LocationManager;->getLastKnownLocation
: Lcom/example/android/helloworld/MainActivity  ;->sendSMS();->Landroid/telephony/SmsManager;->sendTextMessage
: Lcom/example/android/helloworld/MyReceiver;->onReceive();->Landroid/content/Intent;->getStringExtra
: Lcom/example/android/helloworld/MyReceiver;->sendSMS();->Landroid/telephony/SmsManager;->sendTextMessage
```

Figure 5.2: HelloWorld API traces

com.example.android. helloworld], writers=[com.example.android. helloworld]. Inside the MainActivity method, sendSMS method invoked whose second parameter is sensitive. So, variable $r2 in sendSMS is given private label. Now this $r2 variable is going out via a sensitive api which is listed in "sink", thus this transaction has been declared as suspicious.

```
**************************************************
 Lcom/example/android/helloworld/MainActivity;.onCreate.Landroid/location/
 LocationManager;
**************************************************
stmt $r0 := @this: com.example.android.helloworld.MainActivity
{owner=[com.example.android.helloworld], readers=[public, com.example.android.
helloworld], writers=[com.example.android.
helloworld]}
.................
stmt $r6 = virtualinvoke $r5.<android.location.LocationManager: android.
location.Location.getLastKnownLocation
(java.lang.String)>("network")
-----------------------------------------------
Sensitive api :android.location.LocationManager getLastKnownLocation
-----------------------------------------------
{owner=[com.example.android.helloworld], readers=[com.example.android.
helloworld], writers=[public, com.example.android.helloworld]}
stmt virtualinvoke $r0.<com.example.android.helloworld.MainActivity: void
onLocationChanged(android.location.Location)>($r6)
.............
```

```
stmt virtualinvoke $r0.<com.example.android.helloworld.MainActivity: void
sendSMS(java.lang.String,java.lang.String)>("7408080673", $r4)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
com.example.android.helloworld.MainActivity.sendSMS
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
stmt $r0 := @this: com.example.android.helloworld.MainActivity
{owner=[com.example.android.helloworld], readers=[public, com.example.
android.helloworld], writers=[com.example.android.helloworld]}
stmt $r1 := @parameter0: java.lang.String
{owner=[com.example.android.helloworld], readers=[com.example.
android.helloworld], writers=[public, com.example.android.helloworld]}
....................
stmt virtualinvoke $r3.<android.telephony.SmsManager: void sendTextMessage
(java.lang.String,java.lang.String,java.lang.String,android.app.
PendingIntent,android.app.PendingIntent)>($r1, null, $r2, null, null)
!!!!!!!!!!!!!!!!! Suspicious !!!!!!!!!!!!!
subLabel {owner=[com.example.android.helloworld], readers=[com.example.
android.helloworld], writers=[public, com.example.android.helloworld]} tries
to write {owner=[com.example.android
.helloworld], readers=[public, com.example.android.helloworld],
writers=[com.example.android.helloworld]}
!!!!!!!!!!!!!!!!! Suspicious !!!!!!!!!!!!!
```

Following figure 6.6 shows the two suspicious transactions which sends a sensitive data outside through a "sink" API.

Figure 5.3: Static analysis result for HelloWorld app

Chapter 6

Experimental Analysis and Results

In this chapter we have illustrated three experiments to show some cases of data leakage. We have successfully found source and sink APIs for these apps.

Experiment 1: Spy

Spy app is an illustration of how a malicious app really works. This app asks user to fill a form which has three entries as shown in figure 6.1. It has a send button. On clicking the send button user expects that the details filled in the form will go to a server. But what happens in this app is it sends your location data to a server on clicking the send button.

First I instrument the Spy.apk using sensitive API list which has all sensitive APIs including the ones which fetch location data and sends data to a server. After instrumenting the app I need to sign it using my own signature. Now I can run this app on an emulator or a device. On running the app I get the log in figure 6.2.

On clicking the send button it sends the location to a server shown in figure 6.3.

