# FlowDroid: A Precise and Scalable Data Flow Analysis for Android

**FlowDroid: Eine präzise und skalierbare Datenfluss-Analyse für Android**

Master-Thesis von Christian Fritz

Juli 2013

TECHNISCHE UNIVERSITÄT DARMSTADT

SECURE SOFTWARE ENGINEERING GROUP

CASED

EC SPRIDE

FlowDroid: A Precise and Scalable Data Flow Analysis for Android
FlowDroid: Eine präzise und skalierbare Datenfluss-Analyse für Android

Vorgelegte Master-Thesis von Christian Fritz

Prüfer: Prof. Dr. Eric Bodden
Betreuer: Prof. Dr. Eric Bodden

Tag der Einreichung:

**Erklärung zur Master-Thesis**

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 03. Juli 2013

(Christian Fritz)

## Abstract

Computers and information technology are integrated more and more in all areas of our daily life and therefore, a huge amount of data, often confidential, is processed by software. As software has often access to publicly readable storage, is connected to the internet or communicates with other software components, information can be disclosed to unauthorized parties. Especially modern smart phones store a lot of sensitive data and in general, users install small applications which add new functionality to the phone, partly by using this data. These so-called apps are not exclusively provided by the software vendor of the smart phone's operating system or a trusted authority, but by many more-or-less anonymous developers who offer them in app stores with different degrees of moderation. Malicious apps can 'steal' sensitive data, such as the user's contacts or his location. Because access to these information should be generally permitted to generate a value for the user, there is the need to analyze the flow of sensitive data.

In this master thesis we present FlowDroid, a static taint analysis on Java programs. While the basic functionality works on all sorts of Java applications including web services, we customized the analysis to cope specifically with Android apps: It contains a precise model of Android's lifecycle and callbacks and is able to parse and analyze Android apk files. FlowDroid tracks data flows from a set of data sources to a set of data sinks. Both sets can be provided by the user, thus, the analysis can adapt to new Android versions with changed APIs. There are also options to provide default taint propagation rules which improve performance for large analysis targets. Because it is context-, flow-, field- and object-sensitive our approach is very precise.

We created the Android-specific benchmark suite DroidBench which covers typical challenges for static analyses and special test cases for analyses on Android. Version 1.0 contains 39 test apps. We evaluated precision and recall on DroidBench. On this test set FlowDroid outperforms two commercial tools. To validate our results we performed additional tests on vulnerable Android tests apps and on SecuriBench Micro, a test suite for web applications.

## Zusammenfassung

Computer und IT-Systeme im Allgemeinen sind heutzutage in alle Bereiche des täglichen Lebens integriert. Daher werden große Mengen an Daten, die oft vertraulich sind, von Software bearbeitet. Diese interagiert meistens mit der Außenwelt, beispielweise hat sie Zugriff auf das Internet, andere Software oder ein Speichermedium, welches auch von anderer Software verwendet werden kann. Besonders Smartphones verwalten viele sensitive Daten, auf welche von kleinen Programmen, sogenannten Apps, zugegriffen werden kann. Oft geschieht dies rechtmäßig, um eine bestimmte Aufgabe zu erfüllen. Die Apps werden aber nicht ausschließlich vom Hersteller des Smartphone-Betriebssystems oder einer vertrauenswürdigen Entität bereitgestellt, daher sind die Absichten des Autors der App unklar. Bösartige Apps können sensitive Daten stehlen und weiterleiten oder den Aufenthaltsort des Nutzers nachverfolgen. Daher ist es notwendig, den Fluss von solchen sensitiven Daten nachzuvollziehen.

In dieser Masterarbeit stellen wir mit FlowDroid eine statische Datenfluss-Analyse für Java-Programme vor. Sie kann für alle Java-basierten Anwendungen, zum Beispiel Web-Anwendungen, verwendet werden, jedoch liegt der Hauptfokus auf Android. Die Analyse nutzt ein präzises Modell des Lebenszyklus der Android-Komponenten und ist in der Lage, Androids apk-Dateien auszulesen. Im Allgemeinen verfolgt sie Datenflüsse von definierten Datenquellen zu Datensenken, die vom Benutzer bestimmt werden können. Daher kann die Analyse auch mit minimalem Aufwand für neue Android-Versionen mit geänderten Programm-Schnittstellen verwendet werden. Des Weiteren können zur Performance-Optimierung Regeln für die Weitergabe des Datenflusses angegeben werden. Unsere Analyse ist kontext-, fluss-, feld- und objekt-sensitiv und daher sehr präzise.

Die Genauigkeit haben wir anhand von DroidBench, einer eigens dafür erstellten Sammlung von Testfällen, die typische Herausforderungen für statische Analysen und Tests für Android-spezifische Probleme enthält, unter Beweis gestellt. Auf diesen Testfällen konnte FlowDroid ein besseres Ergebnis als zwei kommerzielle Werkzeuge erzielen. Um unsere Ergebnisse mit Hilfe von unabhängigen Tests zu validieren, haben wir die Analyse auf weiteren Android-Apps und auf SecuriBench Micro, einer Sammlung von Tests für Web-Anwendungen, evaluiert.

## Contents

## List of Figures

## List of Tables

## Listings

# 1 Introduction

This chapter motivates the need for tracking of data flows in general and taint analysis on Android or web applications in particular (Section 1.1). It presents the contributions that were made with this master thesis (Section 1.2). In Section 1.3 we explain some terms which are used all over the thesis to facilitate the correct understanding. At the end we describe the structure of the thesis (Section 1.4).

## 1.1 Motivation

Software is used to accomplish everyday tasks and has often access to publicly readable storage, is connected to the internet or communicates with other software components. Thus, confidential information can be disclosed - unintentionally or on purpose - to unauthorized parties. Especially modern smart phones became an important companion for many people and therefore, they store a lot of sensitive data, for instance contact information of business partners, calendar entries and notes. Additionally smart phones have sensors which can record private data such as the GPS location or audio. To extend the functionality of their devices, users can install small applications which can access this data or even add new sensitive data to the phone (e.g. mobile banking applications). These so-called apps are not exclusively provided by the software vendor of the smart phone's operating system or a trusted authority, but by many more-or-less anonymous developers who offer their works in app stores with different degrees of moderation.

In this master thesis we concentrate on Android Apps and Android market places. Beside the official Google Play Store[1] which requires a registration and performs a quick analysis of each submitted app - a creditable, but not satisfactory practice - there are many unofficial stores which have no clear moderation policy or are not moderated at all. This allows to submit malicious apps which can 'steal' sensitive data, such as the user's contacts or her location. Because access to this information is generally permitted to generate a value for the user, there is the need to analyze the flow of this sensitive data to decide if a flow poses an undesired information disclosure.

The example in Listing 1.1 motivates the need for a precise and specifically adapted taint analysis for Android. The code represents the behavior for one screen of an Android application. A password from a text field is read (line 5) whenever the application is restarted. When the user clicks on a button of the activity, it is sent to some constant telephone number via SMS (line 22). This constitutes a data flow from the password field (the source) to the SMS API (the sink). Though this is a small example, similar code can be found in real-world malware apps [59].
In this example, *sendMessage()* is associated with a button in the app's UI. It is a callback method that gets triggered by an *onClick* event. In Android, listeners are defined either directly in the code or in the layout XML file, as is assumed here. Thus, analyzing the source code alone is insufficient—one must also process the meta data files to correctly associate all callback methods.

In this code a leak only occurs if *onRestart()* is called, initializing the *user* variable, before *sendMessage()* executes. To be both sound and precise, a taint analysis must model the app lifecycle correctly, recognizing that a user may indeed hit the button after the app restarts.

A precise analysis is needed to differentiate fields of an object: The *user* object containing two fields, a string for the user name and another one for the password, but only one of them should be considered sensitive. Although there are precise analyses for many program languages including Java, the above example shows that Android requires a specially adapted analysis. This was acknowledged by the research community and there are already some approaches which are capable of analyzing Android apps, however, they are lacking either of necessary precision or are do not consider all relevant Android-specific features (see Section 6.2).

## 1.2 Contributions

We created FlowDroid, a context-, flow-, object- and field-sensitive taint analysis which is focused on Android applications, but can be applied to all other Java-based programs as well. To achieve a better precision than earlier approaches we make use of a very accurate model of the Android lifecycle and a large set of sources and sinks, which can easily adapted to new APIs in forthcoming versions of Android. We made the implementation publicly available (see Chapter 4).

To ensure the quality of our analysis we evaluated it on common benchmark applications and compared it to existing tools. Additionally, we provide DroidBench, a benchmark suite specifically designed for analyses on Android, which contains 39 test apps. Moreover, it is planned to extend and update the test suite as new challenges arise.

[1] Available at `https://play.google.com/`

```
1  public class LeakageApp extends Activity{
2      private User user = null;
3      protected void onRestart(){
4          EditText usernameText = (EditText)findViewById(R.id.username);
5          EditText passwordText = (EditText)findViewById(R.id.password);
6          String uname = usernameText.toString();
7          String pwd = passwordText.toString();
8          this.user = new User(uname, pwd);
9      }
10     //Callback method; name defined in Layout-XML
11     public void sendMessage(View view){
12         if(user != null){
13             Password pwdObject = user.getPwdObject();
14             String password = pwdObject.getPassword();
15             String obfPwd = ""; //must track primitives
16             for(char c : password.toCharArray())
17                 obfPwd += c + "_"; //must handle concat.
18
19             String message = "User: " +
20                user.getUsername() + " | Pwd: " + obfPwd;
21             SmsManager sms = SmsManager.getDefault();
22             sms.sendTextMessage("+44 020 7321 0905", null,
23                 message, null, null);
24         }
25     }
26 }
```

**Listing 1.1:** Motivating example

## 1.3 Terminology

A few concepts and terms are used widely in this thesis, therefore they are explained at the beginning.

**sensitive data** Sensitive data is information that is confidential, either because it contains private content or it allows others to identify and track a person. In context of smart phones, typical sensitive data is the address book, the location of the user or identifiers of device or user (IMEI[2] or IMSI[3]).

**(data) source** A source is a call into a method which reads data from a shared resource and returns non-constant values [3]. We assume that data from this source (e.g. the address book or the current location) is sensitive.

**(data) sink** A sink is a call into a method which writes data to a shared resource and accepts at least one non-constant data value causing a change in the values of the resource [3]. The data which flows into a sink leaves the current environment and cannot be controlled by it any more. Therefore we treat sensitive data at a sink as exposed to the public. In the smart phone context typical sinks are the internet, publicly accessible storage (because other apps can read it) and SMS transmission.

**(data) leak** A leak is a data flow which propagates from a source method to a sink method causing the leakage of sensitive data.

## 1.4 Structure of the Thesis

The following chapter covers a wide range of background topics to provide a better understanding of the subsequent chapters, amongst others we give a short introduction to static analysis in general, intermediate representations and the Android operating system. Chapter 3 describes the algorithm of our analysis in detail, while Chapter 4 highlights the architecture, frameworks and selected components of our implementation as well as current limitations. It is followed by

[2] International Mobile Equipment Identity
[3] International Mobile Subscriber Identity

an evaluation chapter which introduces our benchmark suite DroidBench, presents FlowDroid's performance on test and real-world applications and compares it to commercial tools. Chapter 6 contains a comparative summary of the related work subdivided in analysis approaches for Java and specifically for Android. Chapter 7 comprises open issues and ideas for future improvements and research. Our findings are concluded in Chapter 8.

## 2 Background

This chapter provides basic information about the theoretical concepts that FlowDroid is based upon. First we give a general introduction to static analysis, its areas of application, benefits and differences to dynamic analysis (see Section 2.1). In Section 2.2 we explain a specific kind of data flow analysis, the taint analysis, in detail. We then concentrate on IFDS, a worklist algorithm for taint propagation (see Section 2.3). Afterwards we explain the intermediate representation on which our implementation operates as it is necessary to comprehend the formalization of the analysis (see Section 2.4). Furthermore, we give an overview of Android in Section 2.5, as it is the main target of the analysis, although FlowDroid is generic enough to process all sorts of Java applications. Section 2.6 describes the assumed capabilities of an attacker.

### 2.1 Static Analysis

Static analyses inspect the program code to derive information about the program's behavior at runtime. As nearly every program has variable ingredients (inputs from a user, files, the internet etc.) an analysis has to abstract from concrete program runs. Instead it aims to cover all possibilities by making conservative assumptions. The properties derived from these assumptions can be weaker than the program's properties actually are, but they are guaranteed to be applicable for every program run. In this way the analysis detects a program behavior which might not actually happen during runtime, but it does not miss a behavior which can happen during runtime (i.e. leakage of sensitive data). If an analysis features this overapproximation, it is *sound*.
Static analysis has many fields of application. Besides checking for programming errors and security flaws, which aim at the correctness of a program, there are many analyses included in modern compilers which try to optimize programs. For instance, they perform dead code analyses, nullness analyses and live-variable analyses. A brief listing of static analysis tools was assembled by Chess et al. [9] covering simple approaches like the manual use of the Unix tool `grep` to discover undesired language constructs and more sophisticated ones which for instance check if the bounds of arrays are exceeded in C programs.

In general there are two different approaches to static analysis: type systems and data-flow based approaches. Type systems assign properties to components of the program and checks whether they are going to hold during run time. Relating to our context a value which arises from a source is typed with "high", while a value that flows into a sink is typed with "low". A type error would be reported if a value is typed both with "low" and "high", which means that there is a source-to-sink connection. Unfortunately, type systems are naturally neither flow- nor context-sensitive [25]. Therefore, we decided to implement our analysis as a flow-based approach, on which we will concentrate in the following.

Modern sophisticated tools convert the input (either bytecode or source code) to intermediate representations on which they can efficiently operate. To model the program flow they create control-flow graphs (CGF) and call graphs. While the former represent intra-procedural sequences of statements, the latter contain edges between a call site and the call target. Usually it is not possible to determine these targets unambiguously: The method invoked by the call site can refer to the implementation of the class specified in the call site or any other subclass. For example, a class `A` defines the method `m()` and has a subclass `B`. The call site `x.m()` can either refer to the implementation of `A` or `B`, depending on the initialization of `x`, which might not be statically resolvable. Hence edges to all possible call targets are created. Depending on the field of application the graph can be more or less precise, whereupon in general less precise graphs can be generated faster, but have more spurious edges, which make it more expensive to process them.

Dynamic analyses are performed at runtime. They observe one or more concrete program runs and are able to give detailed information to this specific run, but in many cases it might be difficult to derive general statements that are valid for all program executions. Generally they require less computations and less time, but while in static analysis it is challenging to find an abstraction with a good trade-off between precision, recall and performance, for dynamic analysis it is difficult to find a small set of program executions that covers all relevant behavior of the application [13].
In an information-flow based approach we distinguish multiple types of flows [12]. Given the program extract in Listing 2.1 we can observe several flows: Clearly, the first line contains a direct explicit flow from `y` to `x`. Because the assignment to `w` and therefore the value of `w` depends on `x`, there is a direct implicit flow from `x` to `w`. As there is a flow from `y` to `x` and from `x` to `w`, there is an indirect flow from `y` to `w`.

### 2.2 Taint Analysis

Beside the differentiation of analyses in static and dynamic there are different objectives that an analysis has such as tracking variables which are read after certain program execution point (live variable analysis) or deciding which variables are currently initialized (nullness analysis). FlowDroid utilizes a taint analysis, which is a special type of data flow analysis.

```
1 x = y;
2 if(x){
3     w = z;
4 } else{
5     w = v;
6 }
```

**Listing 2.1:** Different types of flows

```
1 public String source();
2 public void sink(String arg);
```

**Listing 2.2:** Source and sink method for the following examples

The latter tracks data along the program execution path. It can be performed both forwards and backwards. In contrast to information flow analysis, taint analysis does not necessarily consider implicit flows. Therefore program jumps created by constructs like if-conditions can leak clues about the tracked data without being noticed.

