# Precise Static Analysis of Taint Flow for Android Application Sets

Amar Shirish Bhosale

May 9, 2014

Heinz College
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Advisor**
Robert C. Seacord
CERT
Division of the Software Engineering Institute
Carnegie Mellon University

*Submitted in partial fulfillment of the requirements*
*for the degree of Master of Science in Information Security Policy and Management*

**Keywords:** Static analysis, taint analysis, Android, security

*For my loving parents, Sharmila and Shirish*

## Abstract

Malicious and unintentionally insecure Android applications can leak users' sensitive data. One approach to defending against data leaks is to analyze applications to detect potential information leaks. This thesis describes a new static taint analysis for Android that combines and augments the FlowDroid and Epicc analyses to precisely track both inter-component and intra-component data flow in a set of Android applications. The analysis takes place in two phases: given a set of applications, we first determine the data flows enabled individually by each application and the conditions under which these are possible; we then build on these results to enumerate the potentially dangerous data flows enabled by the set of applications as a whole. Our method requires analysis of the sourcecode or bytecode of each app only once, and results can be used for analysis of tainted flows possible for any combination of apps. This analysis can be used to ensure that a set of installed apps meets the user's data flow policy requirements. This thesis describes our analysis method, implementation, and experimental results.

## Acknowledgments

I would like to express my sincere gratitude to my advisor, Robert C. Seacord, for giving me the opportunity to work on this interesting topic. A very special thanks to Dr. William Klieber and Dr. Lori Flynn of CERT[1] for letting me be a part of their team. I thoroughly enjoyed being a part of such a motivated and talented team.

Furthermore, I would like to thank Dr. Limin Jia and Dr. Lujo Bauer for helping us (Will, Lori, and me) in writing the workshop paper "Android Taint Flow Analysis for App Sets" on which this master's thesis is based.

Last but not the least, I would like to thank the CERT editor, Carol J. Lallier for her helpful comments.

[1]Division of the Software Engineering Institute, Carnegie Mellon University

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

One billion Android devices (phones and tablets) are projected to be sold in 2014 [1]. Android apps (applications) are distributed using a marketplace model in which developers publish apps that users can conveniently download and install from app stores. Users can download apps from the official Google Play store[1] and other markets such as the Amazon Appstore[2] for Android. These apps can potentially access a variety of sensitive information, such as a user's location, contacts, and the unique device ID (IMEI). Users can install highly trusted apps such as banking apps and free social networking apps. A significant concern in this setting is exfiltration of sensitive data, which may violate users' privacy and allow undesired tracking of users' behavior. It has been shown that popular Android apps leak sensitive information, including location, device ID, phone number, and the SIM (subscriber identity module) card ICC-ID [2]. In 2010, the SMS Message Spy Pro app disguised itself as a tip calculator and leaked all SMS messages, call logs, browser history, and GPS location to a third party [3]. In 2011, the Skype app was discovered to leak profile and IM information [4], which other apps could read. In 2014, a malicious app that

[1] `https://play.google.com/store/apps`
[2] `http://www.amazon.com/mobile-apps/b?node=2350149011`

allows remote access for use of microphones and cameras was found in the official Google Play store, and a toolkit for making similar apps has been found for sale in underground forums [5].

Ensuring that apps in the app market are secure is not a trivial undertaking [6]. When developers upload apps to the Google Play Store, the Google Bouncer [7] app security analyzer performs a time-limited dynamic analysis on the uploaded apps to detect malicious behavior [8]. Although this effort is encouraging, it has had limited success [9].

Most mobile computing platforms, including Android, use a permission model to attempt to limit the privileges of apps, including their ability to access and exfiltrate sensitive data. However, existing permission systems fail to prevent sensitive data from being leaked [2].

Data can be leaked not only by malicious apps but also by legitimate apps if they do not follow secure coding practices [10]. Additional analysis of data flow is necessary to determine whether sensitive data remains within expected boundaries and to ensure that untrusted data does not contaminate trusted data repositories. Such an analysis is often called *taint analysis*. This thesis focuses on determining whether data can flow from a sensitive data source to an undesired data sink. For instance, for a smartphone, sensitive data sources include the phone's unique identifier, SMS message store, photos, and apps that provide services such as banking. Undesired sinks for such data include the network API, external storage, and other untrusted applications.

Taint analysis can be either static or dynamic. For instance, TaintDroid performs real-time taint tracking to dynamically detect data leaks [2]. In contrast, FlowDroid performs a highly precise taint flow static analysis for each component within an Android application [11, 12], and Epicc [13] performs a specific kind of flow analysis between Android components. However, little work is documented on statically analyzing data flows of a system composed of several applications [14]. Such static analysis is important because data from a source might reach a sink only after passing through one or more compo-

nents [15, 16]. Without a multicomponent data flow analysis, malicious apps (colluding malicious apps, a single malicious app with multiple components, or a malicious app which exfiltrates sensitive data from an unintentionally leaky app) could evade detection, and developers of unintentionally leaky apps may not discover security problems that should be fixed.

## 1.2 Contribution

We developed "DidFail" (Droid Intent Data Flow Analysis for Information Leakage), a new static analysis tool that combines and augments the state-of-the-art tools FlowDroid [11] and Epicc [13] to precisely report undesired information flows between interacting apps. Our approach requires analysis of the source code or bytecode of each app only once and leverages the results to detect potentially dangerous flows enabled by all subsets of analyzed apps. The tool is available at

`https://www.cert.org/secure-coding/tools/didfail.cfm`

We tested our prototype tool on three test apps developed by our team as well as on three relevant apps from the DroidBench[3] benchmark suite.

## 1.3 Terminology

We define a *source* as an external resource (external to an app, not necessarily external to the phone) from which data is read and a *sink* as an external resource to which data is written. Example sources include device ID, contacts, photos, and current location. Example sinks include the Internet, outbound text messages, and the file system.

[3] `http://sseblog.ec-spride.de/tools/droidbench/`

## 1.4 Structure of the Thesis

The rest of the thesis is organized as follows. Chapter 2 provides a background on static analysis and the tools that our analysis extends. It also discusses some Android-specific concepts that are related to this work and introduces a motivating example. Chapter 3 describes our two-phase analysis design, and Chapter 4 describes its implementation. We tested our prototype analyzer with two application sets, and the results are discussed in Chapter 5. We discuss the limitations of our analysis in Chapter 6, related work in Chapter 7, and conclusions in Chapter 8.

# Chapter 2

# Background

This chapter briefly covers some theoretical underpinnings of our analysis. It starts with an overview of Android and then briefly describes static analysis. An overview of the static analysis tools that we build upon is followed by a motivating example set of two apps that contain data flow across each other that existing analyses cannot precisely track.

## 2.1 Android Overview

Android apps are written in the Java programming language and are compiled to a Dalvik bytecode using the Android Software Development Toolkit (SDK). The SDK enables the developer to create an application package (APK), which is an archive with the *.apk* extension. This APK file can be installed on Android devices. The Android application sandbox isolates apps from each other and prevents them from accessing each other's private data. Because each app runs in a process sandbox, apps must explicitly share resources and data by declaring the permissions they need to access shared resources and data outside the sandbox.

However, Android does not completely isolate apps from each other, because apps often need to share data. For example, assume a user wants to take a photograph, edit it using a photo-editing app, and then share it with her friends using a social networking app. This

process requires data to flow across isolated (sandboxed) applications.