On doing static analysis on the logs collected for Spy app we found the data leakage. Static analysis shows following result which proves that location data is going out to some server.

---

sensitive api stmt $r9 = virtualinvoke $r8.¡android.location.LocationManager: android.location.Locat
getLastKnownLocation(java.lang.String)-(”network”)

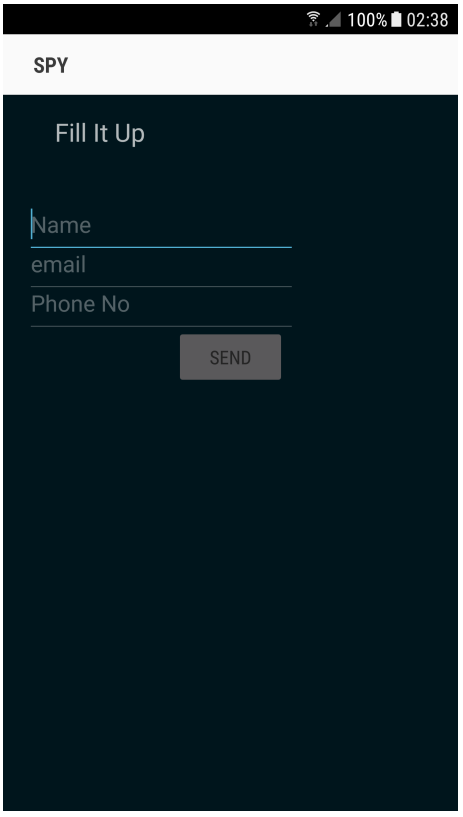

Figure 6.1: Screenshot of an activity in Spy app

```
: Ltashfik/spy/MainActivity$Send;->doInBackground();->Landroid/location/LocationManager;->getLastKnownLocation
: Ltashfik/spy/MainActivity$Send;->doInBackground();->Lorg/apache/http/message/BasicNameValuePair;-><init>
: Ltashfik/spy/MainActivity$Send;->doInBackground();->Lorg/apache/http/message/BasicNameValuePair;-><init>
: Ltashfik/spy/MainActivity$Send;->doInBackground();->Lorg/apache/http/message/BasicNameValuePair;-><init>
: Ltashfik/spy/MainActivity$Send;->doInBackground();->Lorg/apache/http/client/HttpClient;->execute
```

Figure 6.2: API traces collected for Spy app

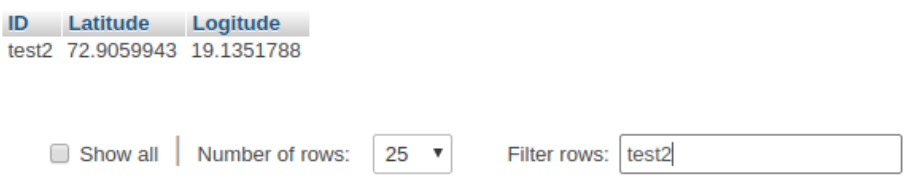

Figure 6.3: Location sent to a Server in Spy app

global output api stmt interfaceinvoke $r2.¡org.apache.http.client.HttpClient: org.apache .http.HttpResponse execute(org.apache.http.client.methods. HttpUriRequest)-($r3)

Experiment 2: ListContact

Created an app ListContact which has an activity with a button pressing which calls a callback function which read the user's Contact list by querying and send it via SMS to a phone number. Snapshot of the logs collected for this app is shown in the fig6.4. On

```
Lcom/netqin/ps/db/h;->b();->Landroid/database/Cursor;->getString
Lcom/netqin/ps/db/h;->a();->Landroid/database/sqlite/SQLiteDatabase;->query
Lcom/ducaller/fsdk/provider/c;->b();->Landroid/content/ContentResolver;->query
Lcom/ducaller/fsdk/provider/c;->b();->Landroid/content/ContentResolver;->query
Lcom/ducaller/fsdk/provider/c;->b();->Landroid/database/Cursor;->getString
Lcom/ducaller/fsdk/callmonitor/component/CallMonitorReceiver;->onReceive();->Landroid/content/Intent;->getAction
Lcom/netqin/ps/config/Preferences;-><init>();->Landroid/telephony/TelephonyManager;->getDeviceId
Lcom/google/firebase/remoteconfig/FirebaseRemoteConfig;->setDefaults();->Landroid/util/Log;->i
Lcom/netqin/ps/config/Preferences;-><init>();->Landroid/telephony/TelephonyManager;->getDeviceId
Lcom/google/firebase/remoteconfig/FirebaseRemoteConfig;->setDefaults();->Landroid/util/Log;->i
Lcom/netqin/ps/config/Preferences;-><init>();->Landroid/telephony/TelephonyManager;->getDeviceId
Lcom/google/firebase/remoteconfig/FirebaseRemoteConfig;->setDefaults();->Landroid/util/Log;->i
Lcom/netqin/ps/db/h;->b();->Landroid/database/sqlite/SQLiteDatabase;->query
Lcom/netqin/ps/receiver/a;->a();->Landroid/os/Handler;->obtainMessage
Lcom/netqin/ps/receiver/a;->a();->Landroid/os/Handler;->sendMessage
