# SecFlowDroid: A Tool for Privacy Analysis of Android Apps using RWFM Model

**MTech. Thesis**

*Submitted in partial fulfillment of requirements for degree of*

**Masters of Technology**

*by*

**Asif Ali**
**Roll No : 143059009**

*under the guidance of*

**Prof. R.K. Shyamasundar**

**Department of Computer Science and Engineering**
**Indian Institute of Technology, Bombay**

Jun, 2017

# Dissertation Approval

This dissertation entitled **"SecFlowDroid: A Tool for Privacy Analysis of Android Apps using RWFM Model"**, submitted by **Asif Ali (Roll No: 143059009)** is approved for the degree of **Master of Technology** in **Computer Science and Engineering** from **Indian Institute of Technology Bombay**.

[illegible]

**Prof. R.K. Shyamasundar**
**Dept. of CSE, IIT Bombay**
**Supervisor**

[illegible]

**Prof. Ashwin Gumaste**
**Dept. of CSE, IIT Bombay**
**Internal Examiner**

[illegible]

**Prof. Virendera Singh**
**Dept. of Electrical Engineering, IIT Bombay**
**Internal Examiner**
**Chairperson**

# Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

[illegible]

Asif Ali
Roll No: 143059009

**Date:** June 22, 2017
**Place:** IIT Bombay, Mumbai

# Contents

## Abstract

Smartphones have become important part of our day to day life. It not only provides its traditional functionality, but also provides other functionalities. They are helpful in our daily life from managing our office life, social life as well as private life. Smartphones now contain our confidential data (such as credit card numbers, bank details etc) to private data (such as photos, videos, music, personal utilities etc). As the usage increases so are its vulnerabilities. Malware writers exploit these vulnerabilities to collect user's sensitive information.

Android dominates the smartphone market due to its huge collection of applications in its store. There are several malicious applications that can leak user's sensitive information and cause sever damage to the user. Even though Android uses permission-based access control system, but it cannot enforce fine-grained control over data flow causing data leakage problem. This dissertation describes a technique to check possible "leak" in apps. As "apps" are used without a formal manual usually, we use dynamic tracing technique to find out whether apps leak information through global logs or other apps or Intents of Android framework.
Broadly, the method consists of the following steps:

- Generate dynamic traces using tools like APIMonitor[12].
- Generate possible data flows between API calls by building "JIMPLE" code fragments between calls using tools like SOOT.[16].
- Analyze the flow of sensitive data between calls assuming the sensitive data (only two classes:- sensitive or otherwise) of the input and also the data stored on the Android by the user, we build the sensitive nature of the data using the recently proposed information flow security model RWFM[1].

Our approach rightly abstracts the possible leaks (tainting as well as information flow) and tracks the points of leakage as points from where they arise or accumulate. We have implemented our integrated tool called SecFlowDroid. We have been able to detect various leaks in several apps. Our approach has the distinct advantage of locating the points of poisoning of data and leaks as opposed to other approaches. Furthermore, ours is the only method captures both direct, indirect and detects the leaks.

# Chapter 1

# Introduction

Android runs each application in a limited sandbox. If an application needs to access resource outside its sandbox it has to explicitly request permission. Now depending upon the permission it can automatically allocated the resource or can check its permission request in manifest file. Most of the android developers uses third-party application in their applications. For example, 50% of free apps include third party libraries (like advertisement etc.). They may used by developers to provide various features at a significant reduced time and cost. For example, in app-purchases, PDF view, cloud computing etc. The support for development of third-party application is the only way to become the dominating player of market. Android has its own set of features which enables the developer to develop its application in easily. Some of its key features are inter-application message passing system and component re-usability but it has its own side-effects. Through message passing there are number of attacks possible which leads to sniffing, modification and theft of user's private data. In other words they can easily exploit the application security policies.
In the following, we discuss importance of Android Security and provide a brief survey of the work on Android Security from literature.

## 1.1 Importance of Android Security

It is known that Android is the primary target for mobile malware. Now more and more people are getting doubtful for applications which asks users for permissions without showing how they are going to use for it.When a user gives any information to the application they are trusting the app for its security. Every single can expose data if developers didn't proper care whether its a settings file, log file, data transmission over the network or web service. Developer should upload privacy policy of app that tells what is going to do with the data. The possible number of risk an Android can face are as follows:

- **Hacking**

  Hacking is unauthorized network access by third party not involved in communication. Depending upon the situation hacking information by third parties can use for their own goals and profits.

- **Spam Mailing**

  Mail is most valuable assets for online marketers. That's why most of the hacking attempt goes for emails which then tries to sell them in bulk.

- **Identity Theft**

  Hackers can create a fake app for phishing all your personal information.

- **Malicious Software**

  Privacy policy provided by the app developers are long with a purpose. They try to protect you from every possible breach but hacker tries to breach your security every possible way. Even a simple clock app can provide hackers access to your personal information.

## 1.2 Approaches for for Assuring Security of Android

There has been an enormous amount of literature survey on language-based security. Bibliography records some of the most prominent language-security literature. Most important of them are as follows:

1. **Android Developers**

   Android Developers[2] gives the structure, layout of Android System and how the applications designed and executed inside the dalvik system like we discussed in chapter 3 .It also gives the tutorial for beginners to design an application from writing your first Hello World application to uses of different android features like location, sms, wifi etc. It gives an overview how permission systems work in an android.

2. **Droidracer**

   Droidracer[4] was created. It has built by instrumenting android_4.0._r1 kernel source code so that application dynamic execution can be traced. It has three components UIExplorer, Trace Generator, Race Detector which scan each activity and created UI maps and events like clicking, long click, typing, back etc. Then these events gets executed and logged using Trace Generator. Trace Generator logs operation which are relevant for race detection like read, write, lock, for etc.) and third component Race Detector uses happens before relationship to detect race conditions. It gives a deeps insight how reads, writes happened in android environment and how race conditions can leak the information.

3. **DroidMate**

   DroidMate[5] which has a powerful test generator and explores an app and monitored the API calls during execution performs dynamic analysis of application. It makes decision tree based on GUI map of application. Decision tree includes (click, long click, text entry etc) and execute it based on the data provided using exploration strategy. It is fully automatic and does not require any human interaction. User can make their own exploration strategy, choose their own monitored methods and can log the API calls. It gives different results in the form of complete logs, summary of API calls which includes serial execution of these API calls and unique API calls happened during execution. It is a part of **Boxmate**[7] which is based on the sandbox mining technique, all the resource accessing APIs are collected using Droidmate user is allowed to check whether there is other resource request which is not present in the application permission requirement then this application is execute under sandbox allowing only these APIs which are collected from droidmate and checks for any behavior changes such as activation of any latent malware, targeted attacks, malicious updates, infections etc.

4. **DroidBox**

   DroidBox[6] is another way of dynamic analysis of application. Droidbox is written under google summer of code by Kun Yang. So instead of changing android kernel source code it embed monitoring code inside an application. It analyses application behavior by extracting application package *apk* file into smali format. Smali is an intermedite representation of application class file and Java file. It then inserts its monitoring code in smali format then repackaged file using apkil tool.

## 1.3 Our Approach

SecFlowDroid not only finds out possible leak but also tracks the point where it is leaking by using taint analysis and information flow. We follow a hybrid approach and utilize information from both the traces generated and source code. We use this information to detect possible leaks and also have ability to detect where exact leakage is happening. First we given dynamically execute given application in APIMonitor which generates a sequence of protected APIs that are invoked during application. We extract JIMPLE code of those methods which invoked protected APIs using SOOT-Android tool. We analyze these JIMPLE files and tracks information flow, including implicit flows, between API invocation using RWFM model and report misuse if found. We have tested SecFlowDroid on number of applications and results are promising.

## 1.4 Organization of Dissertation

The rest of dissertation are organized as follows: Chapter 2 describes RWFM model which is used for detection of information leakage. Chapter 3 describes Android Architecture which gives the basic structure of different components of android and how they interact. Chapter 4 describes Security in Android which show different component ensuring android security. Chapter 5 describes Our Approach to Discover Security Issues which shows our technique in more descriptive way. Chapter 6 includes an implementation of Simple Example considering our approach. Chapter 7 shows SecFlowDroid tool with full description on an example. Chapter 8 extracting code fragments between calls showing description of SOOT and JIMPLE. Chapter 9 shows description about frameworks that we are using in this method. Chapter 10 shows Experimentation. Chapter 9 gives a detail about the example we used during our experimentation. Chapter 10 gives results and conclusion. Chapter 11 gives detailed steps which runs SecFlowDroid. Finally chapter 12 gives future work.

# Chapter 2

# Readers Writers Flow Model

The Reader Writer Flow Model is said to be the milestone in the field of information flow control. The benefit of RWFM over existing model based on Denning's label model that has label structure is that is not only robust, simple and intuitive but also flexible enough to work at any level of abstraction.[3]
Following are the key characteristics of the RWFM model.

1. Explicitly captures the readers and writers of information.
2. Makes the semantics of labels explicit.
3. Immediately provides an intuition for its position in lattice flow policy.
4. Robust declassifications rules as compared to other decentralized models.

## 2.1 Definition

**Reader Writer Flow Model** (RWFM) is defined as a five tuple

$$(S, O, S \times 2^{\mathrm{S}} \times 2^{\mathrm{S}}, (*, \cap, \cup), (*, \supseteq, \subseteq))$$

where $S$ and $O$ denote the set of subjects and objects in the information system respectively. Note that in RWFM information flows upwards in the lattice as readers decrease and writers increase.[3]
From the above definition it is clear that RWFM is yet another derivative of Denning's lattice model. But this is not enough it should be homomorphic to all the necessary aspects of Denning's lattice model which is indeed a tricky part to do but ones we have done this. we can say that all those problem which can be performed by Denning's lattice model, can be done by RWFM model.