### 2.1.1 Components

Android apps can be composed of one or more of the following components:

- An *activity*, which provides a screen with which users can interact to perform a task
- A *service*, which can perform long-running operations in the background and does not provide a user interface
- A *content provider*, which manages access to a central repository of data
- A *broadcast receiver*, which allows apps to register for system and application events

### 2.1.2 Intents

The primary method for inter-component communication, both within and between applications, is via *intents*. For the photo-sharing example, the information (a photo) can flow across multiple components via intents as follows:

$$CameraApp_{activity} \xrightarrow{intent} PhotoEditingApp_{activity} \xrightarrow{intent} SocialNetworkingApp_{activity}$$

An intent can be an *explicit* intent for which the sender explicitly states the receiving component, or an *implicit* intent, for which the intent specifies the *action* to perform and the *category* or *data* on which the action should be performed. The Android OS determines the receiver on the basis of the intent filters defined in the manifest file of all apps installed on the device. Every app has a manifest file, *AndroidManifest.xml*, which contains information about all components and their capabilities. The intent filters are used by the Android OS to determine if any components within the app are eligible to receive a particular implicit intent. It uses a set of filter-matching rules[1] while resolving such intents. A component may also send an intent to itself. A component can be made accessible to other apps by setting the `exported` attribute in the manifest file to `true`. If the `exported` attribute is

[1] `http://developer.android.com/guide/components/intents-filters.html#Resolution`

not defined, the OS makes the component available to other apps by default if an *Intent Filter* is associated with the component. Access to this component can be restricted by using *permissions*. Permissions are also declared in the manifest file, and a component can be accessed by an app only if it has the required permission. Permissions are granted by the user during the app installation and are enforced by the OS at runtime.

### How Intents Are Used

Intents can be used to launch activities; to bind, start, and stop services; and to broadcast information to broadcast receivers. Intents can be sent and received only between activities, services, and broadcast receivers and not between content providers. Table 2.1 lists commonly used methods to send intents to and receive intent results from activities. We use the term "`startActivity` family" to define methods that can be used to launch activities.

| **Purpose** | **Method signature** |
|---|---|
| Launch an activity | `startActivity (Intent intent)`<br>`startActivity (Intent intent, Bundle options)` |
| Launch an activity and expect to receive a result | `startActivityForResult (Intent intent, int requestCode)`<br>`startActivityForResult (Intent intent, int requestCode, Bundle options)` |
| Return data to the caller | `setResult (int resultCode)`<br>`setResult (int resultCode, Intent data)` |
| Read result set by the callee in caller | `onActivityResult (int requestCode, int resultCode, Intent data)` |

Table 2.1: Commonly used intent-related methods for *inter*-activity communication

### How Intents Can Be Misused

Various studies [17, 18, 19] done in the past have highlighted how intents can be misused to carry out component hijacking and intent-spoofing attacks. Component hijacking attacks occur when a malicious app receives an intent that was intended for another app but not explicitly designated for it, that is, when implicit intents are used. The attack can

result in leakage of sensitive data when the intent is received by an unintended recipient. Intent-spoofing attacks, by contrast, can be used to send spoofed commands (via intents) to legitimate apps, causing loss of secure control of the affected apps.

## 2.2 Static Analysis

Static analysis is a program analysis method in which the source code (or bytecode) is analyzed without executing it. Dynamic analysis, on the other hand, involves studying the application behavior by running it in an environment, for instance, analyzing an Android app by running it on an Android device.

Static analysis allows examining all possible execution paths in the program, not just those invoked during execution. This is especially valuable in security analysis, because attacks often exploit apps in unforeseen and untested ways. However, predicting the program behavior without executing it is a nontrivial problem. By reducing it to the *halting problem*, it is possible to prove that finding all possible ways of executing any arbitrary nontrivial program is an *undecidable problem*. That is, there cannot possibly be any program that will always correctly predict the program behavior. However, static analysis can provide useful results by approximating some facets of the actual execution of a program [20].

One of the techniques of implementing static analysis is analyzing the data flow. Taint analysis is a special type of data-flow analysis that tracks data along the program execution path. In this technique, sensitive data is marked with a taint at the source, and this taint is allowed to propagate further through all program execution paths. Presence of this taint at predefined sinks is used to establish a flow between the source and the sink. This flow can be used to detect sensitive data leaks from source to sink.

### 2.2.1 Static Analysis Tools

Our analysis is built upon the FlowDroid and Epicc analyses and the Soot framework.

### FlowDroid

FlowDroid is an open-source static analysis tool for Android apps that is *context-, flow-, object-, field*-sensitive and *lifecycle*-aware [11]. It uses an IFDS (interprocedural, finite, distributive, subset) framework [21], which reduces the program analysis problem to a simple graph reachability problem. It accurately models the Android life-cycle, including callback methods (more detail next paragraph), and precisely maps the user-defined UI elements with the code. These features make FlowDroid highly sound and precise.

Analyzing Android apps is more complicated than analyzing Java programs because these apps run within the Android framework. Java programs have a single entry point, the `main()` method. But Android apps can have multiple entry points, that is, callback methods that are implicitly called by the Android framework. These methods are not directly connected in the app source code. FlowDroid precisely handles this problem by creating a `dummyMain()` method, which accurately emulates the Android lifecycle for each component by connecting the callback methods. It extends the Soot framework to obtain a precise call graph based on *Heros* [22], an IFDS framework implementation. Sources and sinks are identified on the basis of the information provided by *SuSi* [23].

FlowDroid can precisely detect intra-component data flows, but it cannot detect inter-component data flows involving intents.

### Epicc

The Epicc tool precisely and efficiently analyzes the inter-component communication (ICC). It reduces the discovery of ICC to an instance of the IDE (interprocedural distributive environment) data flow problem. IDE is an extension of the IFDS problem that extends the graph reachability problem to a value-computation problem. It identifies properties (such as *action*, *category*, and *data MIME type*) of intents that can be sent and received by components [13]. For example, Epicc might identify that a particular app can send intents only with action `android.intent.action.VIEW` and MIME data type `image/jpg`.

### Soot

Soot [24] is a Java optimization and analysis framework. It provides four intermediate representations for analyzing and transforming Java and Android bytecode. As mentioned previously, static analyses can analyze these intermediate representations more efficiently than analyzing actual source code or bytecode. Soot also enables construction of precise control-flow graphs (CFGs) that provide abstract model of programs.

We use the Soot framework in several parts of our analyzer, described in Chapter 4.

## 2.3 Motivating Example

In section 2.1, we used a simple photo-sharing example to briefly demonstrate why apps need to share data with each other. In this section, we discuss a motivating app set in which apps share data using *intents*. Figure 2.1 shows how the sensitive data can flow from the source to the sink only after traversing through multiple apps.

Listing 2.1 shows the code that is executed when the user clicks on a button in activity `MainActivity` in the SendSMS app. It reads the device ID (source) and stores it in an intent using the `putExtra()` method. Finally, the `startActivityForResult()` method takes that intent as an argument to start a new activity.

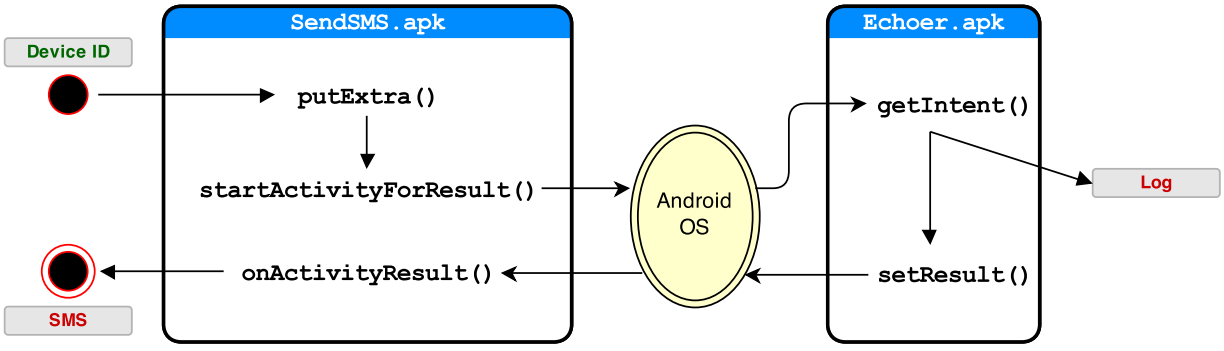


Figure 2.1: Data leak via an intent between `SendSMS.apk` and `Echoer.apk`

```
public class Button1Listener implements OnClickListener {
   private final MainActivity act;
   public Button1Listener(MainActivity parentActivity) {
      this.act = parentActivity;
   }
   public void onClick(View arg0) {
      Intent i = new Intent(Intent.ACTION_SEND);
      i.setType("text/plain");
      TelephonyManager tManager = (TelephonyManager) this.act.getSystemService(
         Context.TELEPHONY_SERVICE);
      String uid = tManager.getDeviceId();      // SOURCE
      i.putExtra("secret", uid);    // write sensitive data to Intent
      this.act.startActivityForResult(i, 0);    // outgoing Intent
   }
}
```