A taint analysis tracks data from predefined data sources to predefined data sinks and aims at discovering all connections between these sources and sinks. It is often used for security-relevant tasks. When the analysis focuses on the integrity of the application, untrusted inputs are specified as sources and should not reach sensitive sinks. This definition holds for web applications, where user inputs should not enter security-sensitive sinks (such as databases) without being sanitized. As stated in Section 1.3 we employ a privacy-based view in which sources are treated as host of sensitive (user-)data and sinks are untrusted.

Hammer et al. [25] describe the area of conflict in which the analysis operates: On the one hand an analysis should be correct in the sense that it finds all data leaks (no false negatives), on the other hand it should provide a high precision by not reporting false positives while being able to cope with realistic applications in reasonable time. Additionally the analysis should be practicable in terms of being effortless to use. The latter correlates with high precision because a high amount of false positives causes an infeasible burden on the user who have to manually check all results.

In the following we explain necessary conditions for a precise analysis. As it is easy to mix them up we explain the different "sensitivities" with taint propagation examples. The two methods in Listing 2.2 represent a source and a sink. An analysis is flow-sensitive if it takes into account the order of the statements. Given the code example in Listing 2.3 a flow-sensitive analysis notices that there is no data flow from the source method to the sink, because the array is passed to a sink method in line 2, but the tainted information are inserted in line 3. While it might be trivial to achieve flow-sensitivity in this trivial example, it is harder if source and sinks are called in different methods. To track objects returned by method calls which are executed on different receiver objects, context-sensitivity is required. For all calls the call site (i.e. the label of the instruction) is considered as context information, especially calls with different arguments are computed individually. In the code example in Listing 2.4 a context-sensitive analysis is able to distinguish the two call sites, either by different call arguments or by the instruction label. In our example, an easy instruction label could be the line number: The first call is located in line 2 and the second call resides in line 3. Thus, the analysis reports no source-to-sink connection because the tainted return value is not applied to all call sites. For this example 1-context-sensitive analysis is sufficient, but in practice there might be cases where not only the call site of the method, but also recursively the call site of the method which contains the method call has to be taken into account leading to n-context-sensitivity.
Milanova et al. [40] and Smaragdakis et al. [48] argue that object-sensitivity is a subtype of context-sensitivity and context-sensitivity as described above is called call-site-sensitivity. Object-sensitivity considers the allocation site of the receiver object of a call to differentiate the call sites. An object-sensitive analysis should not find a data leak in Listing 2.5, because it maps the object used in the call in line 5 to the allocation site in line 2. Similar to call-site-sensitivity, object-sensitivity can have different levels of precision dependent on the levels of the call-stack it examines. Note that

```
1 void flowSensitive1(){
2     String s = "value";
3     sink(s);
4     s = source();
5 }
```

**Listing 2.3:** Code which requires flow-sensitive analysis

```
1  void contextSensitive1(){
2      String s1 = id(source());
3      String s2 = id("123");
4      sink(s2);
5  }
6
7  String id(String s){
8      return s;
9  }
```

**Listing 2.4:** Code which requires context-sensitive analysis

```
1  void objectSensitive1(){
2      IntermediateObject i1 = new IntermediateObject("123");
3      IntermediateObject i2 = new IntermediateObject(source());
4
5      sink(i1.getValue());
6  }
7
8  class IntermediateObject{
9      String value;
10
11     public IntermediateObject(String s){
12         value = s;
13     }
14
15     public String getValue(){
16         return value;
17     }
18 }
```

**Listing 2.5:** Code which requires object-sensitive analysis

1-object-sensitivity cannot differentiate multiple method calls on the same object, so call-site-sensitivity is also required for taint analysis. There are three possibilities to handle fields [35]. A field-sensitive solution is the most precise way to track data. All fields of a base object are treated separately as well as all base objects are handled individually. Note that this definition overlaps with the above definition of object-sensitivity. A field-based solution tracks all fields, but relates them to the set of objects of the base class instead of their concrete base objects, which is less accurate. Field-insensitive (also called field-independent [35]) approaches do not support fields at all and merge taint information to the base object. A field-sensitive analysis is able to differentiate the fields used in the example in Listing 2.6 and therefore does not report any leak.

While less precise approaches are more scalable and have areas of application like call-graph generation, our taint analysis requires a high precision, so we aimed to implement flow-, context-, object- and field-sensitivity. In our implementation the precision is configurable by the length of the access path (see Section 4.4.7).

## 2.3 IFDS

Our inter-procedural data flow analysis problem is formulated in terms of the IFDS framework by Reps et al. [45] and can therefore be reduced to a graph-reachability problem. IFDS solves inter-procedural, finite, distributive subset problems, which implies the following limitations: The set of data flow facts has to be finite and the data flow functions must distribute either over the union or intersection operator.

The algorithm works by creating a so-called *supergraph*, a directed graph which represents the program by assuming all statements in the program to be nodes. The nodes of consecutive statements are connected via edges. Statements representing procedure calls are an exception, because they are split into two nodes: a call node and a return-site node. One can imagine the graph consists of flow graphs for each procedure which all have intra-procedural edges

```
1  void fieldSensitive1(){
2      ObjectA o1 = new ObjectA();
3      ObjectB o2 = new ObjectB();
4      o1.field1 = "123";
5      o1.field2 = source();
6      o2.field1 = source();
7      sink(o1.field1);
8  }
9
10 class ObjectA{
11     String field1;
12     String field2;
13 }
14
15 class ObjectB{
16     String field1;
17     String field2;
18 }
```

**Listing 2.6:** Code which requires field-sensitive analysis

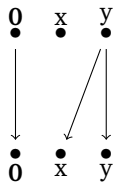


**Figure 2.1:** Non-locally-separable flow function

and connections between them represented by inter-procedural edges. IFDS distinguishes four different edges in the supergraph:

call edges are inter-procedural edges which point from the call node of a procedure call to the start node of the called procedure.

return edges are inter-procedural edges which point from an exit node of the method, for instance a return statement or the end of the method, to the return-site node of the call statement.

call-to-return edges are intra-procedural edges from the call node of a procedure call to the return-site node of this call. They are used to preserve flow facts representing local semantics which do not enter the method call.

normal edges are all other intra-procedural edges, for example assignments or `new` statements.

Note that not all paths in this graph must be considered, but only the valid ones, which respect that a called procedure must always return to its call statement.
A flow function defines the impact of a statement on a set of flow facts. A fact can be data holding information that is valuable for the specific analysis objective. In our case we propagate facts containing amongst others information about the tainted variable and the origin of the taint. As flow function decide which values are generated, killed or transferred they are also called *transfer functions* and are also named according to their underlying edge type, e.g. the normal flow function is a function that belongs to a normal edge. For example, the statement $x = y$ would be associated with the normal flow function that maps a fact set $\{y\}$ (i.e. $y$ is tainted) to a fact set $\{x, y\}$ ($x$ and $y$ are both tainted). Figure 2.1 shows this graphically: The nodes on the top represent the state before executing the flow function, the nodes on the bottom show the state after applying it. Note that the variable $x$ is overwritten and therefore any earlier taint of $x$ is not propagated, as denoted by missing arrows from the upper $x$. The special value **0** is always true and is required to create the taint unconditionally (i.e. at a source statement there is an arrow from **0** to the assigned variable). Hence, there is always an arrow from **0** to **0**.

From the supergraph $G^* = (N^*, E^*)$, the set of flow facts $D$, the distributive flow functions $F \subseteq 2^D \to 2^D$ and a mapping $M : E^* \to F$ from edges of the supergraph to flow functions an *exploded supergraph* is derived. The nodes of this exploded

```
1 double d1 = 9.8;
2 double d2 = 12.4;
3 int i = (int) (d1 + d2);
```

**Listing 2.7:** Java code example to demonstrate conversion in intermediate representation

```
1 double d1, d2, temp$0;
2 int i;
3 d1 = 9.8;
4 d2 = 12.4;
5 temp$0 = d1 + d2;
6 i = (int) temp$0;
```

**Listing 2.8:** The code from Listing 2.7 converted to Jimple

supergraph are created by forming $|D|+1$ pairs of the original supergraph node $n \in N^*$ and a data flow fact $d \in D \cup \{\mathbf{0}\}$. By traversing the exploded supergraph the data flow analysis problem is converted into a graph-reachability problem.

Function calls are modeled using summary functions, which make the approach precise and efficient: At different call sites to the same method $m$, the summary function for $m$ is just reused (gaining efficiency) but it is applied to the taint information at that specific call site (yielding full context sensitivity). Section 3.1 gives more details about how FlowDroid constructs the supergraph.

Once the supergraph is constructed, the algorithm decides whether a variable $x$ at a statement $s$ is tainted simply by computing whether the node representing $(s, x)$ is reachable from a given start node $(s_0, \mathbf{0})$. Because all method calls have been abstracted through summary functions, those queries are fully intra-procedural and therefore highly efficient.

## 2.4 Intermediate Representation

As stated before, static analyses can operate on intermediate representations (IRs) more efficiently than on source code as they are less complex. Our design is based on a stackless, typed 3-address code. Therefore, we give a short introduction to the properties of this representation. To explain the properties we chose the Jimple [53] syntax which is used in our implementation, although the general design is applicable to all other intermediate representations which fulfill these properties. An IR is stackless if it does not support stack operations - instead, variables are assigned to stack positions. The 3-address code restricts statements to have at most one reference on the left side of an assignment and at most two references on the right side of an assignment - except for method calls, which can have more arguments. Only one operation per statement is permitted (for instance, the "+" operator and a cast operation cannot be part of the same statement), hence, nested operations are split into individual ones by using temporary variables. For example, the Java statement in line 3 of Listing 2.7 contains a summation and a cast. Apart from moving the declarations to the top the nested statement is separated into two individual statements (see Listing 2.8). The length of the resulting code will exceed the one of the original source code, but it is easier to parse due to a restricted number of different expressions (Jimple has only fifteen types of statements). Note that even temporary variables have an explicit type. Furthermore, loops are represented by conditional and unconditional jumps. Therefore, statements are not nested and can thus become atomic nodes of a control-flow graph.

## 2.5 Android

Android is an operating system for mobile devices which is developed and maintained by the Open Handset Alliance under the direction of Google. As a recent study [11] shows it is one of the most important operating systems in the mobile-phone market: It achieves a market share of 75% and a growth rate of 79.5% over the past year. Functionality can be added by installing apps from various sources. Google itself hosts an official application store called "Google Play Store"[1] which already contained more than 700.000 apps in march 2013[2]. This makes it an attractive target for malware developers with different motivations, for example they can try to gain a financial benefit (e.g. by calling premium numbers) or collect private information. In this master thesis we concentrate on attackers who try to capture sensitive data.

To prevent those attacks Android implements security measures at its various levels. The system architecture is composed

[1] Available at https://play.google.com/

[2] see http://officialandroid.blogspot.de/2013/03/celebrating-google-plays-first-birthday.html, accessed 05/26/2013

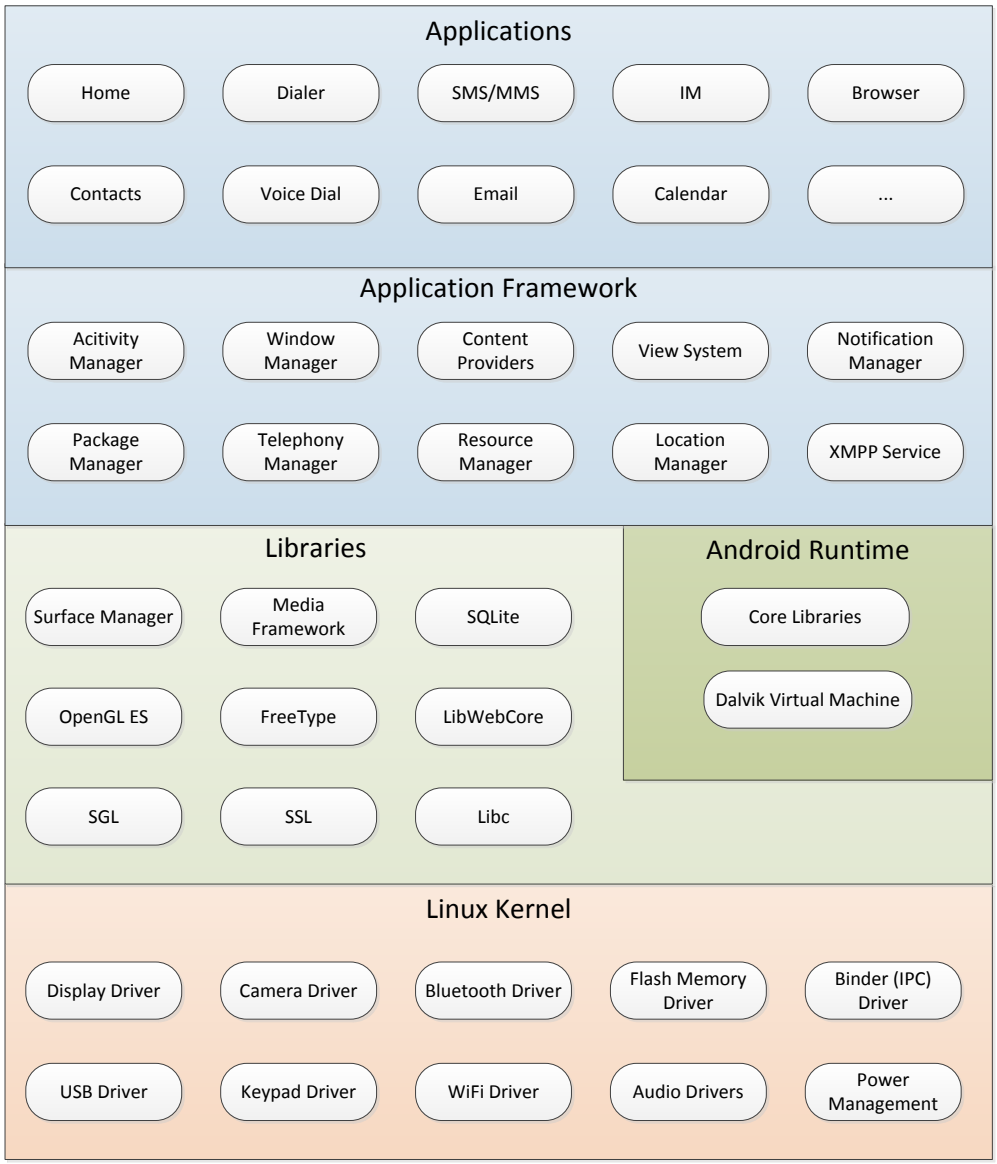


**Figure 2.2:** Android software stack, reproduced from [1]

of different layers [1] which can be visualized as a stack (see Figure 2.2). The lowest level contains the Linux kernel and hardware drivers including all security mechanisms of a Linux system, e.g. process isolation and a user-based permission model. The next level consists of various libraries (for example for databases, media and graphics) and the Android Runtime. On top of this resides the Application Framework which offers different operation system functionality - for example the handling of the user's location and information about telephony services, both containing private information. To access these information applications have to request permissions [20]. The developer of an app has to declare the permissions that the app requires and the user can choose during the installation process: Either she does not want to grant the permission and quits the installation or she grants all requested permissions. The topmost layer is comprised of the applications themselves including the basic ones integrated in Android (like SMS, a clock and the dialer), apps that are delivered by the smart phone vendor and the ones that are installed by the user. Each app runs in its own virtual machine [47] and can store private data which is not accessible for other apps by default. Apps can communicate with each other by using so-called intents which can also be protected by permissions.
However, there are some issues that compromise the effectiveness of this permission-based security model. Often developers declare too many permissions. Felt et al. [14] discovered that from their set of 940 apps from the Android Market (the predecessor of the Google Play Store) one third was overprivileged. They concluded that the high amount of extra permissions is mainly caused by developer confusion. From a user's point of view, there are very few options: They

have to accept all permissions if they want to use the app - otherwise it cannot be installed. In some cases a subset of permissions would be preferable, especially if the user does only want to use a subset of the functionality that does not require some of the permissions. Furthermore, coarse-grained permissions cannot be narrowed down. For example, a social-network app definitely needs the permission *android.permission.INTERNET* to access their servers and fetch content (i.e. activities of friends) and it might also requests the permission *android.permission.READ_CONTACTS* to suggest new friends to the user based on the registered e-mail addresses. Unfortunately the app cannot only read the contacts' mail addresses but also all other stored information which can be used to create or complete an (internal) profile of people which are not even registered to their service. Even worse, in theory the app can send this sensitive information to a completely different server. In this case, it would be convenient to allow only connections to servers which belong to the social network and to restrict access to the relevant part of contact data. Trojans can take advantage of this principle: They are integrated in apps that require permissions for internet and access to private data to work correctly. The malicious part can also use these permissions which is unrecognizable for the user.
Applications have to be signed by the developer. This does not ensure security in any way but allows apps which are signed with the same private key to establish a trust relationship [2]. This relationship allows applications to run in the same process and share code and data through signature-based permissions.