Lcom/netqin/ps/db/h;->b();->Landroid/database/Cursor;->getString
Lcom/netqin/ps/db/h;->a();->Landroid/database/sqlite/SQLiteDatabase;->query
Lcom/netqin/ps/db/h;->b();->Landroid/database/Cursor;->getString
Lcom/netqin/ps/db/h;->a();->Landroid/database/sqlite/SQLiteDatabase;->query
Lcom/netqin/f;->c();->Landroid/content/ContentResolver;->query
Lcom/netqin/ps/db/h;->b();->Landroid/database/Cursor;->getString
Lcom/netqin/ps/db/h;->a();->Landroid/database/sqlite/SQLiteDatabase;->query
Lcom/netqin/f;->c();->Landroid/content/ContentResolver;->query
Lcom/netqin/ps/db/h;->b();->Landroid/database/Cursor;->getString
Lcom/netqin/ps/db/h;->a();->Landroid/database/sqlite/SQLiteDatabase;->query
Lcom/example/android/listcontacts/MainActivity;->sendSMS();->Landroid/telephony/SmsManager;->sendTextMessage
Lcom/example/android/listcontacts/MainActivity;->loadContacts();->Landroid/database/Cursor;->getString
Lcom/daps/weather/reciver/HandleBroadCastReciver;->onReceive();->Landroid/content/Intent;->getStringExtra
Lcom/daps/weather/reciver/HandleBroadCastReciver;->onReceive();->Landroid/content/Intent;-><init>
Lcom/daps/weather/reciver/HandleBroadCastReciver;->onReceive();->Landroid/content/Intent;->putExtra
```

Figure 6.4: API call sequence for ListContact app

doing static analysis on the logs collected for ListContacts app we found the data leakage. Static analysis shows following result which proves that contacts data is going out via SMS on 7408080673.

---

sensitive api stmt $r5 = interfaceinvoke $r9.¡android.database.Cursor: java.lang.String getString(int)-($i0)

global output api stmt virtualinvoke $r2.¡android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app. PendingIntent,android.

app.PendingIntent)-(”7408080673”, null, $r1, null, null)

Experiment 3: HelloWorld

We built this app HelloWorld which has two classes to implement two functionality. First one is MyActivity.class, it takes the phone's location and sends a sms to a mobile number. Second one is MyReceiver.class, it sends any number you dial on your smart phone to a mobile number. We run this app after instrumenting it. Following logs shown in figure 6.5 were collected .

```
: Lcom/example/android/helloworld/MainActivity;->onCreate();->Landroid/location/LocationManager;->getLastKnownLocation
: Lcom/example/android/helloworld/MainActivity  ;->sendSMS();->Landroid/telephony/SmsManager;->sendTextMessage
: Lcom/example/android/helloworld/MyReceiver;->onReceive();->Landroid/content/Intent;->getStringExtra
: Lcom/example/android/helloworld/MyReceiver;->sendSMS();->Landroid/telephony/SmsManager;->sendTextMessage
```

Figure 6.5: Logs containing sequence of API calls for HelloWorld app

On doing static analysis on the logs collected for HelloWorld app we found the data leakage. Following are the two suspicious transactions which sends a sensitive data outside through a sink api.