Listing 2.1: `SendSMS.button1listener.java`

Because the target component for the intent is not specified (it's an *implicit* intent), the OS must find an activity that can handle it. With the help of the *intent filters* defined in the manifest files of all installed apps, the OS chooses the app that can handle this intent. Listing 2.2 shows that the Echoer app can handle this intent.

```
...
...
<activity
   android:name="echoer.MainActivity"
   android:label="@string/app_name" >
   <intent-filter>
      <action android:name="android.intent.action.SEND" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="text/plain" />
   </intent-filter>
</activity>
...
...
```

Listing 2.2: `AndroidManifest.xml` in `Echoer.apk`

The Echoer app receives the intent by using the `getIntent()` method, as shown in Listing 2.3. `Intent i` is stored as a class field inside the `MainActivity` class.

```
public class MainActivity extends Activity {
   Intent i;

   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Button button1 = (Button) findViewById(R.id.button1);
      button1.setOnClickListener(new Button1Listener(this));
   }
   protected void onResume()
   {
      super.onResume();
      i = getIntent();  // read data received in Intent from the caller
      Bundle extras = i.getExtras();
      Log.i("Data received in Echoer: ", extras.getString("secret"));   // SINK
   }
   ...
   ...
}
```

Listing 2.3: `Echoer.MainActivity.java`

The `onClick()` callback method shown in the Listing 2.4 is called when the user clicks on the button `button1` and sends the received data (`Intent this.act.i`) back to the caller of this activity (SendSMS) by using `setResult()`. A callback method `onActivityResult()` inside SendSMS is called when it receives the result, as shown in Listing 2.5.

```
public class Button1Listener implements OnClickListener {
   private final MainActivity act;
   public Button1Listener(MainActivity parentActivity) {
      this.act = parentActivity;
   }
   public void onClick(View arg0) {
      this.act.setResult(0, this.act.i);  // send received data back to the caller
      this.act.finish();
   }
}
```

Listing 2.4: `Echoer.button1listener.java`

```
1  public class MainActivity extends Activity {
2     protected void onCreate(Bundle savedInstanceState) {
3        super.onCreate(savedInstanceState);
4        setContentView(R.layout.activity_main);
5        Button button1 = (Button) findViewById(R.id.button1);
6        button1.setOnClickListener(new Button1Listener(this));
7     }
8     ...
9     ...
10    protected void onActivityResult(int requestCode, int resultCode, Intent data) {  //
       incoming Intent Result
11       sendSMSMessage(data.getExtras().getString("secret"));
12    }
13    protected void sendSMSMessage(String message) {
14        SmsManager smsManager = SmsManager.getDefault();
15        smsManager.sendTextMessage("1234567890", null, message, null, null);   // SINK
16    }
17 }
```

Listing 2.5: `SendSMS.MainActivity.java`

Finally, the intent `data` is sent out via an SMS using the `sendTextMessage()` method. This completes the inter-app data flow that originates at line 11 (source) in Listing 2.1 and gets leaked at line 15 (sink) in Listing 2.3 within the SendSMS app but via the Echoer app.

None of the existing tools, including FlowDroid, can detect such inter-component data flows. A more sound and precise inter-component data-flow analysis is required. In the next chapter, we present our analysis design, which aims at tracing such data flows.

# Chapter 3

# Analysis Design

The overview of our analysis method is shown in Figure 3.1. Our goal is to produce a set of all possible source-to-sink flows within a set of Android apps. The taint flow analysis takes place in two phases. In phase 1, each application is analyzed individually. Received intents are considered sources; sent intents are considered sinks. The output of the phase 1 analysis, for each app, consists of (1) flows within each component, found by FlowDroid; (2) identification of the properties of sent intents, as found by Epicc; and (3) intent filters of each component, extracted from the manifest file.

An intent ID is assigned to every source code line that sends an intent (that is, a source code line that consists of a call to a method in the `startActivity` family), as described in Section 4.1.1. Sent intents with distinct IDs are considered distinct sinks, whereas intents with the same ID are combined.

Phase 2 of the analysis can be carried out on a subset of apps, using the output of phase 1. The output of phase 2 consists of all the source-to-sink flows found in the set of apps.

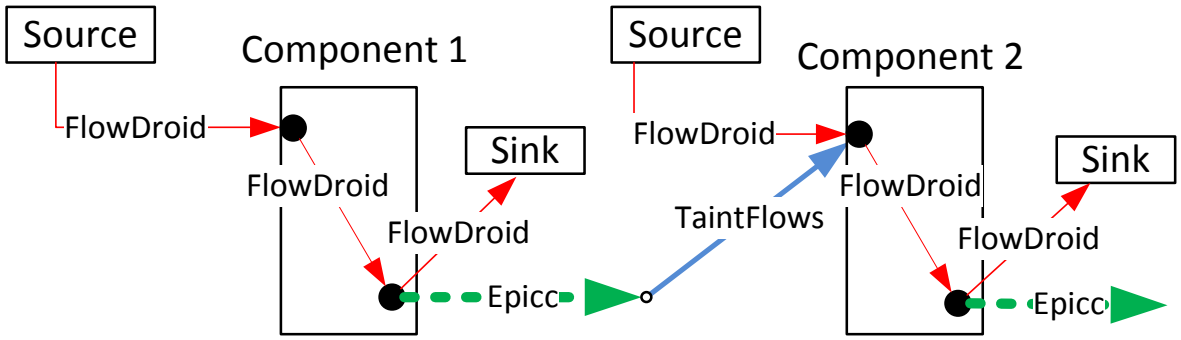


Figure 3.1: Analysis by data flow type: FlowDroid identifies sources (including intents received), flow of the data within the component, and sinks (including intents sent). Epicc identifies characteristics of intents sent by a component. TaintFlows, the analyzer, matches sent intent characteristics to components that could receive the intent, using app manifest data and matching intent IDs from Epicc and FlowDroid. A component could have zero or more sources, sinks, intents received, and intents sent. From beginning to end, a given data flow could be internal to one component or traverse multiple components, which could be in a single app or in multiple apps.

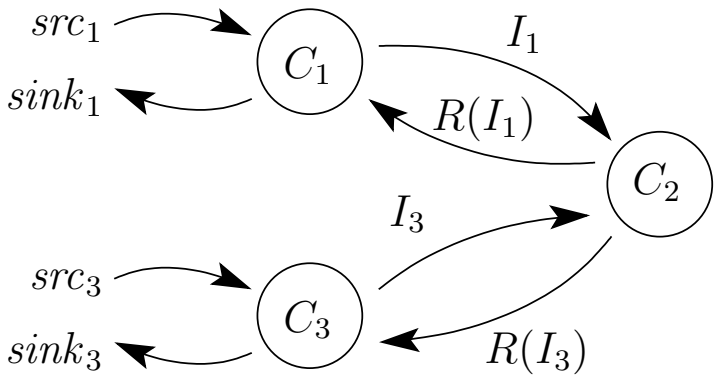


Figure 3.2: Running example described in Section 3.1. $R(I_i)$ denotes the response to intent $I_i$ (set using `setResult()`).