Google itself has acknowledged the problem of malware, and has started to take measures through dynamic analyses. Hence, Google's Bouncer [29] was integrated into the Play Store in 2011. It scans apps for known malware (possibly statically based on signatures, this is not explicitly stated by Google) and checks for malicious behavior by test-running them for five minutes as they are uploaded into the store. Unfortunately, apps can easily circumvent such measures by just holding off from suspicious activity for the prescribed time, or by recognizing the analysis environment through their IP addresses or other clues [43].

From a technical point of view Android apps are Java applications with interfaces described in XML syntax. Their execution is triggered by the Android framework which also provides interfaces to access the hardware and system functionality. Thus, they have no distinct main method. Instead, an app can consist of four different components:

**activity** represents a single screen visible to the user

**service** performs actions in the background like playing music

**content provider** is responsible for storing data in a database-like structure

**broadcast receiver** listens to global events and triggers predefined actions

All components have to be registered in the `AndroidManifest.xml` file (which also includes the required permissions). An application can define one activity which is loaded when the user opens the app via a shortcut icon in the launcher. To integrate a new component in the app a developer has to subclass the corresponding abstract class provided by the framework and overwrite the contained lifecycle methods. By doing so, a specific behavior in response to events like starting, stopping and pausing the component can be defined. Figure 2.3 shows the lifecycle for an activity. If the activity is running in the foreground (State "Activity running" in Figure 2.3) it responds to user interactions via listeners to UI elements. Note that the lifecycles of the other components are slightly different, but less complex. Unfortunately, the lifecycles are more complex than depicted and explicitly documented. Additional methods for saving and restoring state, as well as callbacks that notify the app about additional state changes exist, which have to be taken into account, too.

## 2.6 Attacker model

We assume an attacker that can supply an app with arbitrary malicious bytecode, obfuscated or not. The attacker's goal is to leak private data, especially by using a broad set of permissions granted by the user [4]. Our analysis makes sound assumptions on the installation environment, in particular other apps that are installed on the device, and app inputs, meaning that the attacker is free to tamper with those as well. FlowDroid does assume, however, that the attacker has no way of circumventing the security measures of the Android platform. Also, right now no static analysis for Android, including FlowDroid has a way of dealing with dynamic loading and reflection. Bodden et al. showed how those features can be handled in general [8], however their approach requires access to a load-time instrumentation API, which is something that Android does not currently support. Furthermore, we do not consider data leaks through timing or other side-channel attacks.

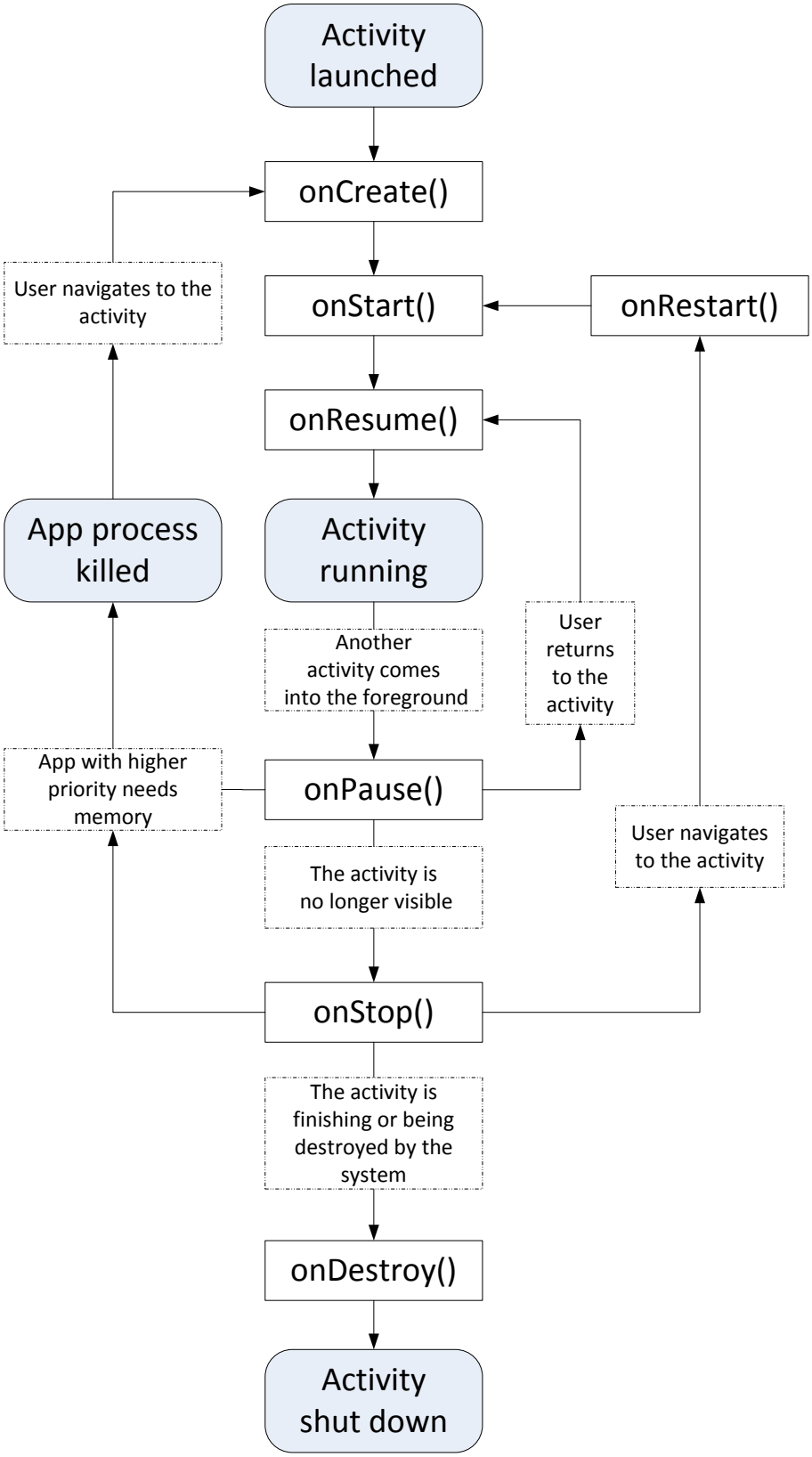


**Figure 2.3:** Android Activity lifecycle, reproduced from [15]

## 3 Design

This chapter gives an overview of the structure of the analysis and defines the flow functions (see Section 3.1). Furthermore, Section 3.2 describes the concept used for generating the main method.

### 3.1 Flow Functions

In this section we give important details about the different transfer functions our analysis associates with program statements. In the following, first we define an abstract model which we use to explain our general taint analysis in Section 3.1.1 and our on-demand alias analysis in Section 3.1.2.

FlowDroid leverages IFDS to propagate each tainted value individually. When processing any given statement, the set **T** of incoming taints are transformed into a set of outgoing taints. At control-flow junctions taints are merged using the union operator. We define the abstract analysis semantics with the standard notation used similarly by Tripp et al. [51]. Table 3.1 shows our abstract domain. For representing the effects of program statements, FlowDroid uses the semantics defined in Table 3.2. $H \in Heap$ defines the current heap, and $E \in Env$ the current environment. A program state is defined as $\sigma = \langle \mathbf{E}, \mathbf{H} \rangle \in States$.

To be able to explain our taint propagation algorithm in detail we declare the following helper functions:

$arrayElem(x) : VarId \rightarrow Boolean$ returns true iff $x$ references an array element

$static(x) : FieldId \rightarrow Boolean$ returns true iff $x$ is a static field

$immut(x) : VarId \rightarrow Boolean$ returns true iff $x$ is a primitive or immutable data type (int, String, etc.)

$source(s) : Stmt \rightarrow \mathscr{P}(VarId)$ returns a set of variable names tainted by the source statement $s$ or $\emptyset$ if $s$ is no source

$native(s) : Stmt \rightarrow Boolean$ returns true iff $s$ contains a call to a native method

$nativeTaint(s) : Stmt \rightarrow \mathscr{P}(VarId)$ returns values which are tainted after the native call $s \in Stmt$. Such values can include the base object on which the method was invoked, the return value, or one or more of the input parameters.

$hasCallEdges(s) : Stmt \rightarrow Boolean$ returns true if $s$ contains a method call which has at least one call edge in the supergraph. This corresponds to the availability of the method body.

To model deep object sensitivity, FlowDroid does not simply propagate simple fields like $x.f$ but instead so-called *access paths* [51] up to a fixed length. $x.f.g.h$ for instance models an access path of length 3. We use the notation $x.f^n$ to describe an arbitrary but fixed access path of length $n$, rooted at $x$. For example $x.f^3$ represents paths such as $x.f.g.h$. Note that $x.f^0$ is equal to $x$. The notation allows us to split field accesses such that $x.f^p = x.f^n.f^m$ where $p = m + n$. In result, **T** is actually a set of currently tainted access paths.

A concrete state is a program state extended by the set of tainted access paths **T** resulting in a triple $\sigma = \langle \mathbf{E}, \mathbf{H}, \mathbf{T} \rangle$. At the beginning it holds that $\mathbf{T} = \emptyset$. Tainted access paths are added to the set whenever the analysis reaches a call to a source, or when processing a statement that propagates an existing taint to a new memory location. We next explain the different transfer functions that FlowDroid uses to compute taints. Section 3.1.2 explains how we use access paths to deal with aliasing.

| Name | Description |
|---|---|
| $VarId$ | Program variables |
| $FieldId$ | Field identifiers |
| $Stmt$ | Statements[2] |
| $Loc$ | Memory locations |
| $Val = Loc \cup \{null\}$ | Values |
| $Env : VarId \rightarrow Val$ | Environment |
| $Heap : Loc \times FieldId \rightarrow Val$ | Heap |
| $States = Env \times Heap$ | Program States |

**Table 3.1:** Formalization domains

[2] FlowDroid operates on an intermediate representation that represents compound statements through individual atomic statements.

| Statement | Semantics |
|---|---|
| x = new Object() | $\sigma = \sigma[\text{x} \rightarrow \text{o} \in \text{Loc. o is fresh}]$ |
| x = y | $\sigma = \sigma[E(x) \rightarrow E(y)]$ |
| x.f = y | $\sigma = \sigma[H((E(x), f)) \rightarrow E(y)]$ |
| x = y.f | $\sigma = \sigma[E(x) \rightarrow H((E(y), f))]$ |

**Table 3.2:** Program semantics

### 3.1.1 Taint Analysis

The taint analysis starts directly at each of the identified and reachable sources. The IFDS framework distinguishes four different kinds of flow functions according to the edge types they operate on: normal, call, return and call-to-return (see Section 2.3 for details). For now, FlowDroid uses a simple two-element security lattice but it can easily be extended to track more information about the kind of taint that is propagated. In our implementation we added additional logic to permit Taintwrappers (see Section 4.4.4), which is not considered here.

#### Normal flow function

Normal flow functions are applied at all statements that are neither calls nor returns. In FlowDroid, only method calls can be the original source of a taint. Thus, a normal flow function can never generate new taints, it can only transfer, preserve, or "kill" existing taints.

FlowDroid is insensitive to array indices, tainting the entire set of array elements even if the program taints just a single element. To be sound, FlowDroid thus needs to assume that the entire contents remain tainted, even if the single array element is overwritten by an untainted value later-on. For an assignment statement $s \in Stmt$ with the structure $x.f^n = y.f^m$ with $n, m \in \mathbb{N}_0$ the following rules apply:

$$T \xrightarrow{s} \begin{cases} T \cup \{x.f^n.f^p\} & \forall p : y.f^m.f^p \in T \\ T \backslash \{x.f^n\} & y.f^m.f^* \notin T \wedge \neg arrayElem(x.f^n) \\ T & \text{otherwise} \end{cases}$$

A special case is the `new` statement which creates a fresh object:
$T \xrightarrow{x.f^n = new \ldots} T \backslash \{x.f^n.f^* \in T | \neg arrayElem(x.f^n)\}$.
Assigning a fresh object erases the taints for the memory location referred to by the left-hand side and all access paths that could be reached through this reference. Since the `new` statement in Jimple has no arguments, new taint cannot be generated.

Assignments of arithmetic operations such as $x = a + b$ are treated by tainting the left-hand side if any of the operands are tainted.

#### Call flow function

Call flow functions handle flows into callees of calls such as $c.m(a_0, \ldots, a_n), n \in \mathbb{N}_0$. To model the context change from the body of the caller to the one of the callee, FlowDroid builds the taint set $T_{callee}$ based on the caller's set $T_{caller}$, replacing references to actual parameters $a_i$ by references to formal parameters $p_i$. If a variable is tainted in the caller's context, FlowDroid converts it to the callee context by replacing $c$ with *this*. Static variables are also copied from $T_{caller}$ to $T_{callee}$, but local variables are not transferred as they are only valid in the caller's context.

$$T_{callee} \xrightarrow{s} \cup \begin{cases} \{this.f^m\} & c.f^m \in T_{caller} \\ \{p_i.f^p\} & a_i.f^p \in T_{caller} \\ \{x.f^q\} & x.f^q \in T_{caller} \wedge static(x.f^q) \end{cases}$$

#### Return flow function

At a return (both exceptional and regular), the return flow function maps taints from the callee's context back to the one of the caller. FlowDroid's return flow function specially treats immutable values. Such values, by their very nature, can never change their taint status. Thus, if a parameter of an immutable type like `String` or `int` was not tainted before the call it cannot be tainted by the callee and is thus still guaranteed to be untainted on return. Local variables are bound to the callee's context and cannot be mapped to the caller. In all other cases, the taint of all tainted access paths is mapped back to the caller's context. The following flow function applies when the callee returns a variable $r$ after being called using a statement of the form $b = c.m(a_0, \ldots, a_n)$:

```
1  void main1(){
2      A a = new A();
3      G b = a.g;
4      foo(a);
5      sink(b.f);
6  }
7
8  void foo(A z){
9      G x = z.g;
10     String w = source();
11     x.f = w;
12 }
13
14 class A{
15     public G g;
16 }
17
18 class G{
19     public String f;
20 }
```

**Listing 3.1:** Code example which requires backward alias analysis

$$T_{caller} \xrightarrow{s} \cup \begin{cases} \{c.f^m\} & this.f^m \in T_{callee} \\ \{a_i.f^p\} & p_i.f^p \in T_{callee} \wedge \neg immut(a_i) \\ \{x.f^q\} & x.f^q \in T_{callee} \wedge static(x.f^q) \\ \{b.f^v\} & r.f^v \in T_{callee} \end{cases}$$

#### Call-to-return flow function

For every call there is also intra-procedural edge propagating all taint values that are independent of the callee. In this function, we generate taints at sources through a simple pattern match against an extensible list of method signatures. We also handle native method calls here (for details, see Section 4.4.3). If access paths starting with one of the arguments or the base object of the call are tainted we check whether the method is investigated by our analysis allowing taints to flow this way. This improves precision as taints could get deleted in the called method. However, if we do not inspect the method, we have to conservatively assume that the taints are preserved. All other taints can be passed on. Again consider statement $s$ with a call $b = c.m(a_0, \ldots, a_n)$:

$$T \xrightarrow{s} \begin{cases} T \cup nativeTaint(s, T) & native(s) \wedge a_i.f^m \in T \\ T \cup \{x\} & x \in source(s) \\ T \backslash \{c.f^*\} & hasCallEdges(s) \\ T \backslash \{a_i^*\} & hasCallEdges(s) \\ T & \text{otherwise} \end{cases}$$

### 3.1.2 On-demand Alias Analysis

During our research we experimented a lot with different ways to resolve aliasing effectively and efficiently. As it turned out, using ahead-of-time analyses like points-to sets is usually too costly (because the analysis computes alias information for *all* program variables, not just those that carry taints) and too imprecise (because the analysis would not support the same level of context-sensitivity as our taint analysis). We hence opted for a demand-driven approach by Tripp et al., which executes within the same context-sensitive IFDS framework as our taint analysis [51]. This on-demand analysis is triggered at assignments to heap variables, i.e., statements of the form `x.f = v`. The alias analysis then walks backward through the control-flow graph. Whenever it finds an alias, it triggers the forward analysis in turn, propagating an aliased taint from the location at which the alias was found. Similar to the forward analysis, we define flow functions which compute the alias propagation information. Let the set $A$ define the alias information. The only initial element in $A$ is the complete access path of the tainted value which caused the alias lookup. The backward solver terminates when $A$ becomes empty.