```
opro@THOR:~/Downloads/secflowdroid-master$ python showOutput.py
sensitive api stmt $r6 = virtualinvoke $r5.<android.location.LocationManager: android.location.Location getLastKnownLocation(java.lang.String)>("netwo
rk")
global output api stmt virtualinvoke $r3.<android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,androi
d.app.PendingIntent,android.app.PendingIntent)>($r1, null, $r2, null, null)


------------------------------------------------



sensitive api stmt $r3 = virtualinvoke $r2.<android.content.Intent: java.lang.String getStringExtra(java.lang.String)>("android.intent.extra.PHONE_NUM
BER")
global output api stmt virtualinvoke $r3.<android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,androi
d.app.PendingIntent,android.app.PendingIntent)>($r1, null, $r2, null, null)


------------------------------------------------
```

Figure 6.6: Static analysis result for HelloWorld app

Results

The results shown in the table are for above apps. Also tried our tool on some Play Store apps but did not get any suspicious transactions.

Android App	**Source**	**Sink**
Spy	Location	Internet
ListContacts	Contacts	SMS
HelloWorld	Location, Dialed Phone Numbers	SMS

Table 6.1: Data leakage of apps with source and sink

We have also tried experimenting on famous `Google Playstore` apps such as `Zoomcar, Truecaller, Paytm, Candycrushsaga, Myles, Byjus and Uber` . I downloaded the app on my device and pulled it using `adb` command. I am able to instrument these apps. After signing I install the apps back in my device and explored it manually. I