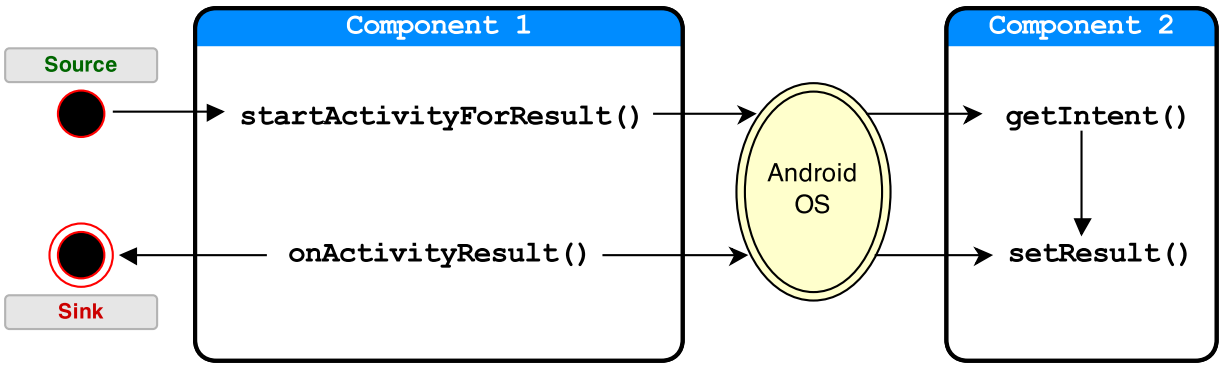


Figure 3.3: Interaction between $C_1$ and $C_2$ in the running example

## 3.1 Example Scenario

This section introduces an example of information flows between multiple components (Figure 3.2) that cannot be precisely analyzed by existing tools. Suppose that component $C_1$ sends data to component $C_2$ and receives data from it in return. Component $C_3$ interacts with $C_2$ in a similar fashion. These three components can belong to different apps or to a single app. As depicted in Figures 3.2 and 3.3, for $i \in \{1, 3\}$:

1. Component $C_i$ calls `startActivityForResult()` to send data from source $src_i$ to component $C_2$ via intent $I_i$.
2. Component $C_2$ reads data from intent $I_i$ and sends that data back to component $C_i$ by calling `setResult()`.
3. Component $C_i$, in method `onActivityResult()`, reads data from the result and writes it to sink $sink_i$.

The analysis should determine that (1) information flows from $src_1$ to $sink_1$ (but not $sink_3$), and (2) information flows from $src_3$ to $sink_3$ (but not $sink_1$). Note that FlowDroid by itself cannot produce a result this precise even if the three components are part of a single app.

## 3.2 Phase 1

In this phase, each app is analyzed individually. An intent is identified by a tuple of (sending component, receiving component, intent ID). An intent sent from $C_1$ to $C_2$ with ID $id$ is denoted by $I(C_1, C_2, id)$.

In phase 1, when a component calls a method in the `startActivity` family, the recipient of the intent is unknown (because each app is analyzed in isolation in phase 1, and the recipient can be a component in another app), so we use null for the recipient field. Likewise, in the `onCreate()` method, we do not know the sender of the intent, so we use null for the sender field. If a component receives an intent $I_1$ and returns information via the `setResult()` method, we denote the returned information by $R(I_1)$.

We write $source \xrightarrow{C} sink$ to denote that information flows from *source* to *sink* in component $C$. For this purpose, we treat intents as both sources (in the component that creates and sends the intent) and sinks (in the component that receives the intent). Using this notation, we represent the phase 1 equations for the flows depicted in Figure 3.2 and described in Section 3.1 as follows:

$$
\begin{aligned}
src_1 &\xrightarrow{C_1} I(C_1, \mathsf{null}, id_1) \\
R(I(C_1, \mathsf{null}, \mathsf{null})) &\xrightarrow{C_1} sink_1 \\
I(\mathsf{null}, C_2, \mathsf{null}) &\xrightarrow{C_2} R(I(\mathsf{null}, C_2, \mathsf{null})) \\
src_3 &\xrightarrow{C_3} I(C_3, \mathsf{null}, id_3) \\
R(I(C_3, \mathsf{null}, \mathsf{null})) &\xrightarrow{C_3} sink_3
\end{aligned}
$$

The flows constitute the desired output of the FlowDroid analysis. Although all the flows in the running example involve intents, in general our analysis will also find flows from non-intent sources to non-intent sinks.

We focus, in both description and implementation, on intents sent and received by `Activity` components; other types of components (services, content providers, broadcast receivers) can be handled similarly.

## 3.3 Phase 2

After all apps in a set have been analyzed, we enter phase 2. Our goal is to discover how tainted information can flow between components. For each sent intent, we find all possible recipients, and we instantiate the phase 1 flow equations (which have missing sender/receiver information) for all possible sender/receiver pairs, as we describe in detail in Section 3.3.1. For the running example, the phase 2 flow equations are as follows:

$$
\begin{aligned}
src_1 &\xrightarrow{C_1} I(C_1, C_2, id_1) \\
R(I(C_1, C_2, id_1)) &\xrightarrow{C_1} sink_1 \\
I(C_1, C_2, id_1) &\xrightarrow{C_2} R(I(C_1, C_2, id_1)) \\
I(C_3, C_2, id_3) &\xrightarrow{C_2} R(I(C_3, C_2, id_3)) \\
src_3 &\xrightarrow{C_3} I(C_3, C_2, id_3) \\
R(I(C_3, C_2, id_3)) &\xrightarrow{C_3} sink_3
\end{aligned}
$$

Let $T(s)$ denote the taint of $s$, that is, the set of sensitive sources from which $s$ potentially has information. The goal of the analysis is to determine the taint of all sinks. Each phase 2 flow equation $s_1 \rightarrow s_2$ relates the taint of $s_1$ to the taint of $s_2$. If data flows from $s_1$ to $s_2$, then $s_2$ must be at least as tainted as $s_1$. Accordingly, we generate a taint equation $T(s_1) \subseteq T(s_2)$. For the running examples, the taint equations we generate are

$$
\begin{aligned}
T(src_1) &\subseteq T(I(C_1, C_2, id_1)) \\
T(R(I(C_1, C_2, id_1))) &\subseteq T(sink_1) \\
T(I(C_1, C_2, id_1)) &\subseteq T(R(I(C_1, C_2, id_1))) \\
T(I(C_3, C_2, id_1)) &\subseteq T(R(I(C_3, C_2, id_3))) \\
T(src_3) &\subseteq T(I(C_3, C_2, id_3)) \\
T(R(I(C_3, C_2, id_3))) &\subseteq T(sink_3)
\end{aligned}
$$

Each non-intent source $s$ is tainted with itself; i.e., $T(s) = \{s\}$. We then find the least

fixed-point of the set of taint equations. The end result of phase 2 is the set of possible source-to-sink flows.

### 3.3.1 Details of Generating Phase 2 Flow Equations

Let $S$ be the set of sources and sinks (including intents and intent results) in the phase 1 flow equations. Consider a transmitted intent $I_{TX}$ and a received intent $I_{RX}$ from phase 1. In all cases, $I_{TX}$ will have the form $I(C_{TX}, \mathsf{null}, id)$, and $I_{RX}$ will have the form $I(\mathsf{null}, C_{RX}, \mathsf{null})$.

In Section 3.3, we said that we instantiate the phase 1 flow equations for all possible intent sender/receiver pairs. We now give the details of how we do this. For each phase 1 flow $src \to sink$, we generate the set of all flows of the form $src' \to sink'$ that satisfy the following conditions:

1. If $src$ is a regular (non-intent) source, then $src' = src$.
2. If $sink$ is a regular (non-intent) sink, then $sink' = sink$.
3. If $src$ has the form $I(\mathsf{null}, C_{RX}, \mathsf{null})$ (for example, the result of a call to `android.app.Activity.getIntent()`), then $src'$ must have the form $I(C_{TX}, C_{RX}, id)$ where there exists an intent $I(C_{TX}, \mathsf{null}, id) \in S$ that matches the intent filter of component $C_{RX}$.
4. If $sink$ has the form $I(C_{TX}, \mathsf{null}, id)$ (for example, an intent object passed to `startActivity()`), then $sink'$ must have the form $I(C_{TX}, C_{RX}, id)$ where component $C_{RX}$ has an intent filter that matches the intent $sink$.
5. If $src$ has the form $R(I(C_{TX}, \mathsf{null}, \mathsf{null}))$ (for example, a parameter of the callback method `onActivityResult()`), then $src'$ must have the form $R(I(C_{TX}, C_{RX}, id))$ where
   (a) there exists an intent $I(C_{TX}, \mathsf{null}, id) \in S$ that matches the intent filter of component $C_{RX}$, and
   (b) $R(I(\mathsf{null}, C_{RX}, \mathsf{null})) \in S$.