## 2.2 Completeness

RWFM is a complete model with respect to Denning's Lattice Model for studying information flow in an information system. RWFM follows a floating-label approach for subjects, with $(s, S, s)$ and $(s, s, S)$ as the "default label" and "clearance" for a subject respectively, where S is the set of all the subjects in the system. RWFM follows a static labeling approach for objects, with the exception of downgrading, which allowed to change the label of an object by adding readers.[3]
Notation $\lambda : S \cup O \to S \times 2^{\mathrm{S}} \times 2^{\mathrm{S}}$ denotes a labeling function. $A_{(\lambda)}(e), R_{(\lambda)}(e)$ and $W_{(\lambda)}(e)$ denote the first (owner/admin), second (readers) and third (writers) component of the security

class assigned to an entity(subject or object) e. Further, the subscript $\lambda$ is omitted when it is clear from the context. Note that $A_{(s)} = s$ always. **Note that in RWFM information flows upwards in the lattice as readers decrease and writers increase.**[3] In Figure 2.1 we are considering two points lattice $\{l_1, l_2\}$, with $l_1 < l_2$. Let $s_1$ and $s_2$, be the only subjects in the system i.e., $S = \{s_1, s_2\}$. Similarly, let $O = \{o_1, o_2\}$. Consider the policy $\lambda_1 : \lambda_1(s_1) = \lambda_1(o_1) = l_1$ and $\lambda_1 : \lambda_1(s_2) = \lambda_1(o_2) = l_2$. $s_1 \in R(o_1)$ because $\lambda_1(o_1) \leq \lambda_1(s_1)$ reduces to $l_1 \leq l_1$ which is true. $s_2 \in R(o_1)$ because $\lambda_1(o_1) \leq \lambda_1(s_2)$ reduces to $l_1 \leq l_2$ which is also true. Therefore $R(o_1) = \{s_1, s_2\}$. Similarly, we can derive the following labels on objects: $R(o_2) = \{s_2\}, W(o_1) = \{s_1\}$ and $W(o_2) = \{s_1, s_2\}$.The labels for subjects are as below: $R(s_1) = \{s_1, s_2\}, R(s_2) = \{s_2\}, W(s_1) = \{s_1\}$ and $W(s_2) = \{s_1, s_2\}$.[1]

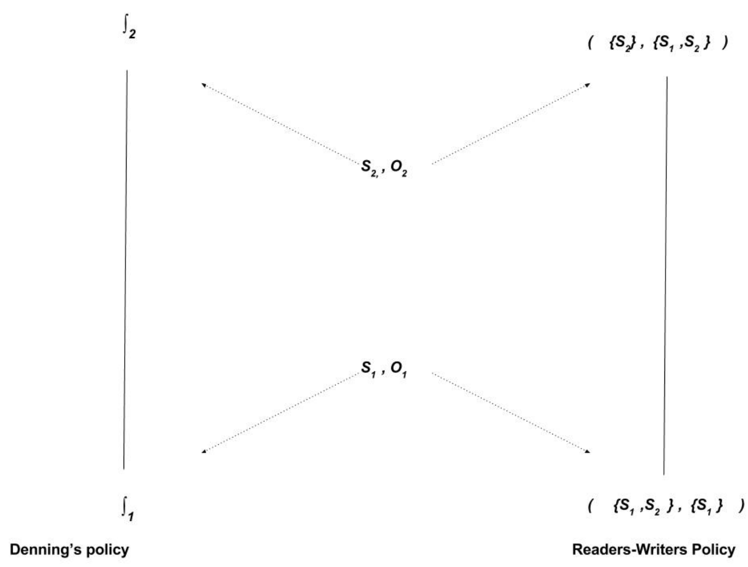


Figure 2.1: Denning's policy and corresponding readers-writers policy inferred

RWFM provides a state transition semantics of secure information flow, which presents significant advantages and preserves useful invariants.

**State of an information system is defined by the set of current subjects and objects in the system together with their current labels.**[1] State transition of an information system are defined by RWFM:

- Subject reads an object.[1]
- Subject writes an object.[1]
- Subject creates a new object.[1]

**RWFM** describes the conditions under which an operation is as safe follows:

**READ** Rule Subject $s$ with label $(s_1, R_1, W_1)$ request read access to an object $o$ with label $(s_2, R_2, W_2)$. If $(s \in R_2)$ then

change the label of $s$ to $(s_1, R_1 \cap R_2, W_1 \cup W_2)$

ALLOW

Else

DENY

**WRITE** Rule Subject $s$ with label $(s_1, R_1, W_1)$ requests writes access to an object $o$ with label $(s_2, R_2, W_2)$. If $(s \in W_2 \wedge R_1 \supseteq R_2 \wedge W_1 \subseteq W_2)$

then

ALLOW

Else

DENY

**CREATE** Rule Subject $s$ with label $(s_1, R_1, W_1)$ requests to create an object $o$.

Create a new object $o$, label it as $(s_1, R_1, W_1)$ and add it to the set of objects $O$.

Given an initial set of objects on a lattice, the above transition system accurately computes the labels for the newly created information at various stages of the transaction/ work-flow.

Chapter 3

# ANDROID ARCHITECTURE

In this section we take an overview of basic components of android architecture and how they work together.

Android[2] is software stack of open-source, Linux-based softwares which are created for wide range of devices. It is roughly divided into five sections and four main layers as shown in figure 3.1.[21]

1. **The Linux Kernel**

   It is the foundation of Android platform which provides basic functionalities such as threading and low-level memory management.

2. **Hardware Abstraction Layer (HAL)**

   It acts as a middleware between hardware capabilities and higher level Java API framework. It contains various library module which provides interfaces to specific hardware component such camera or bluetooth module. When any API make a call to access device hardware, Android system loads that module for that hardware.

3. **Android Runtime**

   ART is written for low-memory devices to run multiple virtual machines by executing DEX files. Some of the features provided by ART are JIT compilation, optimized garbage collection (GC), better debugging support.

4. **Native Libraries**

   Many component such as HAL and ART of Android system require to run libraries that are written in C and C++. Android provides Java Native Interface (JNI) to work with native libraries to apps. For example, you can add or manipulate 2D and 3D graphics using Java OpenGL API in your app. Android provide Android NDK[17] to directly access native libraries to your code.

5. **Java API Framework**

   Android provides entire functionality by APIs that is written in Java language. These acts as a building block to create a Android app by reusing the core, modular system components and predefined services.

6. **Application**

   Android provides some of core apps like contacts, calendars, browsers etc. It is resided on top of the hierarchy and meant to serve end-user.

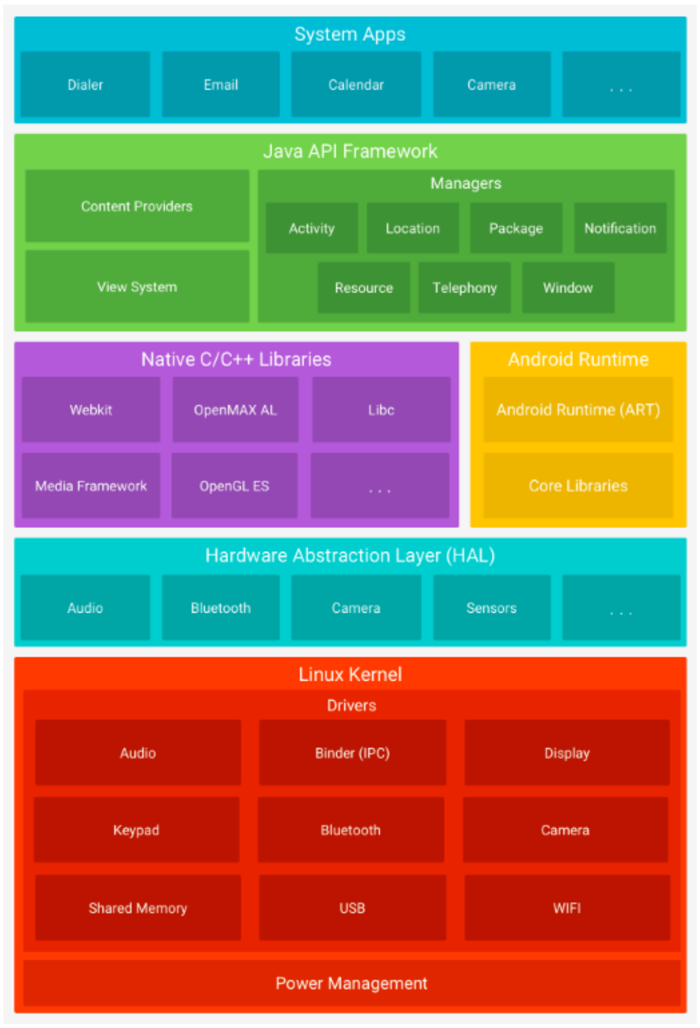


Figure 3.1: Android Software Stack

## 3.1 App Components

Another way of seeing an android application is group of different component interacting with each other in possible number of ways. To use a component, application has to explicitly request to particular component in intent filter under application's manifest file, which defines the set of components and permission an application required during its entire execution.

There are four types of application components which are as follows:

- **AndroidManifest.xml**

  It contains all the resources and permissions that applications is needed during execution. It is like a control system which interact with top components (involving services, content providers and broadcast receivers) and tell them what to do.