Applied to the code in Listing 3.1 this means that in line 11, where the forward analysis assigns a taint to `x.f` (which is an

```
1 void main2(){
2     A a = new A();
3     G b = a.g;
4     sink(b.f);
5     foo(a);
6 }
```

**Listing 3.2:** Code example which requires flow-sensitive backward alias analysis

assignment to the heap) the backward analysis searches upwards for aliases of `x.f`. At line 3, the alias `b.f` is found and then propagated forward. Finally, the sink in line 5 is reached with the tainted value.
However, this analysis is not completely flow-sensitive: We apply a modification to our example by exchanging line 4 with line 5 resulting in the code shown in Listing 3.2 (the classes `A`, `G` and method `foo` are unmodified and therefore not listed again). The taint propagation works as before until we taint `b.f` in line 3. In the next line we detect the sink, but at this point we do not know that our taint is not "active" yet. We have just found an alias to the heap object, but the heap object is not tainted yet, because we reside at an earlier point of execution in the program. Without taking this into account we would assume a source-to-sink connection which is not going to happen during runtime.

Therefore we changed the alias lookup by memorizing the statement which taints the heap location, the so-called *activation statement*. Aliased taints get activated after passing this statement again during forward analysis. Aliasing in different levels of the call stack is managed by performing a backward analysis on each level: Additionally to triggering the backward analysis on aliases in normal flow statements, it is also started if a tainted heap element is mapped to the caller's context in the return edge. As aliases do not necessarily have to enter the method containing the activation statement, we also remember the statement containing the call to the method of the activation statement. If we pass this statement, the taint is also activated. We examine this on the first, unchanged example: In line 11, the backwards analysis is started with the inactive taint `x.f` and the activation statement `x.f = w;`. We find the alias `z.g.f` in line 9 which triggers a forward analysis. On line 11 `z.g.f` gets activated again because it reached the stored activation statement. On the return edge from `foo` to `main1`, the taint is converted to `a.g.f` and a backward analysis is started with the additional activation statement `foo(a)`. In line 3 we find the alias `b.f` and due to our additional activation statement, we can activate it in the forward analysis while passing line 4 (in the call-to-return edge). One can easily see that this approach does not report a leak in the code of Listing 3.2.

During the backwards analysis, we can enter called methods by traversing the method's return edges in the opposite direction. However, there is no need to follow call edges to their caller. Thus, the backward analysis is executed not fully inter-procedural. The whole procedure might sound costly, but the implementation of the solver recognizes already calculated facts and do not recalculate them (as it is based on a fixpoint iteration).

#### Normal flow function

For a statement $x.f = y.g$ the following rule applies:

$$A \xrightarrow{s} \cup \begin{cases} A \backslash \{x.f.f^p\} \cup \{y.g.f^p\} & \forall p : x.f.f^p \in A \\ A & \text{otherwise} \end{cases}$$

For `new` statements, FlowDroid erases all alias information for the left-hand side, as fresh objects cannot be aliased:

$$A \xrightarrow{x.f = new \ldots} A \backslash \{\forall m : x.f.f^* \in A\}$$

#### Call flow function

As we reside in the backward flow, we enter methods by traversing their return edges. The following flow function applies when we find a statement $s$ with a method call of the form $b = c.m(a_0, \ldots, a_n)$. We assume that method $m$ has one or more return sites and returns variables $r_0, .. r_m$. In case there is no return value $s$ does not contain an assignment.

$$A_{callee} \xrightarrow{s} \cup \begin{cases} \{this.f^m\} & c.f^m \in A_{caller} \\ \{x.f^q\} & x.f^q \in A_{caller} \wedge static(x.f^q) \\ \{r_i.f^v\} & b.f^v \in A_{caller} \end{cases}$$

#### Return flow function

The backward analysis is only partly inter-procedural. We do not propagate taint from a called method back to the caller. For any return flow we obtain:

$$A_{caller} \xrightarrow{s} \emptyset$$

**Call-to-return flow function**

For a call site $b = c.m(a_0, \ldots, a_n)$, the call-to-return flow function kills aliases of references obtained through $b$:

$$A \xrightarrow{s} \begin{cases} A \backslash \{b.f^n\} & b.f^n \in A \\ A & \text{otherwise} \end{cases}$$

## 3.2 Main Method Generation

While most stand-alone Java applications contain a static main method as single entry point, web services and especially input-oriented Android apps as applications have multiple entry points that are not statically accessible. Applications can consist of multiple components. While services and broadcast receivers run in parallel, activities can only run sequentially. However, we cannot pre-estimate their order, because it might depend on user input. Therefore we conservatively assume that all components can run in an arbitrary sequential order. Within each component, the lifecycle allows only certain execution orders of the framework method (as already mentioned in Section 2.5). However, UI callbacks are an exception: They can also occur in arbitrary order, but only while the corresponding activity is running. Again FlowDroid assumes that callbacks can be invoked in any possible order.

Fortunately FlowDroid bases its analysis on IFDS, which is not path-sensitive: It joins all analysis results immediately at any control-flow merge point allowing to generate a main method in which every order of components and callbacks is possible. In contrast, path-sensitive solutions have to consider each possible program graph separately, which would cause high costs if all possible program paths have to be taken into account separately.

Note that we generate an individual dummy main method for each app analyzed. Each main method will only involve the fraction of the lifecycles that, according to the app's XML configuration files and the code we have analyzed, can actually occur. Disabled activities are automatically filtered and callback methods are only invoked in the contexts of the components to which they actually belong.

We build a `dummy main method` which contains all calls. Figure 3.1 shows the control-flow graph of the dummy main method for the example app presented in Listing 1.1. Note that the opaque predicate `p` cannot be evaluated statically by FlowDroid (for instance, it can be a check involving an environment variable). Hence, the analysis will consider both branches.

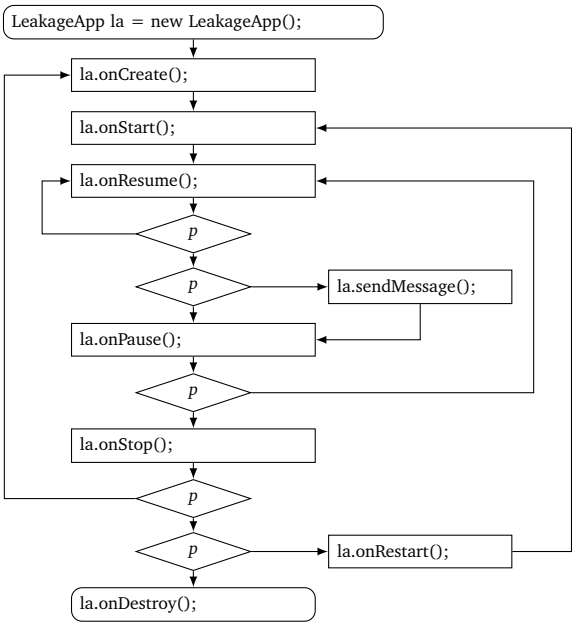


**Figure 3.1:** CFG for dummy main method, taken from [15]

# 4 Implementation

In this chapter we describe the implementation details of FlowDroid starting with the overall architecture (Section 4.1) and the used frameworks (Section 4.2). Section 4.3 highlights an improvement for the implemented flow functions. Subsequent sections explain selected implementation details (Section 4.4) and FlowDroid's current limitations (Section 4.5).
The implementation can be found at GitHub, whereas the core analysis resides at:

`https://github.com/secure-software-engineering/soot-infoflow/`

Android-specific pre-processing for sources, sinks and entrypoints can be found at:

`https://github.com/secure-software-engineering/soot-infoflow-android/`

## 4.1 Overview

Figure 4.1 shows the different steps which are necessary to analyze either an Android app or a Java application. Android applications are packaged in *apk* files (Android Packages), which are essentially zip-compressed archives. After unzipping an archive, FlowDroid searches for lifecycle and callback methods as well as calls to sources and sinks in the application. This is done by parsing various Android-specific files, including the layout XML files, the *dex* files containing the executable code and the manifest file defining the activities, services, broadcast receivers and content providers in the application. We describe the detection of *UI Interactions* in detail below (Section 4.4.6). If we want to analyze a plain Java application we have to manually define the sources, sinks and entrypoints and start directly with the following actions.

Next, FlowDroid generates the main method from the list of entrypoints. This main method is then used to generate a call graph and an inter-procedural control-flow graph (ICFG). We then detect all sources which are reachable from the given entry points. Starting at these sources, the taint analysis tracks taints by traversing the ICFG as explained in Section 3.1. *Native Calls* require a special treatment which is described below (Section 4.4.3). Furthermore, we added a feature called *Taint Wrapping* (see Section 4.4.4) which can be used to substitute code which is not available for analysis as well as for performance optimization.

At the end, FlowDroid reports all discovered flows from sources to sinks. Depending on the options the user has chosen either the whole path with all intermediate variables is displayed or only the source and the sink statement. We provide this information in the unmodified Soot classes so further processing is possible (for example an automatic mitigation strategy could be applied by sanitizing the values).

Currently 171 JUnit test cases ensure that FlowDroid works as specified and future changes do not affect its functionality in a negative way. Detailed information on the test cases can be found in Section A.2.

## 4.2 Frameworks

Instead of building the whole analysis from scratch, we chose to rely on some actively maintained open source frameworks which provide the required basic functionality such as transformation into an easier intermediate representation, call graph generation and the IFDS algorithm.

### 4.2.1 Soot

FlowDroid extends the Soot framework [34] which provides important prerequisites for a precise analysis, in particular a very accurate call graph. Through a plugin called Dexpler [5] Soot supports not only converting Java code but also Android's *dex* files into the Jimple [53] intermediate representation (see Section 2.4) which allows us to implement our analysis for both targets. Soot requires a static main method as entry point. Because Android does not provide such a method, we had to build them on our own (see Section 4.4.2 for more details). Soot offers many options allowing to customize the behaviour of the analysis. We experimented with different call graphs to improve performance.

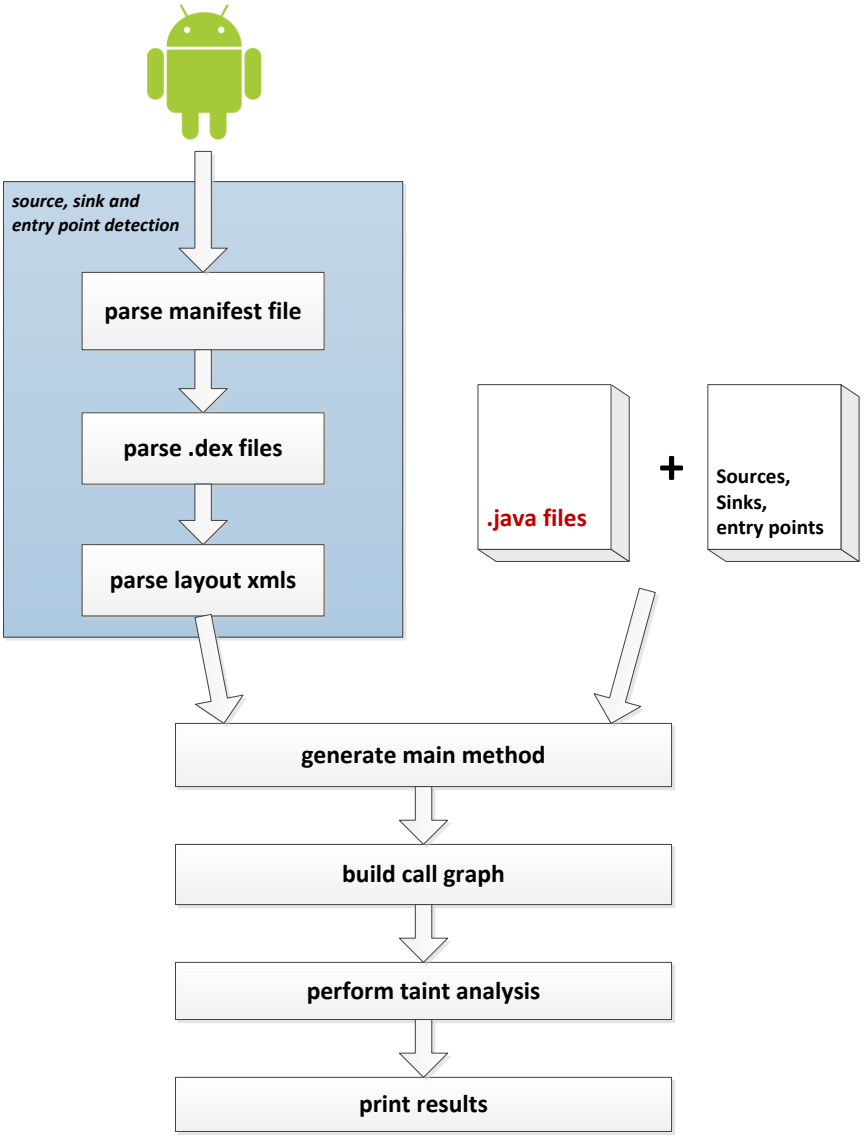


**Figure 4.1:** Overview of FlowDroid

**Call Graph Comparison**

Soot supports four algorithms to create call graphs which will be briefly introduced here (more details can be found in [50]). *Class Hierarchy Analysis* (CHA) is the fastest call graph generation mechanism as it only checks for the statically declared type. This usually results several edges to different method definitions of subtypes if a method is called on that object. *Rapid Type Analysis* (RTA) takes the call-graph generated by CHA, but reduces its size by obtaining additional information about the instantiated classes and removing call edges to methods of classes that were never instantiated. *Variable-Type Analysis* (VTA) creates a call graph by performing a flow-insensitive analysis which helps to build a more precise call graph: Only for methods of classes which instantiation site is connected (via assignment statement or method calls) to the analyzed method call a call edge is added. The *Soot Pointer Analysis Research Kit* (Spark) [35] is included in Soot and creates a pointer assignment graph (PAG) to provide a points-to analysis. Spark can make use of one of the algorithms explained above to generate a call graph, but it can also construct it on-the-fly during computation of the points-to sets of call sites. This approach is more precise, but it is not as fast as the other algorithms as it takes a graph generated by CHA as starting point and refines it in several iterations [35].
Initially we were using Spark with on-the-fly call graph construction, but in search of performance improvements we also investigated the other algorithms. Although the call graph generation is faster for CHA, RTA and VTA, the call graph becomes less precise and therefore far more edges are generated. These additional edges have to be processed by the analysis leading to a longer overall run time for nearly all of our test cases, so we stick to Spark for call graph generation.

```
1 public interface IFDSTabulationProblem<N,D,M, I extends InterproceduralCFG<N,M>>
2         extends SolverConfiguration {
3     FlowFunctions<N,D,M> flowFunctions();
4     I interproceduralCFG();
5     Set<N> initialSeeds();
6     D zeroValue();
7 }
```

**Listing 4.1:** Interface to define an IFDS problem in Heros

### 4.2.2 Heros

Heros [26] is a multi-threaded open-source implementation of the IDE/IFDS algorithm. Although it is general enough to be used with any program analysis framework, it is primarily developed to work with Soot. As described in [7] it provides a lightweight interface to implement an IFDS problem, which is presented in Listing 4.1. The interface extends a `SolverConfiguration` class which allows to configure general settings, for instance the number of threads which should be used by the solver. The generics have the following meaning: `N` represents a node in the inter-procedural control-flow graph, `D` is the type of the data-flow facts (Abstraction in FlowDroid, see Section 4.4.7), `M` is the representation of a method and `I` represents the inter-procedural control-flow graph. The implementation of the interface is straightforward: `flowFunctions()` has to return an object which defines the taint propagation behavior for all four edge types. `interproceduralCFG()` returns the call-graph which must be compliant to the `InterproceduralCFG` interface. FlowDroid implements `initialSeeds()` by returning all statements which include calls to sources and are reachable from an entry point. `zeroValue()` returns a unique constant fact that cannot be created by any statement and is eligible to serve as our **0** value.