6. If *sink* has the form $R(I(\mathsf{null}, C_{RX}, \mathsf{null}))$ (for example, a value passed to `setResult()`), then *sink*$'$ must have the form $R(I(C_{TX}, C_{RX}, id))$ where
   (a) there exists an intent $I(C_{TX}, \mathsf{null}, id) \in S$ that matches the intent filter of $C_{RX}$, and
   (b) $R(I(C_{TX}, \mathsf{null}, \mathsf{null})) \in S$.
7. If *src* has the form $I(\mathsf{null}, C_{RX}, \mathsf{null})$ and *sink* has the form $R(I(\mathsf{null}, C_{RX}, \mathsf{null}))$, then *sink*$'$ must be $R(src')$.

Condition 7 allows us to precisely handle a situation in which a component (such as $C_2$ in the running example) processes data from various callers without intermingling the taintedness of the data. Condition 7 is sound as long as multiple instances of the component can communicate only via flows included in the phase 1 equations. Our current implementation catches most such flows but misses inter-instance communication via static fields.

For example, in Figure 3.2, if all components are part of the same app, then the two launched instances of $C_2$ can store information from $I_1$ and $I_3$ in a static field (which is shared between the two instances of $C_2$). The value in the static field (tainted with both $src_1$ and $src_3$) can then be read and copied into $R(I_1)$ and $R(I_3)$. This flow would be missed by our current analysis.

Although not yet addressed, static fields can be managed in a sound manner. In particular, if an app $A$ has a class $C$ with a static field $sf$, we could modify FlowDroid to introduce a dummy entity $sf_{A,C}$ that can act both as a source and as a sink. Reading from static field $sf$ would be treated as reading from $sf_{A,C}$, and writing to $sf$ would be treated as writing to $sf_{A,C}$. The resulting phase 1 flow equations would enable our phase 2 analysis to soundly handle inter-instance communication via static fields.

Our analysis cannot precisely handle the situation in Figure 3.4, wherein tainted data travels through a chain of apps. In this situation, our analysis would mark all intent results as being tainted with data from both $I_1$ and $I_3$ instead of being able to keep them separate.

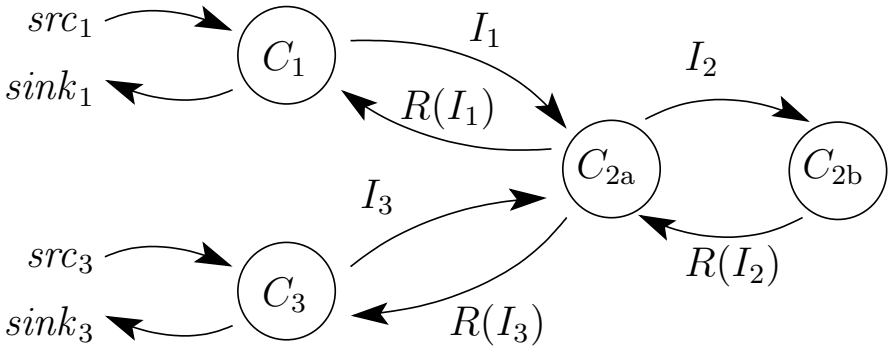


Figure 3.4: Example of inter-app communication flow wherein, for $i \in \{1, 3\}$: $C_{2a}$ receives tainted data from $C_i$, sends it to $C_{2b}$, receives a result with the same taint, and finally sends it back to $C_i$.

### 3.3.2 Rules for Matching Intents

In Section 3.3.1, we used the term *match* in relation to a sent intent and an intent filter. We now more fully define what we mean by match. The Android documentation[1] describes how a sent intent is matched to potential recipients. First, if the intent explicitly designates a recipient, then the intent is matched with that recipient. Otherwise, the intent is matched with a filter if it passes three tests: an action test, a category test, and a data test.

Epicc provides information about outgoing intents in its app analysis, and we use that. It provides no information about the URI fields of intents, so we ignore the URI fields when matching intents with intent filters. Sometimes, Epicc will return `<any_string>` for the action string or `Found top element` for the intent as a whole. For this case, the analyzer has two modes (which can be selected by a command-line option): (1) a sound mode, which assumes that an unknown action string potentially matches any action string in any filter, thereby typically generating many false positives, and (2) a precise mode, which assumes that the unknown action string does not match any filter, thereby potentially generating false negatives. Likewise, in the sound mode, a top-element intent matches every filter, and in the precise mode, it matches nothing.

[1] `http://developer.android.com/guide/components/intents-filters.html#Resolution`

# Chapter 4

# Implementation

We have implemented our approach in the `DidFail`[1] analyzer. Our analyzer (source code and binaries), along with three apps that demonstrate the running example in §3.1, are available at

`http://www.cert.org/secure-coding/tools/didfail.cfm`

## 4.1 Phase 1 Analysis

Figure 4.1 shows the components of our analyzer, the processing sequences, and dataflow paths. The analyzer incorporates use of the previously existing and unchanged tools Epicc, Dare, and Soot; a modified version of FlowDroid; and new tools TransformAPK and TaintFlows. TaintFlows performs the phase 2 analysis.

### 4.1.1 APK Transformer

The APK Transformer must be used in the first step of the analysis to be able to integrate results of the different analytical tools used afterwards. This step is critical to achieve the ultimate goal of outputting detected paths from sources to sinks, including paths that

[1]Droid Intent Data Flow Analysis for Information Leakage

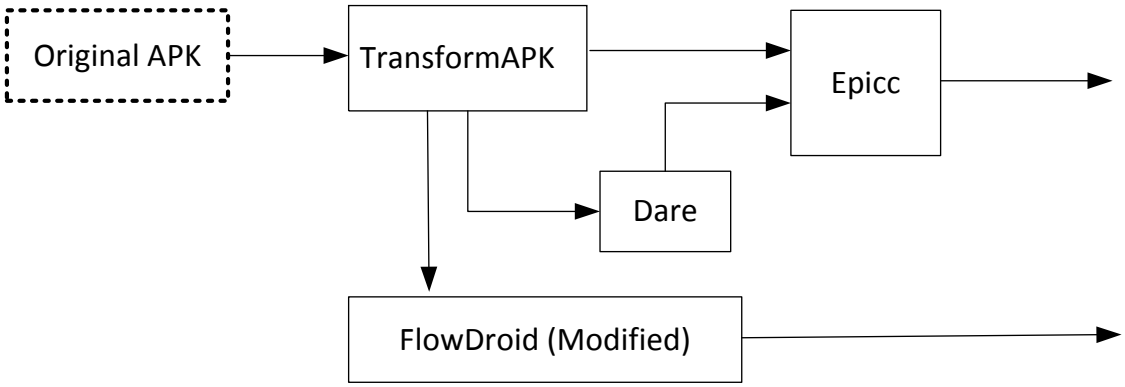


Figure 4.1: Phase 1

contain data flows via intents. Android apps are packaged in files with the extension *.apk*. In Figure 4.1, "Original APK" is the original Android app. With the APK Transformer, our analyzer modifies that app to enable matching intents mentioned in both the Epicc and FlowDroid outputs. To do this, we transformed each original .apk file into a modified .apk file, using Soot. We developed a program that first uses Soot to transform the .dex Android bytecode into an intermediate representation called *jimple*. The program uses the Soot framework to locate method calls that send intents, and immediately before that, we insert new jimple code, which calls an Android method that inserts a unique ID into the intent. Our program then uses Soot to compile the modified jimple code into a new .apk file. When Epicc processes this modified file, it prints the unique intent IDs. As described in Section 4.1.2, we modified the source code of FlowDroid so that its output identifies sent intents by their intent ID, enabling us to match intent analysis from the two tools. We could not modify the source code of Epicc, because the source code is not available yet. According to the Epicc website, the authors plan to publish the source code in the future. Once the source code is available, we might be able to combine FlowDroid and Epicc in a more efficient manner.