- **Activities**

  Its a fundamental block of building any android application. User interact with an app using activities and create smooth user experience if carefully managed. An activity has its own life cycle. User can go back and forth between the stages of life cycle. Activity provides number of interfaces that provide the state changed information.

- **Services**

  A service is basically a background process that is mainly used for task which takes long time. Any application can start the service and that service continues to run in the background even if that application goes to the background. For example, services are used for activities like downloading a file that needs no user interaction.

- **Content Providers**

  Content providers act as a mediator between application data request and data provider such as databases or file. Data are requested in the form of query (like SQL query). It is used to manipulate the private data of applications. It can also used to share your private data with other apps securely. It presents data in the form of tables similar to relational database. Different content providers used to provide different types of data like Contacts Provider used for accessing contacts similarly Calendar provider used to query related to calendars.

- **Broadcasts**

  Applications can send data in broadcast manner to android system as events or intents. Apps can register itself to different broadcasts using intent filter so that when any send any broadcast with specific type android system lists the app which has particular broadcast registered.

## 3.2 Permission System

Some of android permissions implemented as a Linux permission such as Network or External Storage which allows to access these resources in Unix groups, while other permissions are implemented under Android layer where files permission are given in the form of user id or group id. Every android application has a manifest file ($AndoidManifest.xml$) which contain various resource and permission requirement that application is need in the execution. Application during installation request the system to grant these resources or permission to use it. Android system then notifies the user that application is going to use these resources. User can agree on these permissions or can deny at that time. Some applications do not install until they get full permission they require during runtime.

### 3.2.1 Requesting Permissions

Every app has a distinct ID which is based on Linux user ID and group ID. Every has app ID has separate privileges. Linux isolates the apps from these IDs. Android system uses the sandboxing technique in each application runs in a sandbox and require to explicitly request the resources and data needed outside their sandbox. They required permissions for resources that is not provided in the basic sandbox which is declared in manifest file. Depending upon how sensitive the permission the android may grant it automatically or may prompt the user or rejects it.

System permissions are divided into two important levels. These are as follows:

- **Normal Permissions**

  These permissions are granted automatically by the system. It covers an area where your application need resources or data outside sandbox but that data has less chance to leak user's private information. These permission include accessing or changing Network, Wifi state, bluetooth, setting alarm, wallpaper, time-zone, vibrate etc.

- **Dangerous Permissions**

  It involves permission that is related to accessing user's personal data or resources or which could affect the user's personal data or can affect the functioning of other's apps. These permissions require explicitly written in the manifest file. It includes getting user's location, contact information, reading and writing calendar, sending SMS, Call phone,read or write External Storage etc.

There is also a third level of permission which is considered more often.

- **Signature Permissions**

  Signature permission is granted only when an another application which has installed on the system has already has the requested permission and has the same signature. It is useful to control component access to small set of applications developed by the same developer.

### 3.2.2 App Signing

All APKs($.apkfile$) must be signed before installation with a certificate. These certificates are formed by the public/private key pair. Public key acts as a fingerprint that uniquely associates with private key. When any user signs the app, the public key acts as fingerprint for the user that has its private key and facilitates in further updates of application. The main purpose of certifying an app is to distinguish the authors. These certifications allow the system to grant or deny access to signature permissions.

### 3.2.3 User IDs and file access

Whenever an app install in the android system, each package of an app is given as ID which is unique across the whole device like Linux user ID. It remains constant throughout package life span. The same package can have different ID on the different device.
Android security implemented at the process level i.e., the same process can not run two different package code in the same time. Developer can share the ID using the $sharedID$ attribute in $AndroidManifest.xml$ file. By assigning the same shared user ID between two packages, security of those two packages are treated as same and are treated as the same app.

## 3.3 Intents

[18] Android made of several components which are communicates each other using Intent object. Intents are messaging objects that allows components to request functionality from other components. Intent allow the component to interact with components of same application or components of different applications in several ways. It support s intra-application and inter-application communication. For example you can use intent to start an external activity of different application. There are mainly two types of intents as follows:

### 3.3.1 Types of Intents

- **Explicit Intents**

  Explicit intents are used to start an activity within your own app. It requires to explicitly mention the name of the activity or service you want to start. For example, start a download, loading a file in the background etc.[18]

- **Implicit Intents**

  Implicit intents are used to start an external activity. It doesn't require the name instead it require general action to perform. For example showing location of any user, you can generate implicit intent and broadcast it. The activities which are capable of handling it using intent filters can receive and start.[18]

# Chapter 4

# Security In Android

Android has lots of security features similar to Linux kernel. It provide secure inter-process communication facility so that process should not run in isolation with other processes.[19]

## 4.1 System and Kernel Security

- **Linux Security**

  Foundation of android is Linux kernel. Linux kernel provides Android with several security features including: user based permission model, process isolation , secure interprocess communication.

- **Application Sandboxing**

  Each application in an android system runs in a sandbox. The system uses a unique UID to distinguish between different processes. Application sandbox is in the Linux kernel from which security model can extend to native code and operating system applications. An application runs in restrictive permission model in which application is able to use some predefined permission but if tries to do malicious like trying to get data from other process, or open a new application then operating system protects against these acts because application does not have enough permissions.

- **System Partition and Safe Mode**

  The android divides storage into different partitions. The partitions which contains system files such as Android kernel, operating system libraries can't be modified and set to read-only. When user starts the device in safe mode, third party applications can only be launched manually by the user.

- **Cryptography**

  Android provides a set of cryptographic APIs. These include commonly used primitives AES, DSA, RSA, SHA and APIs for higher level security protocol like SSL and HTTPS.

- **Interprocess Communication**

  Processes can communicate each other like they communicate in UNIX systems. They can use any of the medium: File system, local sockets or signals.

  Android provides IPC mechanism as follows:

  - **Binder**
    It is a remote procedural call mechanism which is light weight for the system and delivers high performance communicating in process and cross process calls.

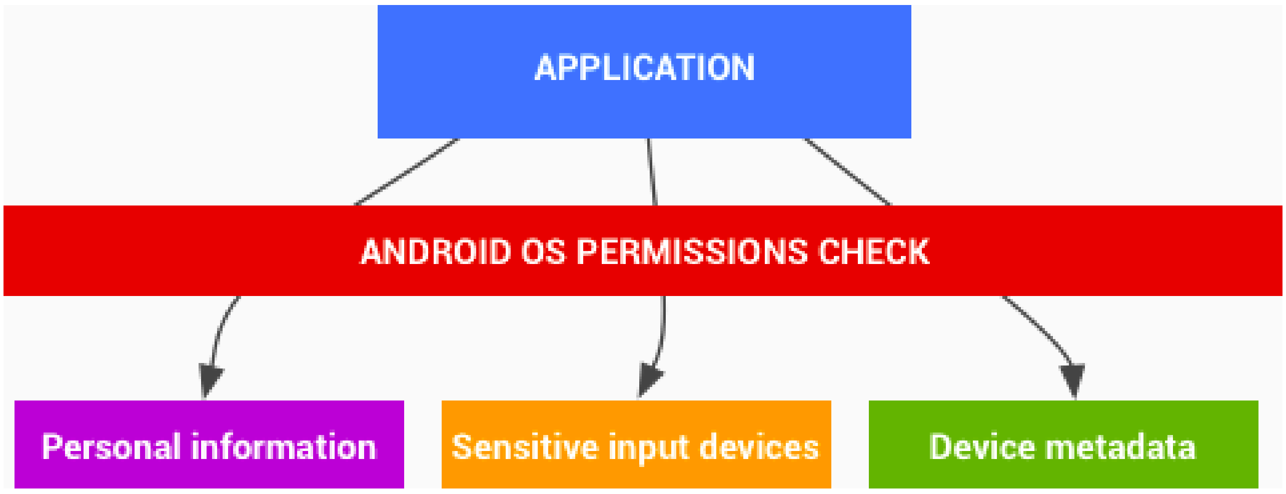


Figure 4.1: Access to Sensitive Data available through protected APIs

- **Services**
  As discussed above provides interfaces to communicate between processes using binder.
- **Intents**
  Intents are messaging object object like we discussed above.
- **Content Providers**
  Content providers considers as a building blocks of Android system. It captures the data and provide to the applications using ContentResolver interface. For example, it used to access the contact list of the user and can be used to read the user's inbox message.

- **Cost-Sensitive APIs**

  Cost Sensitive API (like Telephony, SMS/MMS, NFC Access etc) are APIs that can generate for the network or the user. Android system placed these APIs into a set of protected APIs that is controlled by the android system.

- **Personal Information**

  APIs which provide personal information are placed under protected APIs under android system. Android can share that is accumulated during application to the third party applications. Applications which does this required to go under permission checks of Android OS to protect its data as shown in figure 4.1[20]

- **Sensitive Data Input Devices**

  Sensitive data input devices includes camera, microphone or GPS. To access these devices by third party applications user must explicitly provide access to these devices through permission system of Android OS. For these devices at the time of installation, the installer prompt a dialog box asking user to approve permission requested by the sensor.

- **Device Metadata** Android also restrict access to data that is not normally sensitive but indirectly may reveal user sensitive information. For example, to access user's operating system logs, phone number or browser history, installer prompt a dialog box asking for the permission application can access.