## 4.3 Flow Functions

To optimize the performance we decided to start our taint propagation directly before statements that contain source method calls. As explained in Section 2.3 taint is created unconditionally from a special **0** value. Instead of propagating this value throughout our whole call graph, we eliminate it as soon as we find a different value which gets propagated (which always happens after the first appliance of a flow function as we start directly at source method statements). Because statements that create taint unconditionally are not required except on entry points, there is no need to track the **0** value any further. This improves our performance as we do not have to inspect method calls if no taint flows into them. To reflect this change we adapted our call-to-return flow function. Apart from that, the implementation of the flow functions follows the design depicted in Section 3.1.

## 4.4 Components

In the following sections we highlight different parts of FlowDroid which are crucial for a precise and usable analysis. Therefore, we paid great attention to them.

### 4.4.1 Android Sources and Sinks

The recall of a data flow analysis heavily depends on the given set of sources and sinks. In FlowDroid we use SuSi [3], a fully automated machine-learning approach for identifying sources and sinks. One of the advantages are the low expenditures when a new Android version is released which might contain new API methods including new sources and sinks: As the approach does not require manual activity, one rerun on the new source code is necessary to update the resulting list. We ran SuSi on the current Android source code and used the resulting text file as input for our analysis. It contains hundreds of sources and sinks, more than the usual dozen that is used for evaluation in earlier data flow analyses (see Section 6.2). SuSi also categorizes sources and sinks, however, we do not make use of these labels at the moment, but they could be used to create more usable reports (see Chapter 7). Categories include amongst others Account, Bluetooth, Contact, Calendar, Database, File, Location,, Log, Network, NFC, Phone State, SMS_MMS, Settings and Unique Identifiers. Note that some categories contain only sinks, others contain sources and some of them contain both.
However, our implementation can be used with other sorts of sources and sinks as well: We provide the lightweight interface `ISourceSinkManager` (see Listing 4.2) which contains a method for sources respectively strings. The implementation has to decide whether a given call is a source or a sink based on the current call statement and the inter-procedural call graph.

```java
package soot.jimple.infoflow.source;

public interface ISourceSinkManager {
    public abstract boolean isSource(Stmt sCallSite,
            InterproceduralCFG<Unit, SootMethod> cfg);

    public abstract boolean isSink(Stmt sCallSite,
            InterproceduralCFG<Unit, SootMethod> cfg);

}
```

**Listing 4.2:** Interface ISourceSinkManager

```java
package soot.jimple.infoflow.entryPointCreators;

public interface IEntryPointCreator {

    public SootMethod createDummyMain(List<String> methods);

    public void setSubstituteCallParams(boolean b);

    public void setSubstituteClasses(List<String> l);
}
```

**Listing 4.3:** Interface IEntryPointCreator

Inspired by Sbîrlea et al. [46] we added the method `startActivity(android.content.Intent)` of `android.app.Activity` as sink, which is called for inter-component communication. Sensitive data stored in the intent can be sent to other components which are beyond our analysis. This solution is sound, because the leak is not lost, but might be not meaningful enough as the user does cannot track the data flow any further. However, inter-component and inter-app communication is subject to future work.

### 4.4.2 EntryPointCreator

As explained in Section 3.2 we need a main method which models the activity lifecycle and invokes the different entry points. In our implementation we generalized this concept as we want to support all sorts of Java applications. Therefore we developed the `soot.jimple.infoflow.entryPointCreators.IEntryPointCreator`[1] interface depicted in Listing 4.3. We implemented the `BaseEntryPointCreator` which contains all basic functionality used by all others of our implemented entry point creators like handling of simple types and building of method calls and constructors. The subclass `DefaultEntryPointCreator` is used for Java applications. It creates a class which simply calls all given entry points, each of them preceded by an opaque predicate which allows to assume an arbitrary call order (see Section 3.2). The second subclass of `BaseEntryPointCreator`, `AndroidEntryPointCreator`, is comparatively complex: It analyzes the incoming methods and if it finds methods whose class subclasses an Android component it models the calls according to the Android lifecycle. Although we also evaluated FlowDroid on web services, there might be some functionality of Java applications which require a different main method generation. Therefore the required entry point creator can be implemented individually and passed to the `Infoflow` class.

### 4.4.3 Native Calls

Both Java and the Android platform support invoking native methods written in C or other unmanaged languages. For a Java-based analysis, such methods are black boxes which cannot be resolved. During our testing and evaluation phase we collected information about all called native methods and discovered that there was only one native method call which was used very often: `System.arraycopy`. We thus defined an explicit taint propagation rule which declares that third

[1] all other classes mentioned in this section reside in the `soot.jimple.infoflow.entryPointCreator` package. For brevity, we omit the package name in the following.

argument (the output array) will become tainted if the first argument (the input array) is tainted before the call. For native methods without an explicit rule, we conservatively assume call arguments and the return value to become tainted if at least one parameter was tainted before. This is neither entirely sound nor maximally precise but is probably the only practical approximation in a black-box setting.

### 4.4.4 Taintwrapper

Since some datatypes are used very often and complex to analyze, performance can be optimized by manually defining the taint behavior (similar to [52]). This shortcut was implemented to avoid analyzing larger parts of the Java library again and again, but it also allows to consider taints which flow through external libraries whose code is not accessible to the user. FlowDroid offers the high-level `soot.jimple.infoflow.taintWrappers.ITaintPropagationWrapper` interface which allows to adjust the actual implementation to ones needs.
In general every method call which is found by the analysis is handed to the Taintwrapper, which checks if it can deliver taint information for this method. If the Taintwrapper cannot handle the method, the method is analyzed normally - this allows to define TaintWrappers that cover only a subset of all method calls. Similar to giving explicit taint-propagation rules for native calls (see Section 4.4.3) the taint wrapper can either define hard coded rules or implement a logic based on the call statement and current taint information. If no implementation is set, all reachable calls are inspected.
We include an example implementation which reads the taint information from a configuration file. The file lists the methods that add a taint to the base object or return tainted objects if called with tainted arguments. This simple approach works well for the Java collection classes, string buffers and other commonly used data structures.

### 4.4.5 Substitution Classes

J2EE and Android applications sometimes operate on interface types or abstract class types which are instantiated with concrete implementors or subclasses within the J2EE or Android framework. FlowDroid's EntryPointCreator cannot easily be aware of those concrete subtypes. FlowDroid thus allows the user to provide a list of so-called *substitution classes*. A substitution class provides a concrete implementation for one or more of these interfaces, simply for the purpose of the analysis. The user can choose whether to provide a stub or a real implementation, depending on whether she aims for higher precision or scalability. For Android, substitution classes are required - for instance, for the abstract class `android.context.Context`. We also provided substitution classes for the evaluation with SecuriBench Micro (see Section 5.1).

### 4.4.6 Android Component

Besides the Android lifecycle which we have already explained in Section 2.5 there are two major challenges for the precision of our analysis on Android apps: User interface interactions and callbacks. We will introduce both of them now.

**UI Interactions**

UI elements can be taint sources, e.g., if an application prompts the user for a password. FlowDroid thus scans the layout XML files in the *apk* file for text inputs and links them to the source-code statements where they are accessed. This is non-trivial, as the Android operating system manages access to such resources at runtime and a static analysis tool must simulate these runtime APIs as precisely as possible. Note that we need to over-approximate resource accesses in the general case, though. In Android, resource mappings can be configuration-dependent, for instance to support different layouts for smartphones and tablets. In this case, we can only assume all cases as possible. Not all data in text fields is sensitive, though. FlowDroid can be configured to either consider all text fields or to restrict itself to special sensitive fields like password input fields (the default).

**Callbacks**

To model the app lifecycle correctly (see Section 2.5), FlowDroid contains a list of callback interfaces extracted from the Android documentation. FlowDroid first computes a call graph ignoring callbacks, to determine which activity will potentially register which kind of callback at runtime. This process is repeated until no new callbacks are found. Next, FlowDroid generates a customized main method, taking the respective discovered callbacks into account.

### 4.4.7 Flow Fact Abstraction

The most lightweight possibility which is suitable to track taint is to use the tainted access path as flow fact. However, we need more context information. Some of the additional attributes are technically required, others provide a more usable

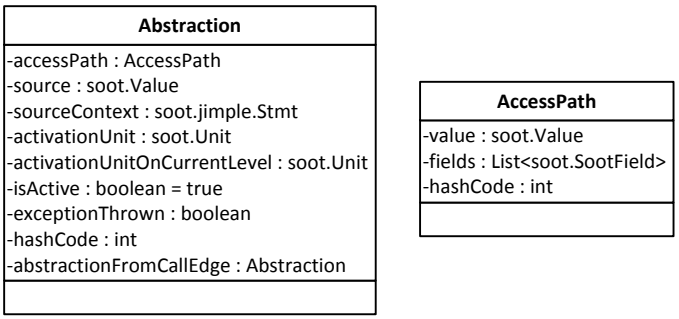


**Figure 4.2:** Class diagram for Abstraction class

result. We spend much time to create an abstraction that does not impede performance, but also contain all relevant information. Figure 4.2 shows the class diagram of our current abstraction class. In the following list we will explain all contents:

accessPath determines the tainted value which the flow fact keeps track of. It consists of a variable and a (probably empty) list of fields. For performance optimization we cache the calculated hash code.

source contains the source, which can be a call to a source method or a parameter of a callback method.

sourceContext provides more information about the source context by storing the whole statement that initialized the taint tracking

activationUnit provides the unit which triggered the backwards alias analysis. Aliases which are found during backward analysis and propagated forwards again are activated (see boolean flag below) when they pass this unit.

activationUnitOnCurrentLevel provides the call statement of the method which triggered the backwards alias analysis or which has a call path to this method (the backward analysis is not limited to a certain level in the call stack). The unit stored in this variable resides in the currently analyzed method and is used to activate the taint information. We cannot use just one activationUnit because we cannot guarantee that we enter this method again: For instance, a local variable is aliased with a call argument. The local variable does not enter this method in forward analysis. Instead, it is activated after passing the corresponding activation unit on the current level.

isActive is a boolean flag describing whether the abstraction represents an active taint or an alias to a taint which is not active yet.

exceptionThrown is a boolean flag which is false by default. It is used to track taints from catch to throw statements: The flag is set to true when the abstraction is propagated by a throw statement. As this flow fact is processed by the corresponding catch statement, the boolean flag is set to false again.

hashCode is stored for performance reasons once it is computed. We take advantage of the fact that all fields which influence the hash value are final, so we do not have to update it.

abstractionFromCallEdge is a technical field. It is required for the transition from backward alias analysis to forward taint analysis, because we cannot generate an edge that "creates" a flow fact out of nothing. Note that even unconditionally created taint values are preceded by an abstraction which contains the **0** value.

In theory, access path can grow infinitely long. Although this is unlikely to happen in real applications, their size can get - depending on the system the analysis is executed on - infeasible large. Therefore we implemented a cutoff length which can be defined by the user (method `setAccessPathLength(int accessPathLength)` in `soot.jimple.infoflow.Infoflow`). This reduces the precision, but improves performance without bothering the soundness of our approach.

## 4.5 Limitations

Although FlowDroid aims to be very precise, it cannot distinguish elements in collections and arrays. Thus, if a tainted element is inserted, the whole collection is assumed as tainted. Although it would be relatively easy to implement the tracking of a tainted element that is inserted with a constant value, indices can also be computed or taken from user input and are therefore not resolvable by static analysis. Hence, we avoid unsoundness by making this conservative assumption. If a field (some object that is stored on the heap) is overwritten, we cannot remove the taint because this requires a must-alias analysis [30]. Because we merge sets of flow facts, our current implementation uses a may-analyis - two variables might be aliased depending on the previous program execution path.
At the moment FlowDroid ignores reflective calls, which is unsound. While specialized static string analyses can be used to simulate reflection to some extent, past research has found such analyses to be incomplete [8], as reflective call targets can be determined by external configuration files. On the Java platform, reflection-analysis tools such as TamiFlex [8] can be used to make static analysis tools aware of reflective calls. Such tools require load-time instrumentation through `java.lang.instrument`, however, which the Android platform does not currently support.
As described above, native code is approximated conservatively using taint wrapping. Another limitation of FlowDroid is its current focus on explicit data flows. Implicit flows caused by control-flow dependencies are currently ignored. FlowDroid also disregard probabilistic and possibilistic leaks caused by multi-threading [19].

# 5 Evaluation

This chapter describes the evaluations we have performed. To test the general functionality of our analysis, we used SecuriBench, a benchmark suite for analyses focused on web applications (see Section 5.1). We decided to design our own benchmark suite, DroidBench, which we present in Section 5.2. Additionally we were looking for Android evaluation apps suitable for our analysis and found InsecureBank and PandemobiumStockTrader, two apps which contain various vulnerabilities (see Section 5.3). To assess FlowDroid's ability to analyze real-world applications, we evaluated some apps from the Google Play Store. The results are described in section 5.4. Beside the examination of FlowDroid's performance on different test sets, we compared it to similar other approaches to validate its usefulness (see Section 5.5).

## 5.1 SecuriBench Micro Tests

FlowDroid was specifically designed for Android and it gains much precision through its complete and precise handling of Android's lifecycle. Nevertheless, there is nothing that would preclude software developers from applying FlowDroid to Java applications as well. To assess how well it performs this use case, we evaluated FlowDroid against Stanford SecuriBench Micro [37] version 1.08, a common set of 96 J2EE micro benchmarks originally intended for web-based applications. Technically, each benchmark class is a subclass of `javax.servlet.http.HttpServlet` with a `doGet` method serving as entry point. For each of them, we manually created a JUnit test and defined the necessary lists of sources, sinks and entry points. Since FlowDroid supports a simple textual file format for defining these parameters, and since all benchmarks cases have the same structure, this was not much effort. For the interfaces `javax.servlet.http.HTTPServletRequest` and `javax.servlet.http.HTTPServletResponse` we needed to provide substitution classes (see Section 4.4.5), as we had no access to framework code that would provide their implementations. Additionally, there are three test cases for which we had to change the return value of a method which fetches the `javax.servlet.ServletConfig` in the `securibench.micro.BasicTestCase` class. We omitted test cases involving sanitization, reflection, predicates and multi-threading from our experiments. As we explained earlier, such features are out of scope for our analysis tool, just as they are for all other existing Android analysis tools known to us.