### 4.1.2 FlowDroid (Modified)

We modified FlowDroid in several ways. For our phase 1 analysis, we consider the points where intent information flows in and out of the component as sources and sinks respectively. Therefore, we added the method `onActivityResult()` as a source and `setResult()` as a sink in FlowDroid. The methods `getIntent()` and `startActivityForResult()` were already present as a source and sink respectively. Although the FlowDroid tool comes with a smaller `SourcesAndSinks.txt` file, the much larger `SourcesAndSinks.txt`[2] file can be substituted from the SuSi analyzer [25]. We also added code to search for the `putExtra()` call we added to insert the unique intent ID, which was added by the APK Transformer. In flows where an intent is the sink, the output of FlowDroid identifies the intent by its unique ID.

In a flow $src \xrightarrow{C} I(C, \mathsf{null}, id)$, how do we identify $C$? When an intent is sent via `base.startActivity()` (including the case where `base` is an implicit `this`), we assume that the class of `base` must be the sending component.[3]

The output of FlowDroid was originally nondeterministic in the order in which flows were listed. To produce deterministic output for regression testing, we simply sorted the flows before printing.

### 4.1.3 Epicc and Dare

The Dare [26] tool takes the transformed .apk file as input, retargets the application, and outputs Java class files. The Epicc analysis takes two inputs: the transformed .apk file and the output of Dare.

[2] `https://github.com/secure-software-engineering/SuSi` (2013-11-25)

[3] We have not yet been able to confirm or refute whether this assumption is sound. To preserve soundness at the expense of precision, we considered an intent as potentially matching the intent filter of all components of an app if it matches the intent filter of any component of the app.

## 4.2 Phase 2 Analysis

Each app in the app set undergoes its own separate phase 1 analysis, with each phase 1 analysis outputting three separate output files (manifest file, Epicc output, and FlowDroid output) that are input to the phase 2 analysis. If there are $n$ apps in the app set, then the phase 1 analysis is performed $n$ times, outputting $3n$ files, all of which are input to the (single) phase 2 analysis. The phase 2 analysis output provides information about data flows from a source to a sink, including intents if they are part of the data flow. Figure 4.2 shows the relationship of the analyses in phase 1 and 2.

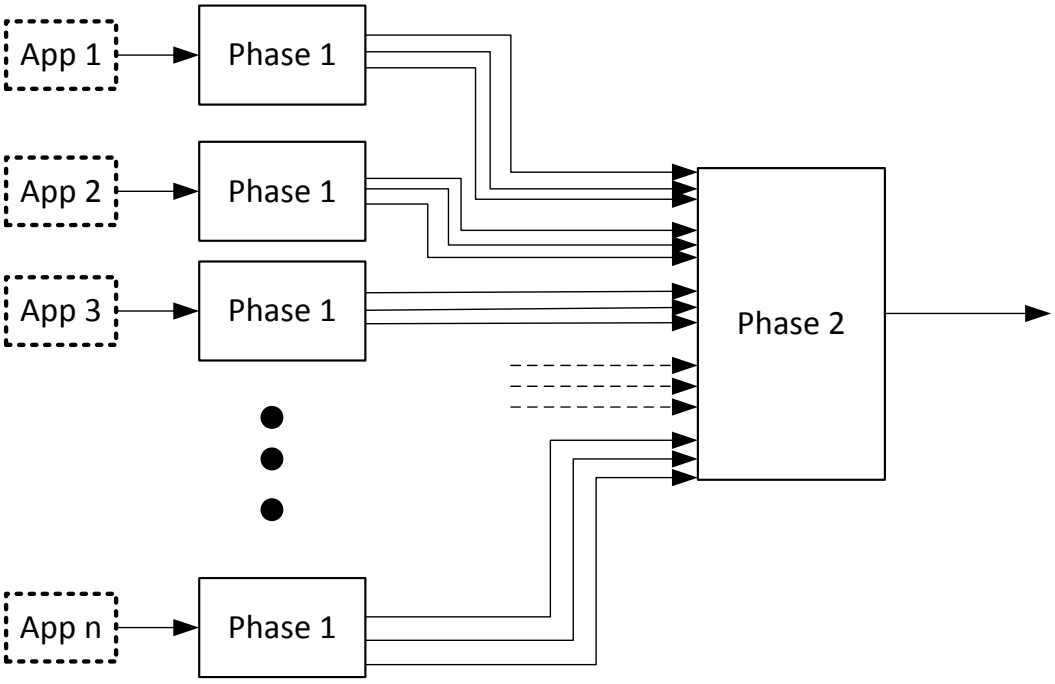


Figure 4.2: Two-phase analysis

# Chapter 5

# Experimental Results

We tested our prototype analyzer on two app sets. App set 1 consists of three apps that we created, which match the running example in Figure 3.2. App set 2 consists of three apps from the DroidBench benchmark suite [27] that use intents for inter-app communication. Our analyzer successfully traced all inter-app and intra-app flows in both app sets. As described in the previous section, we first ran phase 1 analysis on all apps individually, and then for phase 2, we ran TaintFlow analysis for each set of apps.

## 5.1 App Set 1: Colluding Apps

- `SendSMS.apk`: This app leaks the user's device ID through an SMS. It reads the user's device ID, then adds it to an intent using the `putExtra()` method. It then sends this intent out by calling `startActivityForResult()`. Another app receives this intent and responds with a result. When the intent result is received, the `onActivityResult()` callback method is called. Data received in the result is then leaked through an SMS.
- `Echoer.apk`: This app receives intents from other apps. It reads the incoming intents using the `getIntent()` method and writes the received data using `Log`. Also, it sends this data back to the transmitter using the `setResult()` call.

- `WriteFile.apk`: This app is similar to SendSMS except that it reads the user's location and leaks it to the file system.

**Result:**

Some of the data flows detected by our analysis are as follows:

- $getDeviceId() \xrightarrow{SendSMS} putExtra() \xrightarrow{SendSMS} startActivityForResult() \xrightarrow{Int2}$
  $getIntent() \xrightarrow{Echoer} setResult() \xrightarrow{Res4}$
  $onActivityResult() \xrightarrow{SendSMS} getExtras() \xrightarrow{SendSMS} sendTextMessage()$
- $getLastKnownLocation() \xrightarrow{WriteFile} putExtra() \xrightarrow{WriteFile}$
  $startActivityForResult() \xrightarrow{Int1}$
  $getIntent() \xrightarrow{Echoer} setResult() \xrightarrow{Res3}$
  $onActivityResult() \xrightarrow{WriteFile} getExtra() \xrightarrow{WriteFile} write()$

**Note:** Arrows with name above it show intra-component flows within that app or component. Arrows with Int* or Res* above them show inter-component flows via intents and intent results respectively. Figure 5.1 shows all intent-based data flows detected by our analyzer.

```
Int1: Intent(tx='WriteFile', rx='Echoer', intent_id='newField_8')
Int2: Intent(tx='SendSMS', rx='Echoer', intent_id='newField_6')
Res3: IntentResult(i=Intent(tx='WriteFile', rx='Echoer', intent_id='newField_8'))
Res4: IntentResult(i=Intent(tx='SendSMS', rx='Echoer', intent_id='newField_6'))
Snk5: 'Sink: <android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.
    lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent)>'
Snk6: 'Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>'
Snk7: 'Sink: <java.io.FileOutputStream: void write(byte[])>'
Src8: 'Src: <android.location.Location: double getLatitude()>'
Src9: 'Src: <android.location.Location: double getLongitude()>'
Src10: 'Src: <android.location.LocationManager: android.location.Location
    getLastKnownLocation(java.lang.String)>'
Src11: 'Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>'
```