## 4.2 Example Illustrating Subtle Leak

We go through through source code of "GPS_SMS" application developed as a toy app. This application show how despite fulfilling all the android permission requirement, still it is leaking location data. This application has the following permission:

- Contacts
- Location
- SMS

And the function we are analyzing is *BtnClick* function under *MainActivity* class

```
public void BtnClick(View arg){
gps = new GPSTracker(MainActivity.this);
if(gps.canGetLocation()){
        double lat;
        double longt;
        String str;
        lat = gps.getLatitude();
        longt = gps.getLongitude();
        str = "Your Location is Lat: "+lat+" Long: "+longt;
        SmsManager smsManager = SmsManager.getDefault();
        String phoneNo = "123456789";
        smsManager.sendTextMessage(phoneNo, null, str, null, null);
        }
}
```

The above source code shows that sensitive information acquired as **getLatitude()** and **getLongitude()** and send as text message to phone number "123456789" using **sendTextMessage()**

**As we can see in our analysis that sensitive information like GPS Location will flow to a SMS from which it can be send to anyone. This makes this application leaky in nature.**

# Chapter 5

# Our Informal Approach to Discover Security Leaks

Static and dynamic code analysis are performed during source code reviews. Static code analysis is done without executing any of the code while dynamic code analysis relies on how the code behaves during execution. Code behavior can referred as calling the API pattern as which API is invoked and in what manner. When performing comprehensive source code reviews, both static and dynamic testing should be performed. Static analysis source code testing is adequate for understanding security issues within program and can usually pick up about 85% of the flaws on the code. Dynamic code review has the additional ability to find security issues caused by the code's interaction with the other components like SQL databases, application servers or Web services. (Parameters are sent to back-end servers for processing, which could be modified before returning). Dynamic code reviews, presented with a wide range of inputs and security tests, will generally pick up about 85% of the vulnerable loop holes present in the code. Combining both types of code reviews should pick up about 95% of the flaws.Our approach consists of following steps:

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

## 5.1 Dynamic Analysis Using API Monitor

Dynamic analysis picks vulnerable points at runtime. We are using APIMonitor[12] to dynamically analyze the app and to trace out what are the sensitive APIs are invoked during execution.

APIMonitor which is a part of droidbox project. APIMonitor has configuration file which listed all the api calls that have to logged, so that it extract the apk file and insert monitor code for only those APIs. Given the list of APIs in the configuration file, it output to a log file containing monitored API calls that application is executing during its execution. Let API calls be $X_1, X_2, ..., X_n$, where each $X_i$ is in the form of $\{ApplicationClassName, ApplicationFunctionName, APIClassName, APIFunctionName\}$.

## 5.2 Extracting JIMPLE

We have the API calls, class name and function name in which they invoked in source code. Now we have to generate the JIMPLE source code of only these classes.

We filter out classes to convert into $JIMPLE$ using soot-infoflow-android framework then we have to check data flow between every consecutive API $(X_i, X_j)$ calls using RWFM model

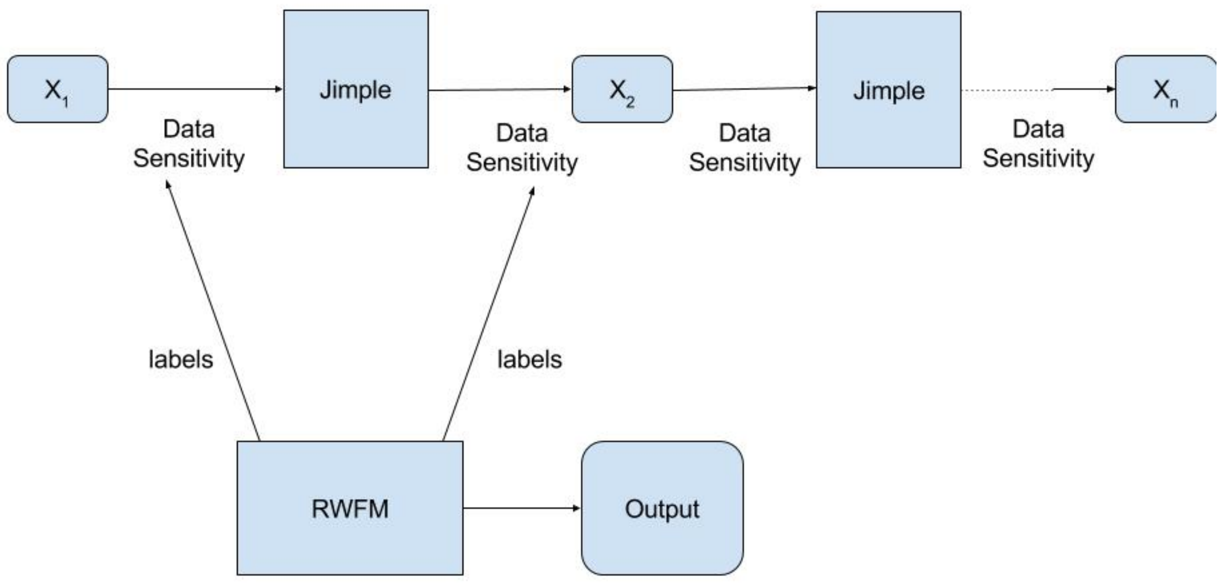


Figure 5.1: Sensitivity Tracing Between Two Consecutive API Calls

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.

## 5.3 GPS_SMS Leak Detected by Our Approach

Let us consider the example from section 4.2. We are considering information flow between sensitive API calls. We run the APIMonitor on the source code of the above function which gives the following output:

1. APICall 1: **getLatitude()**

2. APICall 2: **getLongitude()**

3. APICall 3: **sendTextMessage()**

Dynamic analysis through APIMonitor is not enough to declare as information misuse as data between sensitive API calls can be exchanged with the normal data. Thus, we have to analyze data flow between two consecutive API calls so that sensitivity of data in between API calls can be determined. Source code of above example written as follows:

**lat = gps.getLatitude();** $\lambda(lat) = \lambda(lat) \oplus \lambda(gps)$
**longt = gps.getLongitude();** $\lambda(longt) = \lambda(longt) \oplus \lambda(gps)$
str = "Your Location is Lat: "+lat+" Long: "+longt;
$\lambda(str) = \lambda(str) \oplus \lambda(lat) \oplus \lambda(longt)$
SmsManager smsManager = SmsManager.getDefault();; $\lambda(smsManager) = L$
String phoneNo = "123456789"; $\lambda(phoneNo) = L$
**smsManager.sendTextMessage(phoneNo, null, str, null, null);**
$\lambda(smsManager) = \lambda(phoneNo) \oplus \lambda(str)$
}
}

**As we can see in our static analysis that sensitive information like GPS Location will flow to a SMS from which it can be send to anyone. This makes this application leaky in nature which can easily be detected by the RWFM model.**

# Chapter 6

# SecFlowDroid : A Tool for checking Security Threats

In this chapter, we discuss the implementation of the tool. As highlighted already, our approach consists of three main steps:

## 6.1 Dynamic Analysis

We are using dynamic analysis to trace the sensitive API calls that invoked during execution of application. We are using APIMonitor as we referred formally in section 1, detailed description is given as below:

### 6.1.1 APIMonitor

Most of dynamic analysis based on the instrumentation of android kernel source like we see in Droidracer[4] or executing the app on their own environment[5]. APIMonitor[12] which is a part of DroidBox[6] edit the application package(apk) file. It extracts the apk file into *smali* format and insert the monitor code for API calls written as separate $defaultapicollection$ file using *apkil* package, sign the apk file with its own signature and repackage it with file name with *_new* appended to it. This repackage file when run on an emulator and interact with the user gives the desired logs with *DroidBox* tags on it. It gives the precise information that which API is invoked under which function and under which class and what is the serial of the calls during execution.

With reference to the source code of GPS_SMS application given in section 4.2, we run the APIMonitor with the following steps:

- We edited the APIMonitor so that it logs the protected APIs with the useful meta data (application class and method which this API invoked) then use edited APIMonitor so that protected APIs can be traced out. APIMonitor instruments the apk file given a list of APIs which you want to trace in a separate file.
- Clear the logs and capture the logs having tag **DroidBox**. User will interact with application and logs will come as shown below:

  - **V/DroidBox( 608): Lcom/example/asif/gpstracking/GPSTracker;->getLocation()Landroid/location/Location;->Landroid/location/LocationManager;->getLastKnownLocation ...**
  - **V/DroidBox( 608): Lcom/example/asif/gpstracking/GPSTracker;->getLocation()Landroid/location/Location;->Landroid/location/**

**LocationManager;->requestLocationUpdates ...**
**...**
From above we can infer the following:

* *Lcom/example/asif/gpstracking/GPSTracker* gives the class name of the application in API is invoked
* *getLocation*() gives local function of the class in which API is invoked.
* *getLastKnownLocation* and *requestLocationUpdates* are APIs that are invoked sequentially during interaction of the user with the application.

## 6.2 Static Analysis

From dynamic analysis we get order of execution of protected APIs. Now we have to filter the classes we found out in dynamic traces and apply RWFM model. For that we convert apk file into JIMPLE form. Trace the data flow of each statement of the converted program using RWFM model and report misuse if any happen. RWFM rules can be referred in appendix. Referring to the source code of GPS_SMS application in section 4.2, its RWFM labels with JIMPLE statements can be found by the following steps:

- Run the soot It will process each statement according to the rules given in the appendix. A snippet for above application is given below:

  - **$r0 := @this: com.example.asif.gpstracking.MainActivity**
    $\{owner = [com.example.asif.gpstracking], readers = [public, com.example.asif.gpstracking], writers = [com.example.asif.gpstracking]\}$
    **...**
  - **$r5 = $r0.<com.example.asif.gpstracking.GPSTracker: android.location.Location location>**
    $\{owner = [com.example.asif.gpstracking], readers = [ com.example.asif.gpstracking], writers = [public, com.example.asif.gpstracking]\}$
    **...**
  - **virtualinvoke $r4.<android.telephony.SmsManager: void sendTextMessage ...**
  - ******************* misuse ***************
  - **subLabel {owner=[com.example.asif.gpstracking], readers=[ com.example.asif.gpstracking], writers=[public, com.example.asif.gpstracking]} tries to write {owner=[com.example.asif.gpstracking], readers=[ public, com.example.asif.gpstracking], writers=[com.example.asif.gpstracking]}**
  - ******************* misuse ***************

## 6.3 RWFM Label for Transitive Leak

Here we are assigning the RWFM label to each of the above listed api with the application that can use these api on the basis of the their permission set. With the help of labelling of api calls we can not only maintain the state of an running application but we can also define the set of applications with which it can interact without leaking its information. In this way we can also restrict the transitive leak.Figure 6.1 shows a working of my project. RWFM rules for JIMPLE analysis is given in appendix.