Before executing the test cases we double-checked all of them manually as the stated amount of vulnerabilities in the provided method and in the comments was not always correct. Table 5.1 shows our test results grouped by test categories. The **TP** column shows the *true positives*, i.e., the number of actual leaks that FlowDroid found. For the example of `Inter`, FlowDroid found 14 out of 16. The **FP** column shows the number of *false positives*, i.e., the findings which FlowDroid reported that did not correspond to actual leaks, but were rather artifacts of an overly approximate analysis. In most cases this number is reasonably low or even zero, except for the *Arrays* category. This is due to an imprecision that FlowDroid shares with most other static analyses: for performance reasons, the tools do not differentiate between multiple components inside the same collection, e.g., different indices in arrays or different positions in lists. Treating such cases precisely and efficiently is an open issue in the static-analysis community. The *n/a* entries in the table correspond to test categories such as reflection, which we identified to be out of scope. Without these omitted test cases FlowDroid achieves 93% precision and 98% recall. A detailed list of all test cases with every single result is appended in Section A.1.

| Test-case group | TP | FP |
|---|---|---|
| Aliasing | 11/11 | 0 |
| Arrays | 9/9 | 5 |
| Basic | 60/60 | 0 |
| Collections | 14/14 | 3 |
| Datastructure | 5/5 | 0 |
| Factory | 3/3 | 0 |
| Inter | 14/16 | 0 |
| Pred | n/a | n/a |
| Reflection | n/a | n/a |
| Sanitizer | n/a | n/a |
| Session | 3/3 | 1 |
| StrongUpdates | 0/0 | 0 |
| Sum | 119/121 | 9 |

**Table 5.1:** SecuriBench Micro test results

## 5.2 DroidBench

While there are benchmark suites for analyzing web applications or specifically for detecting different kinds of Java vulnerabilities [37], at the moment there is no Android-specific analysis benchmark suite. This is not satisfactory because the generic Java test suites do not cover aspects like the Android lifecycle, callbacks or interactions with UI elements, e.g. password fields. Thus, they cannot be used for assessing the practical effectiveness of Android analysis tools. We therefore developed an Android-specific test suite called DroidBench. Version 1.0 contains 39 small Android apps. The suite can be used to assess both static and dynamic taint analyses, but in particular it contains test cases for interesting static-analysis problems (field-sensitivity, object-sensitivity, tradeoffs in access-path lengths etc.) as well as for Android-specific challenges like correctly modeling an application's lifecycle, adequately handling asynchronous callbacks and interacting with the UI. Table 5.2 contains a list of all apps in the suite together with their respective names and short descriptions. The table also indicates whether there is a leak or not and if an analysis tool must model the Android lifecycle or callback infrastructure to be able to successfully analyze the app. In future, we want to add more challenging test cases and also envision our test suite to be extended by other researchers as well and then be used to compare the completeness and correctness of the various taint analysis approaches. DroidBench is available on GitHub:

`https://github.com/secure-software-engineering/DroidBench/`

| App Name | Description | Leak | Lifecycle | Callbacks |
|---|---|---|---|---|
| | **Arrays and Lists** | | | |
| ArrayAccess1 | Stores both a tainted and an untainted value in an array and then leaks the untainted one. Array indices are constants. | ○ | ○ | ○ |
| ArrayAccess2 | Stores both a tainted and an untainted value in an array and then leaks the untainted one. Array indices are calculated. | ○ | ○ | ○ |
| ListAccess1 | Both a tainted and an untainted value are stored in a list. Only the untainted value is leaked. | ○ | ○ | ○ |
| | **Callbacks** | | | |
| AnonymousClass1 | Registers a callback handler for location updates in an anonymous inner class and leaks the incoming location data inside the callback. | ● | ● | ● |
| Button1 | The sink is called after the user clicks a button. The button handler is defined via XML. | ● | ○ | ● |
| Button2 | Only clicking buttons in a specific order leads to a data leak. | ● | ○ | ● |
| LocationLeak1 | Registers a listener for location updates, stores the value and leaks it later in the lifecycle. | ● | ● | ● |
| LocationLeak2 | Similar to LocationLeak1, but the activity class directly implements the callback interface. | ● | ● | ● |
| MethodOverride1 | Overwrites an internal Android method to hide a leak. | ● | ○ | ● |
| | **Field and Object Sensitivity** | | | |
| FieldSensitivity1 | Both tainted and untainted data is stored in a data object; the untainted value is leaked. | ○ | ○ | ○ |
| FieldSensitivity2 | Similar to FieldSensitivity1, but source and sink calls are distributed across the lifecycle. | ○ | ○ | ○ |
| FieldSensitivity3 | Both tainted and untainted data is stored in a data object; the tainted value is leaked. Source and sink calls are distributed across the lifecycle. | ● | ○ | ○ |
| FieldSensitivity4 | Field contents are sent before tainting the field. | ○ | ○ | ○ |
| InheritedObjects1 | Chooses an object's actual type based on a conditional. Only one possible type leads to a leak. | ● | ○ | ○ |
| ObjectSensitivity1 | Writes a tainted value into an object and an untainted one into another object of the same type. Leaks the untainted value. | ○ | ○ | ○ |
| ObjectSensitivity2 | Writes a tainted value into a field and then overwrites it with untainted data. | ○ | ○ | ○ |
| | **Inter-App Communication** | | | |
| IntentSink1 | A tainted value is leaked to another application using an intent. | ● | ○ | ○ |

| App Name | Description | Leak | Lifecycle | Callbacks |
|---|---|---|---|---|
| IntentSink2 | Similar to IntentSink, but the value is sent out in a callback method defined in XML. | ● | ○ | ● |
| ActivityCommunication1 | Contains two activities that communicate using static fields. | ● | ○ | ○ |
| **Lifecycle** | | | | |
| BroadcastReceiver-Lifecycle1 | Calls to sources and sinks distributed across a broadcast receiver lifecycle. | ● | ● | ○ |
| ActivityLifecycle1 | Calls to sources and sinks distributed across an activity lifecycle. | ● | ● | ○ |
| ActivityLifecycle2 | Activity class inherited from a superclass containing the lifecycle method which leaks the tainted value. | ● | ● | ○ |
| ActivityLifecycle3 | Calls to sources and sinks distributed across instance state handling methods. | ● | ● | ○ |
| ActivityLifecycle4 | A tainted value is obtained on onPause() and leaked when the activity is restarted later. | ● | ● | ○ |
| ServiceLifecycle1 | Calls to sources and sinks distributed across a service lifecycle. | ● | ● | ○ |
| **General Java** | | | | |
| Loop1 | Contains a simple loop and a data leak. | ● | ○ | ○ |
| Loop2 | Retrieves location information through a callback and leaks it via nested loops. | ● | ○ | ● |
| SourceCodeSpecific1 | Uses unusual code construct `a = p ? b : c`. | ● | ○ | ○ |
| StaticInitialization1 | Passes a tainted value into a static initialization method. | ● | ○ | ○ |
| UnreachableCode | Passes tainted data into a method that is never called. | ○ | ○ | ○ |
| **Miscellaneous Android-Specific** | | | | |
| PrivateDataLeak1 | Summary test case containing various challenges. | ● | ● | ● |
| PrivateDataLeak2 | Leaks a value from a password field. | ● | ○ | ○ |
| DirectLeak1 | The device id is read out and sent via SMS on the activity's onCreate() event. | ● | ○ | ○ |
| InactiveActivity | Data leak in a disabled activity. | ○ | ○ | ○ |
| LogNoLeak | Writes untainted data into a log file. | ○ | ● | ○ |
| **Implicit Flows** | | | | |
| ImplicitFlow1-4 | Test case for implicit flows. | ● | ○ | ○ |

**Table 5.2:** Detailed description of DroidBench's test cases (Version 1.0), taken from [15]

Table 5.3 presents the analysis results for FlowDroid (and two commercial analysis tools, see Section 5.5) explained in the following. The results show that FlowDroid generally performs quite well. As mentioned before, FlowDroid handles array indices imprecisely. The same limitation applies to *ListAccess1*, causing false positives in the first category. Handling indices precisely and efficiently is an open research question. *Button2* causes a false positive because FlowDroid does not currently support strong updates. In result, it cannot kill taints for certain button combinations. Strong updates would require a must-alias analysis which is hard to achieve inter-procedurally. *IntentSink1* is not detected because the test case contains no actual sink. Instead, the tainted value is stored in an intent which is then handed back to the activity by the framework. Such cases are hard to handle without special treatment. *StaticInitialization1* fails because Soot currently assumes all static initializers to execute at the beginning of the program, which in this case is not correct. As most known taint-analysis tools, FlowDroid currently disregards implicit flows caused through control-flow dependencies.

## 5.3 Evaluation Apps

**InsecureBank** [41] is a vulnerable Android app created by Paladion Inc. specifically for the purpose of evaluating analysis tools such as FlowDroid. It contains various vulnerabilities and data leaks similar to those found in real-world applications. FlowDroid found all seven data leaks which we all verified by hand. There were neither false positives nor false negatives.

We also looked at **Pandemobium** [23], a collection of vulnerable apps for both Android and iOS. We picked the most recent version 0.9.3, which contains only one application, PandemobiumStockTrader. However, the focus is less on the privacy leaks of the Android app, but on SQL injections, cross-side scripting vulnerabilities, unrestricted visibility settings and conceptual issues in the design of the cooperation with the server component. Nevertheless we analyzed the app with

FlowDroid and found 12 leaks. Nearly all of them involving data written to the log, for example a value read from a private file. We also detected that a value from a password field is used in a GET parameter of a URL.

## 5.4 Real-World Applications

The above experiments already gave very promising view of FlowDroid's precision, not just for small test cases but also for more realistic apps such as InsecureBank [41]. To strengthen the external validity of our experiments, we nevertheless performed an additional qualitative analysis on a random selection of popular apps from the Google Play store. While the obtained analysis reports do not indicate any malicious apps in this selection, the majority of apps is reported to—probably accidentally—leak information into logs and preference files. Samsung's Push Service, for instance, logs the phones IMEI. Logs are problematic, as the OS does not impose the same access restrictions on logs as it does on files: in Android versions older than 4.1 all logs are readable by any app that has the READ_LOGS permission. The game Hugo Runner stores longitude and latitude into a preferences file. As we verified by hand, though, those preferences were correctly written in private mode, precluding any access by other apps. This indicates that taint analyses could gain precision by considering auxiliary information such as the write mode mentioned above. For most examined apps FlowDroid terminated in under a minute. The longest-running instance, Samsung's Push Service, took about 4.5 minutes to analyze. However, to achieve a higher product maturity our analysis should be tested on even more applications to become more immune to broken apks and other exceptions.

## 5.5 Comparison with Other Tools

The following section presents a comparison of FlowDroid with both commercial and scientific tools. As it was not the main focus of this master thesis, it is only mentioned briefly. However, more detailed information on the comparison with commercial tools can be found in FlowDroid's technical report [15].

### 5.5.1 Comparison with Commercial Tools

We compared FlowDroid against IBM AppScan Source [28] version 8.7 and Fortify SCA [27] 5.14 by HP on all tests from DroidBench. AppScan Source distinguishes three different categories of findings: It can either find a vulnerability with a complete path from source to sink, a source-to-sink connection which includes unknown semantics in the path (for example a possible sanitization) or a suspicious code fragment. Because FlowDroid does not support sanitization we take the first two categories into account. As Table 5.3 shows, AppScan Source finds only about 39% of all leaks (50% if ignoring implicit flows). Major problems occur with the handling of callbacks, the Android component lifecycle and implicit flows.

Fortify also provides different kinds of findings, such as data flows from sensitive sources to public sinks, requests for security-sensitive permissions, calls to security-sensitive methods, etc. In our evaluation, we only considered findings about data flows. As Table 5.3 shows, Fortify SCA faces problems similar to those of IBM AppScan, like the handling of the Android component lifecycle, callbacks and implicit flows. In particular, Fortify detects 4 out of 6 data leaks for the lifecycle tests, but closer inspection shows that this only happens because the data source involves a static field, which Fortify apparently treats in a special way coincidentally causing a leak to be reported.

We can conclude that AppScan Source and Fortify SCA aim for relatively high precision while sacrificing recall, thus risking to miss actual privacy leaks instead of overburden the user with false positives. In comparison, FlowDroid shows higher precision with a significantly higher recall.

### 5.5.2 Comparison with Scientific Tools

We also tried to compare FlowDroid to a number of other tools from the scientific literature, namely TrustDroid [58], LeakMiner [57] and the tool by Batyuk et al. [6]. Unfortunately none of those tools are available online, and even worse the respective authors did not reply to our inquiries.

We tried to run DroidBench on SCanDroid [16], but faced technical difficulties. The tool did not report any findings at all in our setup. Though being in contact with the authors, we were unable to fix these issues. The authors of AndroidLeaks [18] promised to run their tool on DroidBench but never delivered. We also contacted the authors of CHEX [38], but they were unable to provide the tool or any benchmark results due to intellectual property claimed by NEC. Starostin [39] declined to participate in the experiment as his tool ignores aliasing, making any comparison meaningless.

We also contacted the authors of ScanDal [33] who agreed to evaluate their tool with DroidBench. With these new benchmarks they were able to strengthen their analyzer and send us their optimized results. Like FlowDroid, they are not

capable of implicit flows and missed leaks in the test cases *ImplicitFlow 1-4*. They also do not consider inter-app communication and user inputs, leading to false negatives in *intentSink1*, *intentSink2*, *PrivateDataLeak1* and *PrivateDataLeak2*. Similar to our approach, ScanDal cannot handle static initialization correctly causing the *StaticInitialization1* test case to fail. They also abstract from concrete array positions and make use of weak updates on memory references, so their approach finds false positives in *ArrayAccess1*, *ArrayAccess2* and *ObjectSensitivity2*. We send them a preliminary version of DroidBench lacking some of the test cases, for example *Button2*, which also requires strong updates on memory locations. If we ignore these minor differences we can conclude that ScanDal and FlowDroid operate on a similar level of precision. Additionally, the results show that our benchmark suite helps to improve the quality of the evaluated analysis, but it needs to be extended by further diverse challenges to highlight the different strengths of the approaches.

In result, we only managed to compare our approach to one single scientific taint-analysis tool for Android. We believe that the lack of comparative experiments is hindering scientific progress a lot, which is why we make available our entire implementation, documentation and benchmarks as open source.