Listing 5.1: Legend for reading the graph shown in Figure 5.1

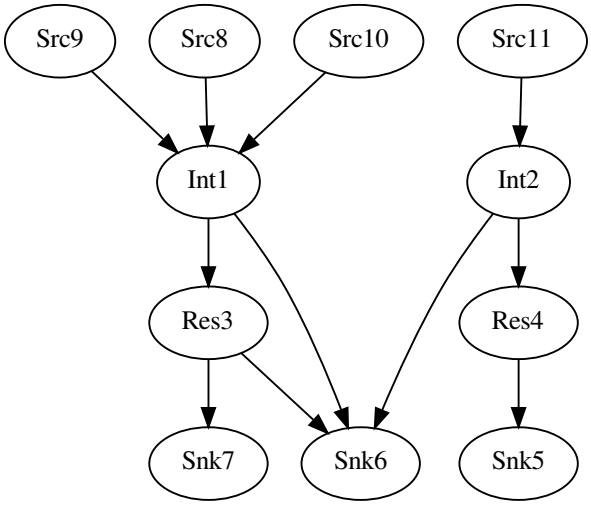


Figure 5.1: Tainted intent-based data flow in test apps

## 5.2 App Set 2: DroidBench Benchmark Suite

DroidBench[1] is a set of open-source Android applications that can be used as a testing ground for static analysis tools.

- `IntentSource1.apk`: This app reads the incoming intent using `getIntent()` and sends the intent out by calling `startActivityForResult()`. When another app receives this intent and responds with a result, this app logs the result (sink).
- `InterAppCommunication_IntentSink1.apk`: This app reads the user's device ID (source), adds it to the received intent, and then sends the intent result out by calling `setResult()` method.
- `InterAppCommunication_IntentSink2.apk`: This app also reads the user's device ID (source), adds it to a new `Intent` object, and sends the intent out by calling `startActivity()`.

[1] `https://github.com/secure-software-engineering/DroidBench` (2014-03-26)

**Result:**

Some of the data flows detected by our analysis are as follows:

- $getDeviceId() \xrightarrow{IntentSink2} putExtra() \xrightarrow{IntentSink2} startActivity() \xrightarrow{Int3}$
  $getIntent() \xrightarrow{IntentSource1} startActivityForResult() \xrightarrow{Int4}$
  $getIntent() \xrightarrow{IntentSink1} putExtra() \xrightarrow{IntentSink1} setResult() \xrightarrow{Res8}$
  $onActivityResult() \xrightarrow{IntentSource1} Log.i()$
- $getDeviceId() \xrightarrow{IntentSink1} putExtra() \xrightarrow{IntentSink1} setResult() \xrightarrow{Res8}$
  $onActivityResult() \xrightarrow{IntentSource1} Log.i()$

**Note:** Arrows with name above it show intra-component flows within that app or component. Arrows with Int* or Res* above them show inter-component flows via intents and intent results respectively. Figure 5.2 shows all intent-based data flows detected by our analyzer.

```
Int3: Intent(tx='IntentSink2', rx='IntentSource1', intent_id='newField_5')
Int4: Intent(tx='IntentSource1', rx='IntentSink1', intent_id='newField_6')
Int6: Intent(tx='IntentSource1', rx='IntentSource1', intent_id='newField_6')
Res8: IntentResult(i=Intent(tx='IntentSource1', rx='IntentSink1', intent_id='newField_6
    '))
Snk11: 'Sink: <android.content.Intent: android.content.Intent setAction(java.lang.
    String)>'
Snk12: 'Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>'
Src13: 'Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>'
}
```

Listing 5.2: Legend for reading the graph shown in Figure 5.2

**Note:** In DroidBench suite, `InterAppCommunication_IntentSink1.apk` and `InterAppCommunication_IntentSink2.apk` use the same package, `de.ecspride`. Because Android does not allow two packages with the same name, we modified the package name for the first and the second app to `de.ecspride.IntentSink1` and `de.ecspride.IntentSink2` respectively.

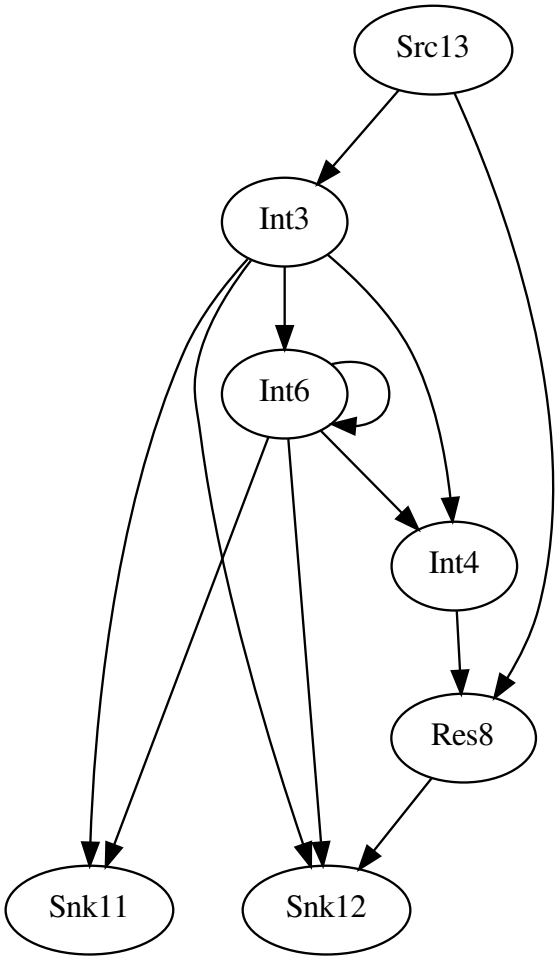


Figure 5.2: Tainted intent-based data flow in DroidBench apps

# Chapter 6

# Limitations

Soundness and precision are important characteristics of any program analysis. DidFail inherits sources of unsoundness and imprecision from its building blocks, FlowDroid and Epicc.

## 6.1 Sources of Unsoundness

Sources of unsoundness cause the analysis to fail to identify a tainted flow. Sources of unsoundness in our analysis include reflection and native code, which are not addressed by Epicc. FlowDroid also does not consider reflective calls. However, FlowDroid does analyze calls that invoke native code, using a heuristic called *taint wrapping*. It defines explicit taint propagation rules for commonly called native methods. For all other native methods, FlowDroid uses the following heuristic: if the input array was tainted before the call, then FlowDroid determines that all call arguments and any return value are tainted. FlowDroid's handling of native calls is unsound; it does not analyze the native code in the callee. For example, native code can read from sources and write to sinks, which will not be detected by FlowDroid. FlowDroid also is unsound because it does not trace some leaks caused by multithreading and some implicit flows.

DidFail does not consider implicit flows where information is not read from the received

intents. For example, suppose an app $A_T$ wants to communicate a bit vector $\langle b_n, ..., b_0 \rangle$ to an app $A_R$ without being detected by our analysis. App $A_R$ can have two components, $C_{R0}$ and $C_{R1}$, which have mutually exclusive intent filters. Then $A_T$ can send a sequence of intents $\langle I_n, ..., I_0 \rangle$ where

- intent $I_i$ matches $C_{R0}$ iff $b_i = 0$, and
- intent $I_i$ matches $C_{R1}$ iff $b_i = 1$.

To ensure that intents arrive in proper order, App $A_T$ can use `startActivityForResult()` to send the intent and then wait until $C_{R0}$ or $C_{R1}$ calls `setResult()` to acknowledge receipt.