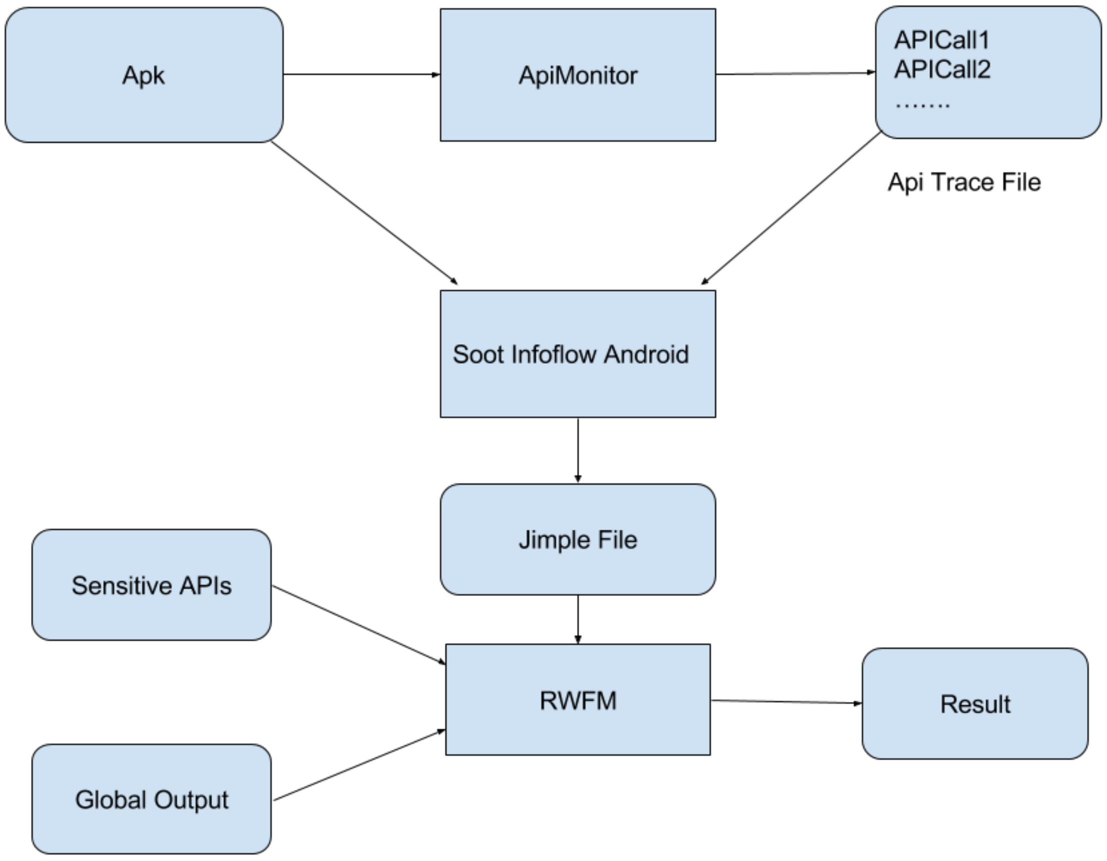


Figure 6.1: Static Analysis of an Application

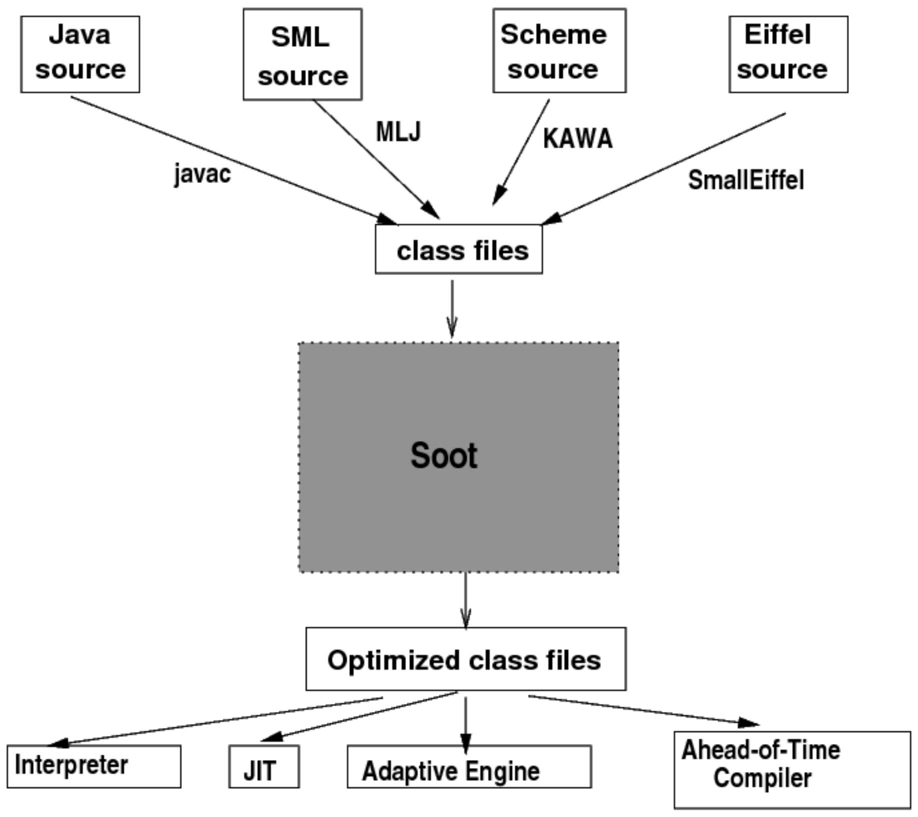


Figure 6.2: An overview of Soot and its usages

## 6.4 Extracting Code Fragments between Calls

Android applications develop in Java language. However they use native language code ( like C/C++, Python etc). but core development language is Java. Java is chosen due to its many attractive features such as platform independence, execution safety, garbage collection and object orientation. These features help in application development but have their own cost. For example, applications in Java are much slower as compared to written in C/C++. To use these we have to pay performance penalty, sophisticated optimizations and runtime systems are required. SOOT[13] provides a great deal of effort in optimizing Java bytecode.

### 6.4.1 SOOT

SOOT framework converts Java bytecode into intermediate representation and provide a set of JAVA APIs for optimizing it. Some of the Java bytecode instructions consume more time as compared to other instructions. For example, object allocation, virtual method calls, interface calls, catching exceptions. Soot uses method inlining, and static virtual method call resolution, which reduces a use of these time consuming bytecodes. Java's safety execution guarantees that all potentially illegal memory access are checked before execution . In some situations, it can be determined at compile time particular checks are unnecessary[14]. Soot communicate these bounded checks to Java Virtual Machine using annotation so that it can speed up by not performing these redundant checks. The Soot framework is used as follows: (shown in figure 6.2).

1. Bytecode is produced from a variety of sources, such as *javac* compiler.

2. This bytecode is fed into SOOT, and SOOT transforms and/or optimizes the code and produces new class files.

3. This new bytecode can then be executed using any standard Java Virtual Machine (JVM) implementation, or it can be used as the input to a bytecode $\rightarrow$ C or bytecode $\rightarrow$ native-code compiler or other optimizers.

The best combination of optimizations implementation can yield a speed to 38%. SOOT provides the three intermediate representations BAF, JIMPLE and GRIMP which are stack-based intermediate representation.

## 6.5 Soot-Infoflow-Android

SOOT-Infoflow-Android[16] is a framework built on SOOT[13]. Soot needs a starting point for its processing like Main function in Java or C/C++. Our RWFM model based on data-flow analysis and data analysis require an entry point. Android doesn't have any a single starting point instead it has many entry-points. Android defines the complete life cycle for all the components in an application. There are four different kinds of components in an applications. These are defined as: *activities* are single focused user actions, *services* perform background tasks, *contentproviders* define database-like storage, and *broadcastrecievers* listen for global events. All these components are implemented by custom class which is registered in $AndroidManifest.xml$ file and overwriting the life cycle methods. The Android framework calls these methods to start/stop of component or to pause/resume it, depending on environment needs. An application can contain multiple components like three activities and one service. However these activities run sequentially but their order can't be predetermined. Flowdroid[11] which soot-infoflow-android is a part generate *dummymainmethod* in every component in every order of individual component life cycles and callbacks is possible.

### 6.5.1 JIMPLE

Jimple is a typed and compact 3-address code representation of bytecode. It features traditional optimizations such as copy propagation and virtual method resolution that Java require. It consists of following grammar:

#### 6.5.1.1 Design : Stackless

JIMPLE is based on stackless 3-address code form. Every statement in JIMPLE follows 3-address code. 3-address code is a standard representation where instructions are kept as simple as possible, and most of them are of the form: $x = y\ op\ z$.

In JIMPLE, variables can be local or stack variable. The stack has been eliminated and replaced by extra variables as stack variables. Local variables representing stack variables are prefixed by dollar($) sign.