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

| App Name | AppScan Source | Fortify SCA | FlowDroid |
|---|---|---|---|
| **Arrays and Lists** | | | |
| ArrayAccess1 | | | ⋆ |
| ArrayAccess2 | ⋆ | ⋆ | ⋆ |
| ListAccess1 | ⋆ | ⋆ | ⋆ |
| **Callbacks** | | | |
| AnonymousClass1 | ○ | ⊛ | ⊛ |
| Button1 | ○ | ⊛ | ⊛ |
| Button2 | ⊛ ○ ○ | ⊛ ○ ○ | ⊛ ⊛ ⊛ ⋆ |
| LocationLeak1 | ○ ○ | ○ ○ | ⊛ ⊛ |
| LocationLeak2 | ○ ○ | ○ ○ | ⊛ ⊛ |
| MethodOverride1 | ⊛ | ⊛ | ⊛ |
| **Field and Object Sensitivity** | | | |
| FieldSensitivity1 | | | |
| FieldSensitivity2 | | | |
| FieldSensitivity3 | ⊛ | ⊛ | ⊛ |
| FieldSensitivity4 | ⋆ | | |
| InheritedObjects1 | ⊛ | ⊛ | ⊛ |
| ObjectSensitivity1 | | | |
| ObjectSensitivity2 | ⋆ | | |
| **Inter-App Communication** | | | |
| IntentSink1 | ⊛ | ⊛ | ○ |
| IntentSink2 | ⊛ | ⊛ | ⊛ |
| ActivityCommunication1 | ⊛ | ⊛ | ⊛ |
| **Lifecycle** | | | |
| BroadcastReceiverLifecycle1 | ⊛ | ⊛ | ⊛ |
| ActivityLifecycle1 | ⊛ | ⊛ | ⊛ |
| ActivityLifecycle2 | ○ | ⊛ | ⊛ |
| ActivityLifecycle3 | ○ | ○ | ⊛ |
| ActivityLifecycle4 | ○ | ⊛ | ⊛ |
| ServiceLifecycle1 | ○ | ○ | ⊛ |
| **General Java** | | | |
| Loop1 | ⊛ | ○ | ⊛ |
| Loop2 | ⊛ | ○ | ⊛ |
| SourceCodeSpecific1 | ⊛ | ⊛ | ⊛ |
| StaticInitialization1 | ○ | ⊛ | ○ |
| UnreachableCode | | ⋆ | |
| **Miscellaneous Android-Specific** | | | |
| PrivateDataLeak1 | ○ | ○ | ⊛ |
| PrivateDataLeak2 | ⊛ | ⊛ | ⊛ |
| DirectLeak1 | ⊛ | ⊛ | ⊛ |
| InactiveActivity | ⋆ | ⋆ | |
| LogNoLeak | | | |
| **Implicit Flows** | | | |
| ImplicitFlow1 | ○ ○ | ○ ○ | ○ ○ |
| ImplicitFlow2 | ○ ○ | ○ ○ | ○ ○ |
| ImplicitFlow3 | ○ ○ | ○ ○ | ○ ○ |
| ImplicitFlow4 | ○ ○ | ○ ○ | ○ ○ |
| **Sum, Precision and Recall — including implicit flows** | | | |
| ⊛, higher is better | 14 | 17 | 26 |
| ⋆, lower is better | 5 | 4 | 4 |
| ○, lower is better | 22 | 19 | 10 |
| Precision $p = ⊛/(⊛ + ⋆)$ | 74% | 81% | 86% |
| Recall $r = ⊛/(⊛ + ○)$ | 39% | 47% | 72% |
| F-measure $2pr/(p+r)$ | 0.51 | 0.60 | 0.78 |
| **Sum, Precision and Recall — excluding implicit flows** | | | |
| ⊛, higher is better | 14 | 17 | 26 |
| ⋆, lower is better | 5 | 4 | 4 |
| ○, lower is better | 14 | 11 | 2 |
| Precision $p = ⊛/(⊛ + ⋆)$ | 74% | 81% | 86% |
| Recall $r = ⊛/(⊛ + ○)$ | 50% | 61% | 93% |
| F-measure $2pr/(p+r)$ | 0.60 | 0.70 | 0.89 |

**Table 5.3:** DroidBench test results, taken from [15]

# 6 Related Work

The following chapter lists previous and related work in the fields of information and data flow analysis in general (Section 6.1) and for Android in particular (Section 6.2). When applicable, we compare it to our approach.

## 6.1 General

Since the first approaches to flow analysis are originated in 1973 [32] a lot of research was accomplished. In this section we concentrate on the most closely related and influential works.
Genaim et al. [17] claim that they have implemented the first information flow analysis for Java byte code. It is implemented with the static analyzer Julia [31]. The taint information which is propagated consists of a single boolean value which is very lightweight, but not sufficient for all fields of applications (see Section 4.4.7 for a description of FlowDroid's Abstraction class). The control-flow graph consists of basic blocks which aggregate byte code instructions. Their approach is able to detect implicit flows in loops and exceptions while preserving flow- and context-sensitivity. Because all fields are treated as static class variables, it is not field-sensitive, but field-based.

JOANA (Java Object-sensitive ANAlysis) [22] is an information flow analysis which tracks taint in Java applications. In contrast to FlowDroid it is able to perform information flow analysis by considering indirect flows. Like our approach it is flow-, field-, object- and context-sensitive. However, JOANA does not support Android applications yet, although they state that they are currently working on an Android integration [22].

The static taint analysis tool TAJ (Taint Analysis for Java) [52] is implemented in WALA [54] and focuses on web applications. As it is part of a commercial product it possesses a certain degree of maturity: For instance, it scales to large applications by using a priority-driven call-graph construction which provides intermediate partial results based on a priority function. Additionally it utilizes a mechanism similar to FlowDroid's Taintwrapper (see Section 4.4.4) to gain performance. The authors utilize a flow- and context-sensitive taint propagation for variables which is combined with a flow-insensitive propagation for heap contents. Reflection is supported if it can be statically inferred (for example if the target is a constant value). Similarly, TAJ also tries to derive the index of containers to distinguish the individual elements. Tripp et al. specifically adapted the tool to analyze Java EE applications; hence, it is able to handle Java beans and frameworks configured by XML files. Further optimizations of the run time include the parsing of the artifacts that are used as source to generate code instead of analyzing the generated code. As FlowDroid is focused on Android it does not have to deal with code generation and sanitizers, at least few possible applications for sanitizers in Android are imaginable.

Andromeda [51], another tool from Tripp et al. used in a commercial product, is also a static taint analysis for web applications. Because alias analysis and even (partial) call-graph generation are invoked on demand, it is very scalable. It utilizes Framework For Frameworks (F4F) [49], a taint analysis specifically designed for frameworks like Apache Struts or Spring. Additionally Andromeda is capable of performing incremental analyses on updated web applications. As already explained in Section 3.1.2 it introduces a backward analysis to find aliases to heap elements in a context-sensitive analysis, which we enhanced in FlowDroid.

Although we build upon Andromeda's approach to alias analysis, there are also other concepts for tracking heap elements: ACTARUS [24] uses a heap graph to track references in JavaScript. It compares an access path that is stored on the heap with all access paths which are based on variables known in the same context and have a non-empty intersection of their points-to sets. In the beginning we followed a similar approach in FlowDroid, but discovered soon that our points-to analysis was not precise enough for our purpose.

Hammer et al. [25] present a flow-, context- and object-sensitive information flow analysis for Java applications based on program dependence graphs. Beside precision and performance, usability and flexible adaption to different possible applications might be also desirable for information-flow analysis. In contrast to FlowDroid their analysis allows to define different levels of security for data sources. This might lead to more accurate results, but also requires more effort to tag the sources and sinks in advance. However, we decided to stick to a more simple approach and distinguish only tainted from untainted data. For every leak we store source and sink, so theoretically we are able to decide (possibly based on the user's preferences) on the criticality of a privacy violation. See Chapter 7 for more details.

## 6.2 Android

There are several approaches to static analysis of Android applications differing in precision, runtime, scope and focus. Due to Android's dominant market position and extraordinary growth rate, this relatively new field of research seems to create major interest.

Payet et al. [42] try to help developers by checking for common programming errors using different static analyses implemented with Julia, for example a nullness and dead code analysis. In contrast to FlowDroid, their approach cannot perform a data flow analysis and is not focused on security.

LeakMiner [57] appears similar to our approach from a technical point of view: like FlowDroid, it is based on Soot, uses Spark for call-graph generation and implements the Android lifecycle. The paper states that an app can be analyzed in 2.5 minutes on average. However, the analysis is not context-sensitive which is likely a major source of imprecision. The set of sources and sinks contains some well-known entries but is far from complete. According to their evaluation, their precision is roughly 50%, while their recall is not assessable as their test set of Android applications from the market is not annotated. Unfortunately we were unable to perform a systematic comparison using DroidBench as the authors did not respond to our inquiries.

AndroidLeaks [18] also states the ability to handle the Android lifecycle including callback methods. It is based on WALA's context-sensitive System Dependence Graph with a context-insensitive overlay for heap tracking. It is not as precise as FlowDroid, because it taints the whole object if tainted data is stored in one of its fields. For the determination of sources and sinks a permission mapping is created between Android API calls and the permissions they require. The authors manually selected a subset for sources respectively sinks.

SCanDroid [16] is another tool for reasoning about data flows in Android applications. Its main focus is the inter-component (e.g. between two activities in the same app) and inter-app data flow. This poses the challenge of connecting intent senders to their respective receivers in other applications. SCanDroid prunes all call edges to Android OS methods and conservatively assumes the base object, the parameters, and the return value to inherit taints from arguments which is much less precise than FlowDroid's treatment. FlowDroid, on the other hand, currently does not resolve intent-based communication. Such a feature would require the resolving of URIs and string analysis, which we leave to future work. At the moment, SCanDroid cannot handle .apk files as it requires Java source code or JVML bytecode.

There are also approaches with a much broader focus like Batyuk et al. [6]. It does not only track data flows, but also generate a user-friendly report and automatically sanitizes the application binaries of malicious apps by replacing their sources with safe equivalents. For example, their approach would replace the method which fetches the IMEI by a universally unique identifier generator. The authors state that their so-called detectors perform static analysis with the help of regular expressions and statistic approaches, but give neither details on their implementation in Python nor on the precision of the approach. Hence, we were unable to conduct a detailed comparison to FlowDroid.

One of the most sophisticated approaches is CHEX [38], a flow- and context-sensitive analysis to detect hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources or sinks. Similar to FlowDroid, CHEX builds searches for all entry points of Android Applications, but it does not rely on a precise model of the Android lifecycle to determine the order of these entry points. Since CHEX looks for vulnerabilities, target applications are supposed to be benign. The derived assumptions (no obfuscation etc.) do not hold for malicious apps. Additionally their main focus is not to analyze only the flow of permission-protected data but data that is meant to be used only internally as well as recognizing unwanted modification of internal data. Furthermore, CHEX does not analyze calls into the Android framework itself but instead requires a (hopefully complete) model of the framework. In FlowDroid such a model is optional and, except for native calls, is used only to increase performance.

Other approaches like CopperDroid [44] dynamically observe interactions between the Android components and the underlying Linux system to reconstruct higher-level behavior. Special stimulation techniques are used to find malicious activities. Attackers, however, can easily modify an app to detect whether it is running inside a virtual machine and then leak no data during that time. Alternatively, data leaks might only occur after a certain runtime threshold. Aurasium [55] and DroidScope [56] largely suffer from the same shortcomings with respect to static leak detection.

TrustDroid [58] is a hybrid approach which can both operate on the phone and on a server. It takes the Android byte code and converts it into a textual description using Jasmin syntax. Similar to FlowDroid, native code from the system library is analyzed in advance and precomputed taint propagation rules are used. To overcome performance issues when running on a device with limited computational power and battery, the authors implemented various selectable levels of granularity. Unfortunately they did not provide more information how these different settings perform on benchmarks or real applications.

ScanDal [33] is a static analyzer which also converts the Dalvik VM byte code in an intermediate language defined by the authors. Similar to our approach, their solution is path-insensitive, but has other imprecisions in terms of being only 1-context-sensitive and not fully flow-sensitive. Their initialization phase respects the order, but for events they assume a random order which is less precise than a sophisticated lifecycle model. Furthermore, the sources and sinks are limited to a relatively small fixed set. Unfortunately the evaluation is not so meaningful, because a lot of privacy leaks found by the analysis are classified neither as true positive nor as false positive. These "unknown" leaks could not be reproduced through manual inspection. A benchmark suite which describes all existing leaks like DroidBench could help to provide a better evaluation (indeed they used our benchmark suite to improve ScanDal, see Section 5.5.2 for details). The authors claim there false positives are mainly caused by their conservative wrapping of the calls to Android framework methods.

FlowDroid avoids this problem by either providing taint wrapping to improve performance or analyzing those methods as well.

Mann et al. [39] propose a security type system to detect privacy leaks in Android apps. It operates on a modified set of Dalvik instructions. We are assuming that their approach is not flow- and context-sensitive as this is not inherently included in type systems. Unfortunately no information about this is provided in the paper. While security type systems are generally capable of handling implicit flows, their current implementation is not able to recognize them. As this approach is very different from FlowDroid, it would be interesting to compare them in terms of precision and performance, particularly because their evaluation consists only of self-developed test applications.

# 7 Future Work

We have explained limitations of the implementation in Section 4.5. Some of them are inherently given in a static analysis and can only be overcome by the combination with a dynamic analysis, in particular the resolving of non-constant strings, e.g. used for indices in collection accesses and as target description in reflection. Our implementation can be extended to cover simple cases, for instance the use of constant strings. A must-alias analysis would help to provide strong updates on fields and memory locations. Additionally, the analysis could be improved to be even more precise by considering implicit flows. Because they cannot be detected at the moment, the corresponding SecuriBench Micro Tests (see Section 5.1) are failing. Liu et al. [36] suggest to introduce additional edges to track this information. This would require not only changes in the analysis itself but in the Heros framework as well.

Taint analysis for Android can be improved by implementing inter-component analysis: As explained in Section 4.4.1 we included a very basic support for sending tainted data with intents by treating them as sink, which is reasonable as we only look at single components of an app and cannot track the taint any further. In general it might be useful to know which sensitive data can be leaked in an app or even in a whole environment, for example on a smart phone that should be used for secure communication and contains a specific fixed set of applications (Note that according to Grace et al. [21] even stock applications can help to leak data). The analysis could be extended to support both intra-app and inter-app communication by treating the sink of one component as a source of another component. This includes intent communication, but also usage of shared resources like the internal phone storage. There are already approaches which specialize on inter-app communication, for example ComDroid [10].

For web applications, sanitizers are very important as they are an effective counteraction against frequent attacks such as SQL injection. Thus, web application analysis must consider them as they disrupt source-to-sink connections. We believe that for a taint analysis specialized on Android applications, sanitizers are of minor importance, because we do not know of any application yet which sanitizes data so it does not contain sensitive information any more. However, if sophisticated support for web applications should be added, sanitizers have to be supported.

Aside from the analysis itself, reporting the results to the user is still an open issue: Currently source-to-sink connections, optionally with all intermediate tainted variables, are printed to the console. A more usable report would offer a practical output, for instance an XML structure accompanied by a stylesheet which allows to parse the results for further processing as well as present them in a readable fashion to human users. Unfortunately, even the best representation requires manual revision: The analysis collects all data leaks, but some of them are certainly necessary to ensure the application functionality. For example a service which shows your friends nearby has to send your location to a server in order to compare the location with the ones of your friends and make it accessible to them. The analysis, however, cannot distinguish this flow from a malicious one, for instance location data which is sent to an advertising server. This differentiation can be performed by subsequent analyses which have more information about the analyzed applications - possibly in conjunction with a machine learning approach.

A preceding selection of sources and sinks by the user could improve performance. As the Android sources and sinks are annotated with categories (see Section 4.4.1) one could limit the analysis to specific data flows, for instance location information which are sent via network.

## 8 Conclusions

In this master thesis we have presented FlowDroid, a highly precise static taint-analysis for Android applications in particular and Java programs in general. We explained the theoretical background it is based on, described the design and gave an overview of the implementation. We have shown that many existing approaches either do not adequately model Android-specific challenges like the application lifecycle or are imprecise, leading to either missed leaks or false positives.

Furthermore we have seen that until now, there was no established benchmark suite for static analyses on Android. Hopefully this will change with DroidBench, a collection of test applications, which we made publicly available. We used it for comparing FlowDroid to two commercial tools, showing that besides finding more real leaks, FlowDroid also has a higher precision resulting in less false positives. We tried to compare it to other approaches, but although there are many static analyses, only very few of them are open source. We believe that the research community would benefit from more transparency, both by making their scientific results available to the public and by using more reproducible benchmarks than a more or less randomly picked amount of apps from the market.

We also emphasized the current limitations of our approach which overlap with current research challenges in static analysis and gave hints to further improve the analysis.

## 9 Acknowledgements

I would like to express my gratitude to my supervisor Eric Bodden for mentoring and very helpful advices and discussions, in particular about the conceptional design and FlowDroid's implementation.