Data can flow between components of different apps via file or database accesses such as writes to and reads from shared external storage, internal storage, and shared public directories on the device. These same files and databases can be accessed to allow (and sometimes to restrict) data flow between components of a single app. FlowDroid considers a read from a file to be a source and a write to a file to be a sink. Within one component, the FlowDroid analysis finds a tainted flow if there is a read from a file and a call to a sink or a call to a source and a write to a file. Although our analyzer finds some tainted flows, including file access, which FlowDroid does not, it does not soundly analyze taint flows involving files accesses. Our analysis finds a multicomponent tainted data flow that ends with a write to a file sink. However, it does not trace a multicomponent tainted data flow that starts with a read from a file source. Also, our analyzer is unsound because it does not trace a multi-component tainted data flow with a read from a file in one component after a write to it by another.

Additional sources of unsoundness in the analyzer include shared static fields. FlowDroid traces tainted data within a component (or within an entire app, depending on command-line arguments) that is written to and/or read from a shared static field. Unsoundness resulting from inter-instance static field communication is discussed in Section 3.3.1.

## 6.2 Sources of Imprecision

Imprecision in the analysis would result in the analysis reporting a possible tainted flow where such a flow is not possible in the real system. For instance, the Epicc analyzer over-approximates inter-component communication via intents because it does not handle URIs, which are used by Android to match intents to receiving components. As previously described, FlowDroid's analysis of native calls is not precise and sometimes overapproximates returned tainted fields. DidFail does not use permissions to restrict possible matching of intent senders and receivers. This overapproximated matching is a source of analysis imprecision.

In our phase 1 FlowDroid analysis, all the received intents for a component are conflated together as a single source. As future work, to be more precise, we plan to modify FlowDroid so that when a callback function such as `onCreate()` is analyzed, it can report the data flows as a function of the properties of the received intent. For example, we might report that a component $C$ has a flow $camera \xrightarrow{C} R(I)$ iff $I.hasExtra(\text{“}\texttt{cam}\text{”}) = \texttt{true}$. Similarly, we can make analysis of `onActivityResult()` be sensitive to the value of the `requestCode` parameter.

# Chapter 7

# Related Work

The Epicc tool performs the most precise static analysis of Android intents and inter-component intent communication of any Android analyzer known to us, finding vulnerabilities with far fewer false positives than the next best tools. The authors showed that the intent ICC problem can be reduced to an IDE problem, so the existing algorithms for efficient IDE solutions could be used. Epicc builds on a preexisting IDE framework within the Soot library.

Daniel Hausknecht's 2013 thesis [28] describes VarDroid, intended to integrate intra-component and inter-component static analyses, which is similar in some ways to our method. His concept is modular whereby different analyses can be switched out for the intra-component and inter-component data-flow tracing. Where we use a modified FlowDroid, his concept could use Chex [19], FlowDroid, or another analyzer. His thesis says he did not complete integrating FlowDroid in his system. Instead, he simulated data-flow analysis through probabilistically generated information simulating results of the intra-component and inter-component analyses.

The Kirin tool [29] provides a formalized model for stating data policy and compares stated policies to information extracted from app manifest files, processing this information on the phone, to determine whether an app should be installed. The SORBET [30] system modified a standard Android system to enable formal definition of desired secu-

rity properties, which were proven to hold on SORBET but not on Android. Livshitz et al. [31] performed static analyses on Java code to detect policy violations with security implications, including taint analysis. TaintDroid [2] performs real-time taint tracking to dynamically detect data leaks.

Felt et al. [32] found that about one-third of 940 Android apps tested asked for more privileges than they actually use. They found evidence that a cause of overprivilege is developer confusion resulting in part from inadequate Android API documentation. Furthermore, malicious apps can use permission re-delegation attack methods [33], which when successful take advantage of a higher-privilege app performing a privileged task for an application without permissions. The ComDroid [17] tool analyzes inter-app communication in Android, looking at intents sent and the manifest files for potential vulnerabilities resulting from insecure intent communication. Although it examines vulnerabilities, the ComDroid analysis does not trace and identify data paths between sources and sinks.

# Chapter 8

# Conclusion and Future Work

This thesis introduced a new analysis that integrates and enhances existing Android app static analyses in a two-phase method. We demonstrated feasibility by implementing our approach and testing apps with it. Future work includes enhancing the inter-component part of the taint flow analysis to include additional data channels such as static fields, SQLite databases, and SharedPreferences. We plan to test a large number of publicly available Android apps. We envision that a two-phase analysis such as ours can be used as follows. An app store can run the phase 1 analysis on each of the apps in the app store. When a user wants to install a new app, the app store would conduct the phase 2 analysis and tell the user about the new flows that would be made possible if the new app is installed.

# Bibliography


[1] Gartner, Inc. Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013. http://www.gartner.com/newsroom/id/2665715. 1.1

[2] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. OSDI*, 2010. 1.1, 7

[3] Christian Zibreg. Geek.com: Android's Openness Challenged by a Biased Market Survey? http://www.geek.com/android/androids-openness-challenged-by-a-biased-market-survey-1266598/, 2010. 1.1

[4] WIRED. Skype's Android App Could Expose Your Personal Details. http://www.wired.com/gadgetlab/2011/04/skype-android-security-exploit/, April 2011. 1.1

[5] Dan Goodin. Malware Designed to Take Over Cameras and Record Audio Enters Google Play. http://arstechnica.com/security/2014/03/malware-designed-to-take-over-cameras-and-record-audio-enters-google-play/, March 2014. 1.1

[6] Patrick McDaniel and William Enck. Not So Great Expectations: Why Application Markets Haven't Failed Security. *Security & Privacy, IEEE*, 8(5):76–78, 2010. 1.1

[7] Google, Inc. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, 2014. 1.1

[8] Timothy Vidas and Nicolas Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proc. CCS*, 2014. 1.1

[9] Jon Oberheide. Duo Security: Dissecting Android's Bouncer. https://www.duosecurity.com/blog/dissecting-androids-bouncer, 2014. 1.1

[10] CERT Coding Standards: Android-Only Secure Coding Rules and Guidelines. https://www.securecoding.cert.org/confluence/display/java/Android. 1.1

[11] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proc. PLDI*, 2014. To Appear. 1.1, 1.2, 2.2.1

[12] Christian Fritz. FlowDroid: A Precise and Scalable Data Flow Analysis for Android. Master's thesis, TU Darmstadt, July 2013. 1.1

[13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proc. USENIX Security*, 2013. 1.1, 1.2, 2.2.1

[14] Tereza Pultarova. Colluding Apps a Growing Smartphone Threat. http://eandt.theiet.org/news/2014/feb/colluding-apps.cfm, 2014. 1.1

[15] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network; Distributed System Security Symposium (NDSS)*, volume 17, pages 18–25, 2012. 1.1

[16] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Information Security*, pages 346–360. Springer, 2011. 1.1

[17] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing

Inter-Application Communication in Android. In *Proc. MobiSys*, 2011. 2.1.2, 7

[18] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011. 2.1.2

[19] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. CCS*, 2012. 2.1.2, 7

[20] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977. 2.2

[21] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural DataFlow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995. 2.2.1

[22] Eric Bodden. Heros: Multi-threaded, Language-Independent IFDS/IDE Solver. http://sable.github.io/heros/. 2.2.1

[23] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks. Technical Report TUD-CS-2013-0114, EC SPRIDE, May 2013. 2.2.1

[24] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot – A Java Bytecode Optimization Framework. In *Proc. CASCON*, 1999. 2.2.1

[25] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proc. NDSS*, 2014. 4.1.2

[26] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Proc. FSE*, 2012. 4.1.3

[27] ECSPRIDE. DroidBench – Benchmarks. http://sseblog.ec-spride.de/tools/droidbench/,

March 2014. 5

[28] Daniel Hausknecht. Variability-Aware Data-flow Analysis for Smartphone Applications. Master's thesis, TU Darmstadt, September 2013. 7

[29] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, 2009. 7

[30] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and Enhancing Android's Permission System. In *Proc. ESORICS*. Springer, 2012. 7

[31] V Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proc. USENIX Security*, 2005. 7

[32] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proc. CCS*, 2011. 7

[33] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proc. USENIX Security*, 2011. 7