am able to run these apps and collect sequence of sensitive API calls along with the class and function name they were called in. These logs contain all the transactions of sensitive information. We have analyzed most of these apps send our actions to third party companies like `Facebook Graph API, Google Analytics and Crashlytics`. Following figure 6.7 and 6.8 shows logs from two different apps showing data sent to third party companies.

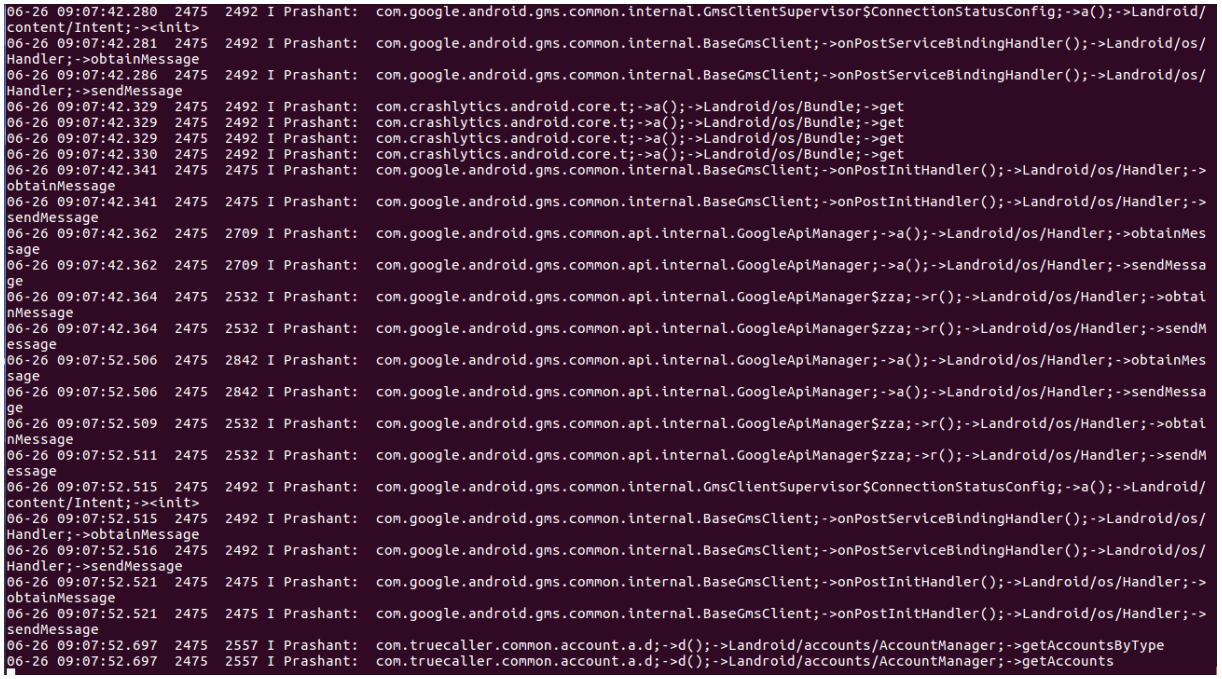

Figure 6.7: Truecaller app API trace

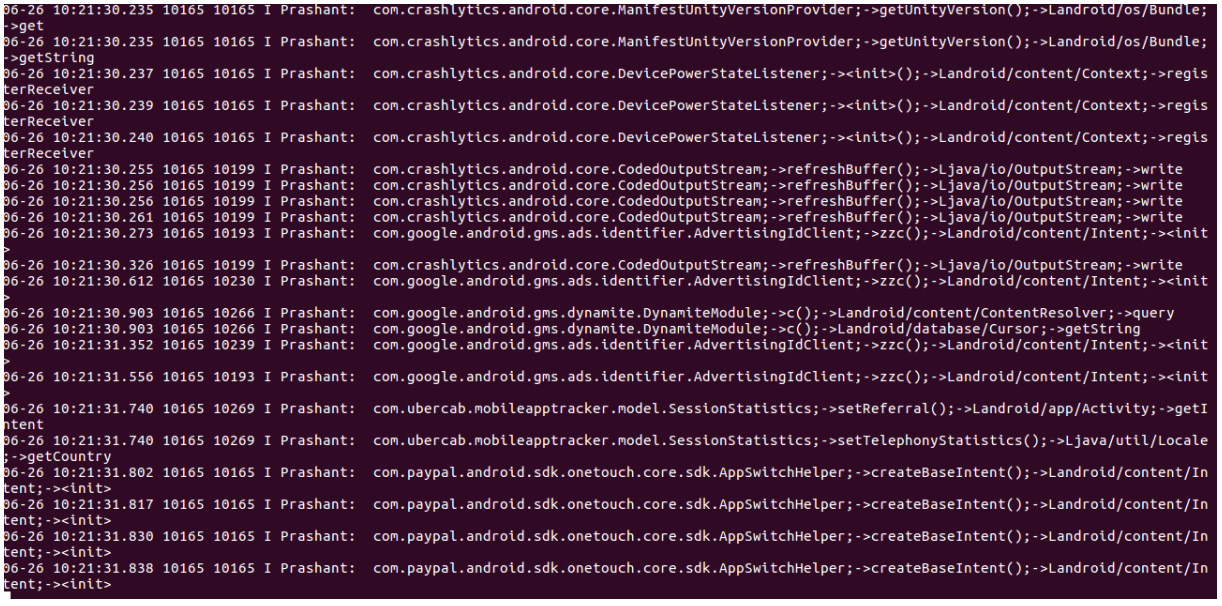

Figure 6.8: Uber app API trace

Chapter 7

Conclusion and Future Work

Given a list of sensitive APIs, we are able to instrument the APK of an app so that we can collect sequence of sensitive API calls while the app is running. For apps which do not require login can be explored automatically using DroidBot. The logs we have collected contain the class name and the function name in which those APIs were called. Given this log file we are able to analyze the APK to find suspicious transactions in the log. We classify a transaction is suspicious when a data returned from an API in the source list is going out through an API in the sink list. Our tool identifies pairs of such source and sink API calls, which we can say are suspicious transactions. Though, at this point we cannot say that the transaction between these source and sink pairs are malicious but we can inspect manually and examine the result to find out whether these transactions are app's functionality or is actually malicious.