#### 6.5.1.2 Design : Compact

JIMPLE is very compact as compared to Java. Its compactness makes it suitable for analysis and optimizations. Java bytecode has around 200 different bytecode instructions while JIMPLE has only 19 instructions. Less the number of instructions, the more sophisticated the language for analysis because it is far easy to construct test cases needed for analysis.

#### 6.5.1.3 Description

JIMPLE is a typed and compact 3-address code representation of bytecode which ideal for performing optimization and analysis. There are essentially 11 different types of JIMPLE statements. Their representation and RWFM labeling are discussed in section 6.2.

# Chapter 7

# Using SecFlowDroid

Steps to analyze application using SecFlowDroid is given in flowchart as shown in figure 7.1. We have maintained the source code of edited APIMonitor[23] and SOOT-Android[24] in github. As it is mentioned README file of each package. Run the SecFlowDroid in the following step:
**Input**: apk file
**Output**: File showing misuse from source API call to sink API call

1. Create an emulator using AVD Manager with the following configurations:
   - **Device:** Nexus 4
   - **Target:** Android 4.1.2- API Level 16
   - **CPU:** ARM(armeabi-v71)
   - **RAM:** 2048MB
   - **Internal Storage:** 1024MB

2. run command:

   $$./run.sh \ < apk\_file\_name >$$

   It will generate the following:
   - Create $apimonitor_output$ in the same directory which contains $orig_smali$ and $new_smali$ file.
   - Create new apk file with $\_new$ append with $< apk\_file\_name >$.
   - Install the application in the emulator.
   - Clears the logs and capture the logs having “DroidBox” tag.
   - Save the log file having name $< apk\_file\_name >$ *.log* in *log* directory when user after interacting with the application.

3. User interact with the application where protected APIs are recorded in the log file.

4. Run the SOOT-Android framework with following command:

   $$java \ - android.jar \ path/to/android/platforms \ < apk\_file\_name > < apk\_log\_path >$$

   It will give us an output file which shows data flow between source APIs as sensitive data APIs and sink as global output APIs.

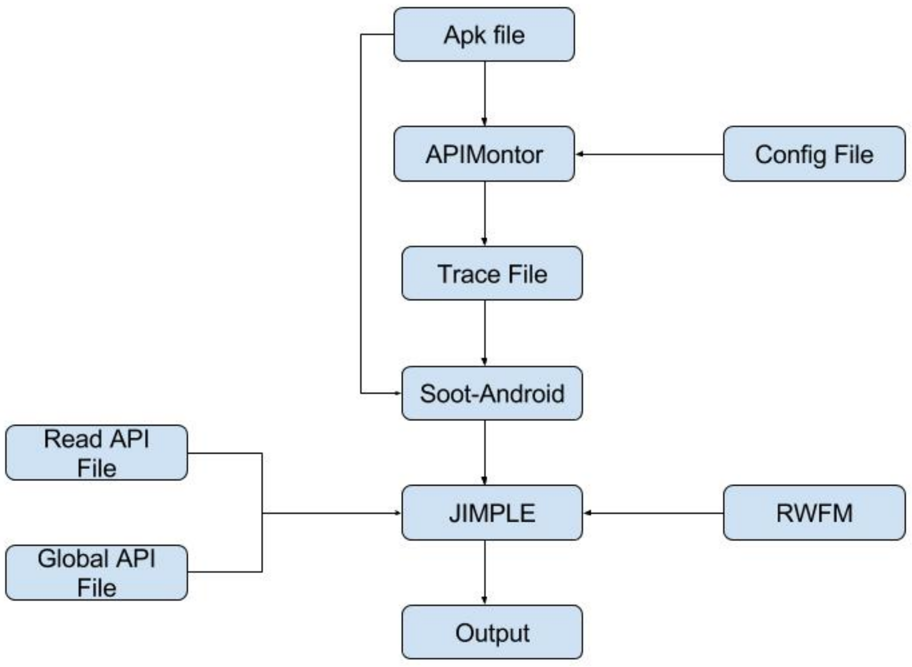


Figure 7.1: Flowchart : Using SecFlowDroid

# Chapter 8

# Experimental Analysis and Results

On these classes we implemented RWFM model as follows.

- Initially Subject $S$ is given default label as $\{\{ApplicationName\}, \{ApplicationName, Public\}, \{ApplicationName\}\}$
- Read the $JIMPLE$ class files according to API calls $(X_1, X_2, ..., X_n)$ given in trace file.
- Label the variables from starting of the file to API call $X_i$ according to RWFM implementation to each type of statement defined above, label the variable calling or participating in $X_i$. If $X_i$ is private change the labels of participating variable to private label $\{\{ApplicationName\}, \{ApplicationName\}, \{ApplicationName, Public\}\}$.
- Trace the data flow from $X_i$ to $X_j$ and report if any misuse happen according RWFM rules.

## 8.1 Experiment 1: GPSTracking

Created a demo application which get location of an emulator using $Location.requestLocationUpdates$ using a callback function and show it to the user as a Toast message using $\$Toast.makeText$ API call. Now using the APIMonitor which gives and convert it to jimple, applying labels and checking information misuse gives the desired result.Application and leaking shown in figures 8.1,8.2.

## 8.2 Experiment 2:Horoscope

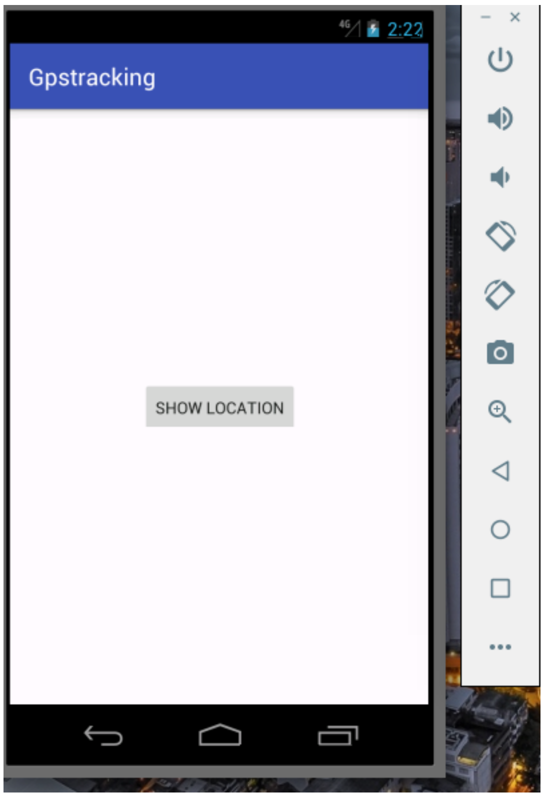
We download the horoscope application from google app store. We analyze the application and did not find any misuse.

## 8.3 Experiment 3: ContactWriter

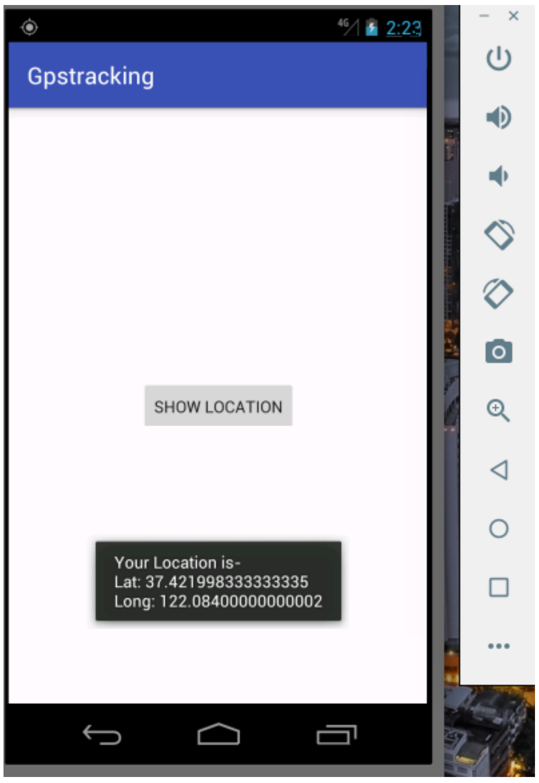
Created an application which initially opens an activity that shows a button. Upon pressing it callbacks a writer function which collects the contact by querying the content resolver using $ContentURI$ and writes it to the vulnerable file using $java.io.OutputStream.write()$. This information can be accessed by any malicious application and misuse this information.

## 8.4 Experiment 4: ReadSMS

Created an application which opens an activity having a button pressing of which calls a callback function which read the user's SMS by querying to the user's inbox using $contentResolver.query()$ and show it to the user using $Toast$.