I thank everyone who participated in the development of FlowDroid, especially Steven Arzt, who took responsibility for the implementation of the Android parsing and analyzing component and fixed not only our own bugs but the ones of the used frameworks as well. I want to thank Siegfried Rasthofer, who brought his extensive knowledge of the Android operating system to the creation of DroidBench (along with the ones already mentioned). Overall I really enjoyed working in a motivated team like the Secure Software Engineering group.

Also I am grateful to the authors of FlowDroid's technical report, which this master thesis bases upon. Besides the ones already mentioned, these are Alexandre Bartel, Jacques Klein and Yves le Traon from the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, and Damien Octeau and Patrick McDaniel from the Department of Computer Science and Engineering, Penn State University. I want to thank Stephan Huber from Fraunhofer SIT and Dr. Karsten Sohr from TZi Bremen who assisted us with the evaluation of FlowDroid and all active participants of the Soot mailing list, which helped me a lot resolving common beginner's mistakes.

Furthermore I would like to thank everybody who supported me in any other way.

## Bibliography


[1] Android. Android security overview. `http://source.android.com/tech/security/`, retrieved 2013-06-23.

[2] Signing your applications. `http://developer.android.com/tools/publishing/app-signing.html`, retrieved 2013-06-23.

[3] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks. Technical report, EC SPRIDE Technical Report TUD-CS-2013-0114. http://www.bodden.de/pubs/TUD-CS-2013-0114.pdf, 2013.

[4] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: an application to Android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, New York, NY, USA, 2012. ACM.

[5] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[6] L. Batyuk, M. Herpich, S.A. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66 –72, oct. 2011.

[7] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 3–8, New York, NY, USA, 2012. ACM.

[8] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: International Conference on Software Engineering*, pages 241–250. ACM, May 2011.

[9] B. Chess and Gary McGraw. Static analysis for security. *Security & Privacy, IEEE*, 2(6):76–79, 2004.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[11] International Data Corporation. Worldwide Quarterly Mobile Phone Tracker 1Q13. `http://www.idc.com/getdoc.jsp?containerId=prUS24108913`, retrieved 2013-05-29.

[12] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[13] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *IN WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[15] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly Precise Taint Analysis for Android Application. Technical report, EC SPRIDE Technical Report TUD-CS-2013-0113. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf, 2013.

[16] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, Department of Computer Science, University of Maryland, College Park, November 2009.

[17] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 346–362, Berlin, Heidelberg, 2005. Springer-Verlag.

[18] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Dennis Giffhorn and Gregor Snelting. Probabilistic noninterference based on program dependence graphs. Technical Report 06/2012, KIT, Faculty of Informatics, June 2012. revised 2013, submitted for publication.

[20] Google Inc. Permissions. `http://developer.android.com/guide/topics/security/permissions.html`, retrieved 2013-06-23.

[21] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[22] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2013.

[23] Denim Group. Pandemobium. `https://code.google.com/p/pandemobium/`, retrieved 2013-06-23.

[24] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187, New York, NY, USA, 2011. ACM.

[25] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, October 2009.

[26] Heros. `http://sable.github.io/heros/`, retrieved 2013-06-23.

[27] L.P. Hewlett-Packard Development Company. Fortify 360 Source Code Analyzer (SCA). `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1214365#.UW6CVKuAtfQ`, retrieved 2013-06-23.

[28] IBM Rational AppScan. `http://www.ibm.com/software/products/us/en/appscan`, retrieved 2013-06-23.

[29] Google Inc. Android and security. `http://googlemobile.blogspot.de/2012/02/android-and-security.html`, retrieved 2013-06-23.

[30] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, PLAS '06, pages 27–36, New York, NY, USA, 2006. ACM.

[31] Julia. `http://www.juliasoft.com/products`, retrieved 2013-06-23.

[32] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[33] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. SCANDAL: Static analyzer for detecting privacy leaks in android applications. In *Proceedings of the Workshop on Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2012.

[34] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oktober 2011.

[35] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[36] Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Ben Livshits. Stanford SecuriBench Micro. `http://suif.stanford.edu/~livshits/work/securibench-micro/`, retrieved 2013-06-23.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[39] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1457–1462, New York, NY, USA, 2012. ACM.

[40] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[41] Paladion. Insecurebank test app. `http://www.paladion.net/downloadapp.html`, retrieved 2013-06-23.

[42] Étienne Payet and Fausto Spoto. Static analysis of Android programs. In *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, pages 439–445, Berlin, Heidelberg, 2011. Springer-Verlag.

[43] Nicholas J Percoco and Sean Schulte. Adventures in bouncerland. *Blackhat USA*, 2012.

[44] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *EUROSEC*, Prague, Czech Republic, April 2013.

[45] 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*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[46] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic Detection of Inter-application Permission Leaks in Android Applications. Technical report, Technical Report TR13-02, Department of Computer Science, Rice University, January 2013.

[47] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A Comprehensive Security Assessment. *Security Privacy, IEEE*, 8(2):35–44, 2010.

[48] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[49] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 1053–1068, New York, NY, USA, 2011. ACM.

[50] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *SIGPLAN Not.*, 35(10):264–280, October 2000.

[51] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.

[52] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[53] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations, 1998.

[54] T. J.Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php/Main_Page`, retrieved 2013-06-23.

[55] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for Android applications. In *USENIX Security 2012*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[56] Lok Kwong Yan and Heng Yin. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *USENIX Security 2012*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[57] Zhemin Yang and Min Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104, 2012.

[58] Zhibo Zhao and F.C.C. Osono. TrustDroid: Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 135–143, 2012.

[59] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

# A Appendix

## A.1 Testresults SecuriBench

Table A.1 presents the detailed results of our evaluation with the SecuriBench Micro benchmark suite. The first column contains the name of the testcase. The second column, **TP**, comprises the true positives, at which the first number is the count of the leaks FlowDroid found and the second number is the count of the actual existing leaks. The third column, **FP** represents the false positives. The subsequent column describes the type of the test, where "p" stands for a positive test which contains leaks and "n" represents a negative test without any leaks. Where necessary, we give a short comment on the test case or the test result in the last column.

| Testcase | TP | FP | Type | Comment |
|---|---|---|---|---|
| Aliasing1 | 1/1 | 0 | p | |
| Aliasing2 | 0/0 | 0 | n | |
| Aliasing3 | 0/0 | 0 | n | test case was incorrectly tagged |
| Aliasing4 | 2/2 | 0 | p | |
| Aliasing5 | 1/1 | 0 | p | |
| Aliasing6 | 7/7 | 0 | p | |
| Arrays1 | 1/1 | 0 | p | |
| Arrays2 | 1/1 | 2 | p | We do not distinguish array elements |
| Arrays3 | 1/1 | 0 | p | |
| Arrays4 | 1/1 | 0 | p | |
| Arrays5 | 0/0 | 1 | p | We do not distinguish array elements |
| Arrays6 | 1/1 | 0 | p | |
| Arrays7 | 1/1 | 0 | p | |
| Arrays8 | 1/1 | 1 | p | We do not distinguish array elements |
| Arrays9 | 1/1 | 0 | p | |
| Arrays10 | 1/1 | 1 | p | We do not distinguish array elements |
| Basic1 | 1/1 | 0 | p | |
| Basic2 | 1/1 | 0 | p | |
| Basic3 | 1/1 | 0 | p | |
| Basic4 | 1/1 | 0 | p | |
| Basic5 | 3/3 | 0 | p | |
| Basic6 | 1/1 | 0 | p | |
| Basic7 | 1/1 | 0 | p | |
| Basic8 | 1/1 | 0 | p | |
| Basic9 | 1/1 | 0 | p | |
| Basic10 | 1/1 | 0 | p | |
| Basic11 | 2/2 | 0 | p | |
| Basic12 | 2/2 | 0 | p | |
| Basic13 | 1/1 | 0 | p | |
| Basic14 | 1/1 | 0 | p | required modification of BasicTestCase class |
| Basic15 | 1/1 | 0 | p | |
| Basic16 | 1/1 | 0 | p | |
| Basic17 | 1/1 | 0 | p | |
| Basic18 | 1/1 | 0 | p | |
| Basic19 | 1/1 | 0 | p | |
| Basic20 | 1/1 | 0 | p | |
| Basic21 | 4/4 | 0 | p | |
| Basic22 | 1/1 | 0 | p | |
| Basic23 | 3/3 | 0 | p | |
| Basic24 | 1/1 | 0 | p | |
| Basic25 | 1/1 | 0 | p | |
| Basic26 | 1/1 | 0 | p | |
| Basic27 | 1/1 | 0 | p | |

| Testcase | TP | FP | Type | Comment |
|---|---|---|---|---|
| Basic28 | 2/2 | 0 | p | |
| Basic29 | 2/2 | 0 | p | |
| Basic30 | 1/1 | 0 | p | |
| Basic31 | 3/3 | 0 | p | |
| Basic32 | 1/1 | 0 | p | |
| Basic33 | 1/1 | 0 | p | |
| Basic34 | 2/2 | 0 | p | |
| Basic35 | 6/6 | 0 | p | |
| Basic36 | 1/1 | 0 | p | |
| Basic37 | 1/1 | 0 | p | |
| Basic38 | 1/1 | 0 | p | |
| Basic39 | 1/1 | 0 | p | |
| Basic40 | 1/1 | 0 | p | |
| Basic41 | 1/1 | 0 | p | required modification of BasicTestCase class |
| Basic42 | 1/1 | 0 | p | required modification of BasicTestCase class |
| Collections1 | 1/1 | 0 | p | |
| Collections2 | 1/1 | 0 | p | |
| Collections3 | 2/2 | 0 | p | |
| Collections4 | 1/1 | 0 | p | |
| Collections5 | 1/1 | 0 | p | |
| Collections6 | 1/1 | 1 | p | We do not distinguish collection elements |
| Collections7 | 1/1 | 1 | p | We do not distinguish collection elements |
| Collections8 | 1/1 | 0 | p | |
| Collections9 | 0/0 | 0 | n | test case was inconsistently tagged |
| Collections10 | 1/1 | 0 | p | test case was inconsistently tagged |
| Collections11 | 1/1 | 0 | p | |
| Collections12 | 1/1 | 0 | p | |
| Collections13 | 1/1 | 1 | p | We do not distinguish collection elements |
| Collections14 | 1/1 | 0 | p | |
| Datastructures1 | 1/1 | 0 | p | We fixed the SecuriBench Code, obviously the wrong method was called |
| Datastructures2 | 1/1 | 0 | p | |
| Datastructures3 | 1/1 | 0 | p | |
| Datastructures4 | 0/0 | 0 | n | test case was incorrectly tagged |
| Datastructures5 | 1/1 | 0 | p | |
| Datastructures6 | 1/1 | 0 | p | |
| Factories1 | 1/1 | 0 | p | |
| Factories2 | 1/1 | 0 | p | |
| Factories3 | 1/1 | 0 | p | |
| Inter1 | 1/1 | 0 | p | |
| Inter2 | 2/2 | 0 | p | |
| Inter3 | 1/1 | 0 | p | |
| Inter4 | 1/1 | 0 | p | test case was inconsistently tagged |
| Inter5 | 1/1 | 0 | p | test case was inconsistently tagged |
| Inter6 | 0/1 | 0 | p | correct static initialization required |
| Inter7 | 1/1 | 0 | p | |
| Inter8 | 1/1 | 0 | p | |
| Inter9 | 2/2 | 0 | p | test case was inconsistently tagged |
| Inter10 | 1/1 | 0 | p | test case was inconsistently tagged |
| Inter11 | 1/1 | 0 | p | |
| Inter12 | 0/1 | 0 | p | |
| Inter13 | 1/1 | 0 | p | |
| Inter14 | 1/1 | 0 | p | |
| Pred1 | | | n | predicates are out of scope |
| Pred2 | | | p | predicates are out of scope |
| Pred3 | | | n | predicates are out of scope |
| Pred4 | | | p | predicates are out of scope |

| Testcase | TP | FP | Type | Comment |
|---|---|---|---|---|
| Pred5 | | | p | predicates are out of scope |
| Pred6 | | | n | predicates are out of scope |
| Pred7 | | | n | predicates are out of scope |
| Pred8 | | | p | predicates are out of scope |
| Pred9 | | | p | predicates are out of scope |
| Refl1 | | | p | reflection is out of scope |
| Refl2 | | | p | reflection is out of scope |
| Refl3 | | | p | reflection is out of scope |
| Refl4 | | | p | reflection is out of scope |
| Sanitizer1 | | | p | sanitizers are out of scope |
| Sanitizer2 | | | n | sanitizers are out of scope |
| Sanitizer3 | | | n | sanitizers are out of scope |
| Sanitizer4 | | | p | sanitizers are out of scope |
| Sanitizer5 | | | p | sanitizers are out of scope |
| Sanitizer6 | | | n | sanitizers are out of scope |
| Session1 | 1/1 | 0 | p | |
| Session2 | 1/1 | 1 | p | We do not distinguish collection elements |
| Session3 | 1/1 | 0 | p | |
| StrongUpdates1 | 0/0 | 0 | n | |
| StrongUpdates2 | 0/0 | 0 | n | |
| StrongUpdates3 | 0/0 | 0 | n | |
| StrongUpdates4 | 0/1 | 0 | n | threads are out of scope |
| StrongUpdates5 | 0/0 | 0 | n | |

**Table A.1:** Detailed test results for SecuriBench Micro

## A.2 List of Internal Testcases

Table A.2 shows the different categories of internal tests. Particularly we provide test for Java classes which implement the `java.util.Collection` interface.

| TestcaseClass | Description | Number of Testcases |
|---|---|---|
| ArrayTests | check taint propagation in arrays, for instance the propagation for copied arrays and arrays converted to lists. | 9 |
| CallbackTests | aim to produce a setting similar to the one that occurs when callback methods, for instance from the LocationListener are executed. | 3 |
| ConstantTests | ensure proper taint propagation for constant values. | 4 |
| EasyWrapperListTests | test functionality of Taintwrapper. Additionally all tests can be executed with Taintwrapper by setting the debug flag in `soot.jimple.infoflow.test.junit.JUnitTests` to true | 15 |
| FutureTests | contain test for functionality which is currently not supported by FlowDroid. | 4 |
| HeapTests | test aliasing of heap references | 12 |
| HierarchyTests | cover taint propagation of fields with classes from a class hierarchy | 4 |
| HTTPTests | test taint propagation within `java.net.URL` and `java.net.HttpURLConnection` classes | 2 |
| InFunctionTests | contain tests for Taintwrapper and parameters as sources and sinks | 5 |
| InheritanceTests | cover taint propagation within inherited classes | 4 |
| LengthTests | test taint propagation of fields. A certain length of the access paths is required to distinguish the tainted fields. | 3 |

| TestcaseClass | Description | Number of Testcases |
|---|---|---|
| ListTests | check taint propagation in all sorts of lists, for example LinkedLists, ArrayLists and Stacks. | 16 |
| MapTests | contain test cases for taint propagation in Maps. | 9 |
| MethodRepresentationTests | check the conversion of Soot's String representation into our internal data format. | 1 |
| MultiTest | contain various tests with more than one source, conditional statements, loops and java-internal functions on tainted objects | 9 |
| OperationSemanticTest | require the taint tracking engine to correctly evaluate the semantics of primitive operations. | 1 |
| OtherTests | contain several tests that cannot be assigned to the other categories - often they are added due to findings in real-world applications, including negative tests and tests for lifecycle handling | 20 |
| OverwriteTests | test the overwrite behavior of tainted variables, fields and static variables | 9 |
| QueueTests | test taint propagation in queues | 2 |
| SetTests | test taint propagation in sets | 9 |
| StaticTests | contain tests which check taint propagation for static variables | 2 |
| StringTests | covers taint propagation for Strings, String functions such as `concat`, `toUpperCase()` or `substring`, but also StringBuilder and concatenation via + operator | 26 |
| VectorTests | test taint propagation in vectors | 6 |

**Table A.2:** List of different test categories for FlowDroid's internal test cases