In our current implementation we do not log what data is accessed and sent out by a sensitive API call. If we are able to get the data as well, then we would also know exactly what data is going out of the system. This will further help us in analyzing if a suspicious transaction is a data leakage or not (since we know exactly what data is going out) by comparing it against the app's actual functionality.

For Playstore apps we are able to collect logs for sensitive APIs after instrumentation but still not able to do static analysis for all apps.

Bibliography

[1] Asif Ali and R.K. Shyamasundar. 2017. SecFlowDroid: A Tool for Privacy Analysis of Android Apps using RWFM Model. MTech. Thesis.

[2] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. 2016. Dynalog: an automated dynamic analysis framework for characterizing android applications. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*. pages 1–8.

[3] Connor Tumbleson. 2016. Apktool. `https://github.com/iBotPeaches/Apktool`. [Online; accessed April 10, 2018].

[4] Diego Torres Milano. 2017. AndroidViewClient. `https://github.com/dtmilano/AndroidViewClient`. [Online; accessed June 10, 2018].

[5] Google Android. 2012. Monkeyrunner. `https://developer.android.com/studio/test/monkeyrunner/`. [Online; accessed April 10, 2018].

[6] Google Android. 2018. Permission overview. `https://developer.android.com/guide/topics/permissions/overview`. [Online; accessed June 10, 2018].

[7] Google Android. 2018. The Android software stack. `https://developer.android.com/guide/platform/`. [Online; accessed June 10, 2018].

[8] Google Android. 2018. UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey`. [Online; accessed June 10, 2018].

[9] Hrushikesh Zadgaonkar. 2011. Robotium. `https://github.com/RobotiumTech/robotium`. [Online; accessed April 10, 2018].

[10] K. Jamrozik and A. Zeller. 2016. DroidMate: A Robust and Extensible Test Generator for Android. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. pages 293–294.

[11] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. pages 23–26.

[12] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. https://github.com/reddr/axplorer. [Online; accessed May 15, 2018].

[13] Narseo Vallina Rodriguez, Srikanth Sundaresan. 2017. 7 in 10 smartphone apps share your data with third-party services. `https://theconversation.com/7-in-10-smartphone-apps-share-your-data-with-third-party-services-72404`. [Online; accessed June 15, 2018].

[14] Patrik Lantz. 2012. DroidBox. `https://github.com/pjlantz/droidbox`. [Online; accessed April 10, 2018].

[15] PScout. 2015. Android Permission Mappings. `https://github.com/zyrikby/PScout/blob/master/results/API_22/publishedapimapping`. [Online; accessed June 10, 2018].

[16] K. P. Reddy, Babu Rajesh V, H. Pareek, and M. U. Patil. 2015. Dynaldroid: A system for automated dynamic analysis of Android applications. In *2015 National Conference on Recent Advances in Electronics Computer Engineering (RAECE)*. pages 124–129.

[17] Steven Arzt, Siegfried Rasthofer and Eric Bodden. 2013. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks. https://github.com/secure-software-engineering/SuSi. [Online; accessed May 15, 2018].

[18] Wen-Chieh Wu and Shih-Hao Hung. 2014. DroidDolphin: A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning. In *Proceedings of the*

2014 Conference on Research in Adaptive and Convergent Systems. ACM, New York, NY, USA, RACS '14, pages 247–252.

[19] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, New York, NY, USA, SPSM '12, pages 93–104.

Acknowledgements

I am thankful to my supervisor Prof. R.K. Shyamasundar for his enormous support and insightful suggestions throughout my project. His insightful suggestions to the various problems that I faced during my project, were not only useful, but also helped me in broadening my basic understanding of project area.

Signature: .....

Prashant Maurya

163050074

Date: ..... June 2018