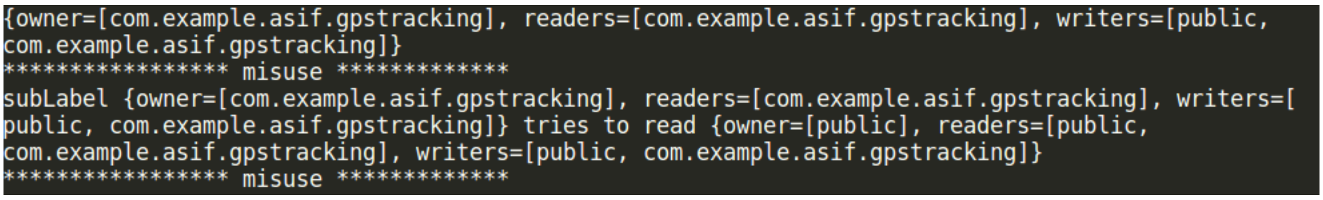
(a) Starting Activity



```
Your Location is-
Lat: 37.421998333333335
Long: 122.08400000000002
```

(b) Data leakage

Figure 8.1: Leakage of User Location

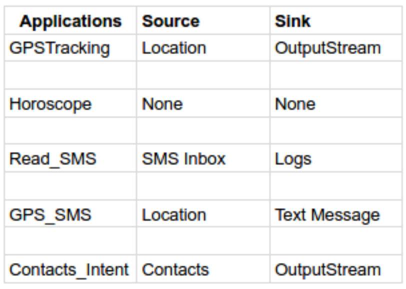


```
{owner=[com.example.asif.gpstracking], readers=[com.example.asif.gpstracking], writers=[public,
com.example.asif.gpstracking]}
***************** misuse *************
subLabel {owner=[com.example.asif.gpstracking], readers=[com.example.asif.gpstracking], writers=[
public, com.example.asif.gpstracking]} tries to read {owner=[public], readers=[public,
com.example.asif.gpstracking], writers=[public, com.example.asif.gpstracking]}
***************** misuse *************
```

Figure 8.2: Leakage in Log File

## 8.5 Experiment 5: GPS_SMS

Created an application which reads user's location using *Location.requestLocationUpdates* and send it to external user using a pre-entered number using *SmsManager* After analyzing the applications we can clearly see that out five applications only one application is not leaky. We compare it with FlowDroid[11] which is a taint analysis tool whose results are shown in the figure 8.5. We have done our experiment on the number of android application ranging from open source application to proprietary application. This give us wide range of spectrum to test our model on different types of application.



| Applications | Source | Sink |
|---|---|---|
| GPSTracking | Location | OutputStream |
| | | |
| Horoscope | None | None |
| | | |
| Read_SMS | SMS Inbox | Logs |
| | | |
| GPS_SMS | Location | Text Message |
| | | |
| Contacts_Intent | Contacts | OutputStream |

Figure 8.3: Data Leakage of Apps

## 8.6 Result

So far we have analyzed five applications, we will add further analysis to the result. From results, it can be seen that out of 5 applications 4 applications are leaky (as shown in fig 8.3). We build a number of test cases for Java language from JIMPLE our approach becomes more and more sophisticated and can work on any application.

### 8.6.1 FlowDroid

FlowDroid[11] is used for static analysis of application code. It is context, field, object sensitive analysis tool. It uses **soot**[13] and **Heros**[10] which helps it create call-graph from which it ensures flow and context sensitivity. It uses Heros which works on IFDS based flow functions which ensures field and object sensitivity. It uses Source and sink file for taint analysis which is created by SuSi which a machine learning approach to find source and sink for information leakage data flow.

### 8.6.2 Comparison with FlowDroid

We have verified our analysis with FlowDroid and the results are encouraging (as shown in fig 8.5). Our approach is accurate and fast as compared to FlowDroid. SecFlowDroid has following advantages as compared to FlowDroid:

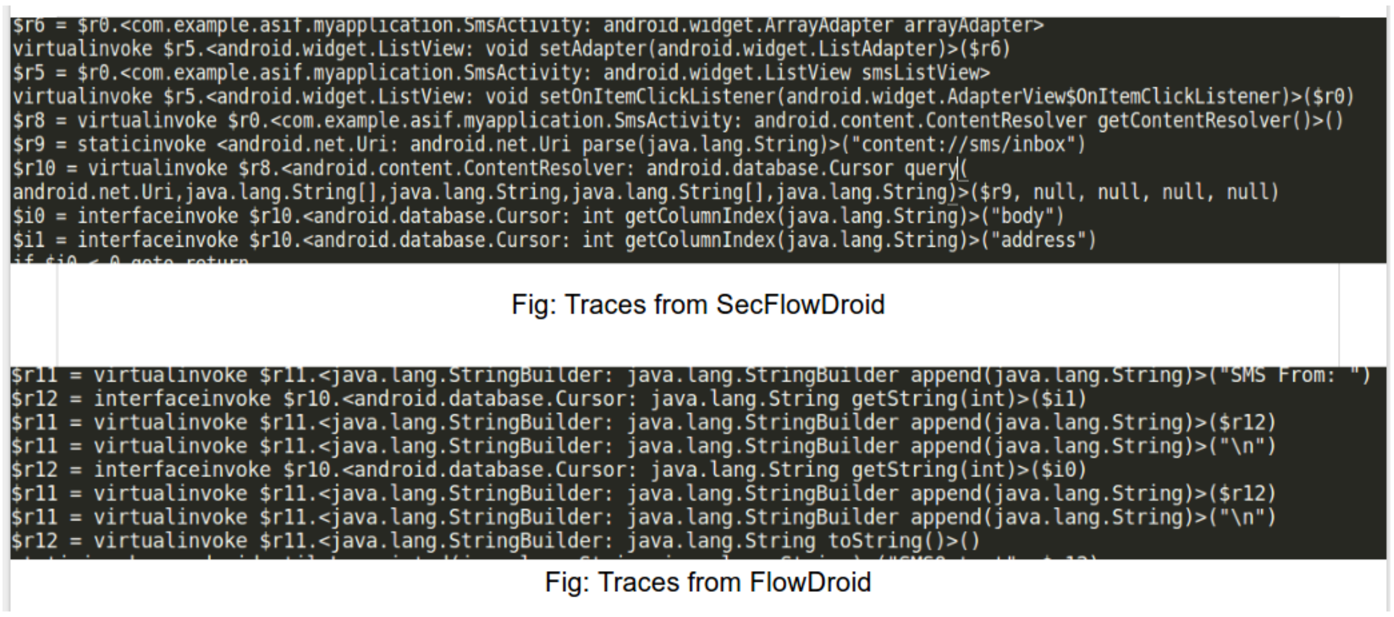
1. Finer Analysis
   **GPSTracking**

   As we analyze GPSTracking application under both FlowDroid and SecFlowDroid. SecFlowDroid is able to detect leak while FlowDroid does not give any output.

2. Gives Point of Leakage
   **Read_SMS**

   For Read_SMS application, SecFlowDroid gives more accurate point of leakage as compared to FlowDroid. as shown in figure Fig 8.4

3. More readable output



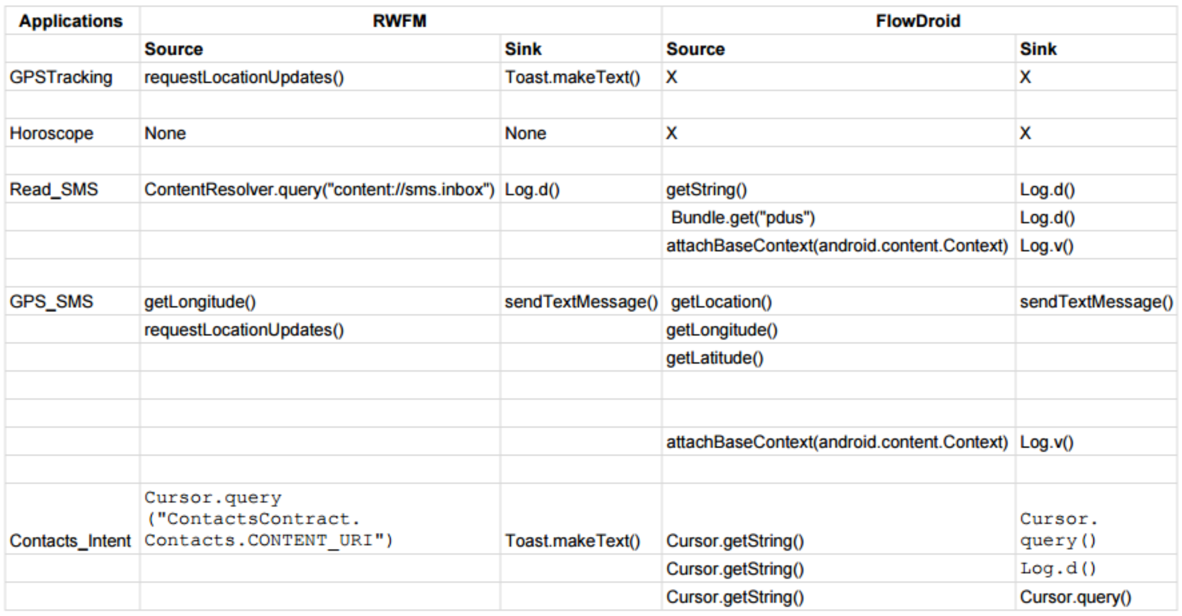
```
$r6 = $r0.<com.example.asif.myapplication.SmsActivity: android.widget.ArrayAdapter arrayAdapter>
virtualinvoke $r5.<android.widget.ListView: void setAdapter(android.widget.ListAdapter)>($r6)
$r5 = $r0.<com.example.asif.myapplication.SmsActivity: android.widget.ListView smsListView>
virtualinvoke $r5.<android.widget.ListView: void setOnItemClickListener(android.widget.AdapterView$OnItemClickListener)>($r0)
$r8 = virtualinvoke $r0.<com.example.asif.myapplication.SmsActivity: android.content.ContentResolver getContentResolver()>()
$r9 = staticinvoke <android.net.Uri: android.net.Uri parse(java.lang.String)>("content://sms/inbox")
$r10 = virtualinvoke $r8.<android.content.ContentResolver: android.database.Cursor query(
android.net.Uri,java.lang.String[],java.lang.String,java.lang.String[],java.lang.String)>($r9, null, null, null, null)
$i0 = interfaceinvoke $r10.<android.database.Cursor: int getColumnIndex(java.lang.String)>("body")
$i1 = interfaceinvoke $r10.<android.database.Cursor: int getColumnIndex(java.lang.String)>("address")
if $i0 < 0 goto return
```

Fig: Traces from SecFlowDroid

```
$r11 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("SMS From: ")
$r12 = interfaceinvoke $r10.<android.database.Cursor: java.lang.String getString(int)>($i1)
$r11 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r12)
$r11 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("\n")
$r12 = interfaceinvoke $r10.<android.database.Cursor: java.lang.String getString(int)>($i0)
$r11 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r12)
$r11 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("\n")
$r12 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.String toString()>()
```

Fig: Traces from FlowDroid

Figure 8.4: Comparison between point of Leakage between SecFlowDroid and FlowDroid of Read_SMS



| Applications | RWFM | | FlowDroid | |
|---|---|---|---|---|
| | Source | Sink | Source | Sink |
| GPSTracking | requestLocationUpdates() | Toast.makeText() | X | X |
| | | | | |
| Horoscope | None | None | X | X |
| | | | | |
| Read_SMS | ContentResolver.query("content://sms.inbox") | Log.d() | getString() | Log.d() |
| | | | Bundle.get("pdus") | Log.d() |
| | | | attachBaseContext(android.content.Context) | Log.v() |
| | | | | |
| GPS_SMS | getLongitude() | sendTextMessage() | getLocation() | sendTextMessage() |
| | requestLocationUpdates() | | getLongitude() | |
| | | | getLatitude() | |
| | | | | |
| | | | | |
| | | | attachBaseContext(android.content.Context) | Log.v() |
| | | | | |
| Contacts_Intent | Cursor.query ("ContactsContract. Contacts.CONTENT_URI") | Toast.makeText() | Cursor.getString() | Cursor. query() |
| | | | Cursor.getString() | Log.d() |
| | | | Cursor.getString() | Cursor.query() |

Figure 8.5: Comparison with FlowDroid

Chapter 9

# Conclusion & Future Work

## 9.1 Conclusion

We tried to solve the problem of data leakage among applications. We scan through different approaches that uses dynamic and static approach. We find out a hybrid approach that uses both static and dynamic analysis. The contribution of this thesis is to give a novel approach to detect information leakage among applications. We presented RWFM model used to track data flow between different components of application. We used APIMonitor for dynamic analysis and SOOT-Android framework for static analysis. SOOT-Analysis converts apk file to JIMPLE code which has its own grammar. We define RWFM label of each statement for JIMPLE grammar. The implementation of this framework is roughly around 2000 lines of code. Unlike other approaches, it uses labels to track data which can be used for inter-application communication and is more faster than other analysis. We till now analyzed five applications and verified by FlowDroid, which is taint analysis tool and find results encouraging. We are going to generalize the model so that it can implement every constructs in Java language.

## 9.2 Future Work

We have considered most of the constructs that developer can use to write application to interact with data. We have to consider other possible constructs to complete it. The whole analysis depends upon an APIMonitor which uses dynamic analysis so we have to execute application in all possible ways so that every most of APIs can be covered. For that we can write a UIAutomater which explores the application in every possible ways. Android source code will be more secured when it includes SecFlowDroid approach.

# Chapter 10

# Appendix

RWFM Rules for JIMPLE statements are as follows:

1. **AssignStmts** is most used in JIMPLE instructions. It is represented in four different representations: assign a *rvalue* to a local, or an *immediate*(a local or a constant) to a static field, to an instance field or to an array reference. Here *rvalue* can be field access, array reference, an immediate or an expression. Any complex statement in JIMPLE can be broken down into smaller simple statements with locals being made as temporary. For example field copy $this.x := this.y$ can be broken down to two JIMPLE statements, a field read and field write ($tmp := this.y$ and $this.x := tmp$). If there is a variable which does not have a label, we create the label according to RWFM model rule.

   **RWFM Representation** For assignments $V_l := V_r$ we check $READ$ of $V_r$ with respect to $V_l$ . If $V_r$ is a field variable like $this.x$ and assignment is like $tmp = this.x$ we check $READ$ of $x$ with respect to $tmp$. If $V_r$ is an expression or statement in the form of: $x := y\ op\ z$. we take the $LUB(y, z)$ and check $READ$ with respect to $x$.

2. **IdentityStmts** are mostly used at the time method entry. It is used to define locals with special values such as *this* value or parameters. For example : $l_0 := @this$ where this value is normally random value.

   **RWFM Representation** For statements like $l_0 := @this$ we assign default label for $l_0$ and for $l_0 := parameter0$ we check $READ$ of $@parameter0$ with respect to $l_0$.

3. **IfStmt** and **gotoStmt** represents conditional and unconditional jumps respectively.

   **RWFM Representation** For If *cond* then *Stmt*; for every $v_i \in cond$ we take $LUB(v_i)$ and check $READ$ this $LUB$ with all the left operand inside *Stmt*. Process all statement in *Stmt* according to *AssignStmt*, *IdentityStmt* and $IfStmt$.

   IfStmt also handles If/Else statements; for else body we handle same as we handle statements in If body.

4. **InvokeStmt** represent invoking a function call without an assignment to local. (The assignment happen after the function returns due to *AssignStmt* with *InvokeExpr* on the right hand side of an assignment variable). Some are InvokeStmt which does not use any reference and there are some which does not have any left hand operand.

   **RWFM Representation** For statements like $V_l := V_r.functionCall(args_1, args_2, ...)$ we take the least upper bound $LUB(Label(V_r), Label(args_1), Label(args_2), ...)$ and check $READ$ for $LUB$ with respect to $V_l$ and update the labels of $V_r$ and all the arguments $args_i$. For statements like $V_r.functionCall(args_1, args_2, ...)$ and $functionCall(args_1, args_2, ...)$ take $LUB$ of all participating variables and update each variable labels to $LUB$.

5. **SwitchStmt** can either be a *lookupswitch* or a *tableswitch*. The *lookupswitch* takes a set of integer values whereas *tableswitch* takes a range of integer values. Target destination is declared by labels.

   **RWFM Representation** This will handle same as IfStmt.

6. **MonitorStmt** represents enter/exit monitor byte codes.

7. **ReturnStmt** can either be return void or return with some value which is specified by a constant or a local.

   **RWFM Representation** For statement like $returnVal$ return the Label($Val$) and for statement *return* we return $NULL$.

8. **ThrowStmt** represents explicit throwing of an exception.

   **RWFM Representation** For statement like $newThrowException(e)$ and $catch\{catchblock\}$; we will do the same thing as we did in IfStmt where $e$ becomes *cond* and *catch* becomes *Stmt* for IfStmt.

9. **BreakPointStmt** and **NopStmt** represents *breakpoint* and *nop* instructions respectively.

   **RWFM Representation** There is no representation for NopStmt

# Bibliography

[1] NV Narndra Kumar and RKS, IEEE Bigdata and Cloud Computing, Dec 2014 b. ibid, ACM CCS 2015, Poster, October 2015 c. S Susheel, NV Narendra Kumar and RKS, 11 Int Conf on Information System Security (ICISS 2015), Dec 2015 4. Nv Narendra Kumar RKS, RWFM Model: Soundness and Completeness, manuscript 2015.

[2] Android Developer `https://developer.android.com/training/index.html`

[3] Denning, D. E. (1976). A lattice model of secure information flow. Communications of the ACM, 19(5), 236-243.

[4] Maiya, Pallavi, Aditya Kanade, and Rupak Majumdar. "Race detection for Android applications." ACM SIGPLAN Notices. Vol. 49. No. 6. ACM, 2014.

[5] Jamrozik, Konrad, and Andreas Zeller. "DroidMate: A Robust and Extensible Test Generator for Android."

[6] DroidBox `https://code.google.com/archive/p/droidbox/wikis/APIMonitor.wiki`

[7] Jamrozik, Konrad, Philipp von Styp-Rekowsky, and Andreas Zeller. "Mining sandboxes." Proceedings of the 38th International Conference on Software Engineering. ACM, 2016.

[8] Boxmate `http://www.boxmate.org/`

[9] Lam, Patrick, et al. "The Soot framework for Java program analysis: a retrospective." Cetus Users and Compiler Infastructure Workshop (CETUS 2011). Vol. 15. 2011.

[10] Heros `http://sable.github.io/heros/`

[11] Arzt, Steven, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." Acm Sigplan Notices 49.6 (2014): 259-269.

[12] APIMonitor `https://github.com/pjlantz/droidbox/tree/master/APIMonitor`

[13] Vallée-Rai, Raja, et al. "Soot-a Java bytecode optimization framework." Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research. IBM Press, 1999.

[14] Rajiv Gupta. 1990. A fresh look at optimizing array bound checking. In Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90). ACM, New York, NY, USA, 272-282. DOI=http://dx.doi.org/10.1145/93542.93581

[15] Steven S. Muchnick. Advanced Compiler and Implementation. Morgan Kaufmann, 1997

[16] Soot-Infoflow-Android `https://github.com/secure-software-engineering/soot-infoflow-android`

[17] Android NDK `https://developer.android.com/ndk/index.html`

[18] Intent and Intent Filters https://developer.android.com/guide/components/intents-filters.html

[19] System and Kernel Security https://source.android.com/security/overview/kernel-security

[20] Android Application Secutiry https://source.android.com/security/overview/app-security

[21] Platform Architecture https://developer.android.com/guide/platform/index.html

[22] Importance of Android App Security http://www.zdnet.com/article/google-engineers-stress-importance-of-android-app-security-at-io/

[23] ModifiedAPI https://github.com/asifazamali/apimonitor_modified

[24] SOOT-Android https://github.com/asifazamali/soot-infoflow-android_modified