FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden
EC SPRIDE
Technische Universität Darmstadt
firstName.lastName@ec-spride.de

Alexandre Bartel, Jacques Klein, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
firstName.lastName@uni.lu

Damien Octeau, Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
{octeau,mcdaniel}@cse.psu.edu

Abstract

Today's smartphones are a ubiquitous source of private and confidential data. At the same time, smartphone users are plagued by carelessly programmed apps that leak important data by accident, and by malicious apps that exploit their given privileges to copy such data intentionally. While existing static taint-analysis approaches have the potential of detecting such data leaks ahead of time, all approaches for Android use a number of coarse-grain approximations that can yield high numbers of missed leaks and false alarms.

In this work we thus present FLOWDROID, a novel and highly precise static taint analysis for Android applications. A precise model of Android's lifecycle allows the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field and object-sensitivity allows the analysis to reduce the number of false alarms. Novel on-demand algorithms help FLOWDROID maintain high efficiency and precision at the same time.

We also propose DROIDBENCH, an open test suite for evaluating the effectiveness and accuracy of taint-analysis tools specifically for Android apps. As we show through a set of experiments using SecuriBench Micro, DROIDBENCH, and a set of well-known Android test applications, FLOWDROID finds a very high fraction of data leaks while keeping the rate of false positives low. On DROIDBENCH, FLOWDROID achieves 93% recall and 86% precision, greatly outperforming the commercial tools IBM AppScan Source and Fortify SCA. FLOWDROID successfully finds leaks in a subset of 500 apps from Google Play and about 1,000 malware apps from the VirusShare project.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.4.6 [*Security and Protection*]: Information flow controls

PLDI '14, June 9-11 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594299

1. Introduction

According to a recent study [9], Android has seen a constantly growing market share in the mobile phone market, which is now at 81%. With Android phones being ubiquitous, they become a worthwhile target for attacks on users' privacy-sensitive data. Felt et al. classified different kinds of Android malware [12] and found that one of the main threats posed by malicious Android applications are privacy violations which leak sensitive information such as location information, contact data, pictures, SMS messages, etc. to the attacker. But even applications that are not malicious and were carefully programmed may suffer from such leaks, for instance when they contain advertisement libraries [16]. Many app developers include such libraries to obtain some remuneration for their efforts, but few of them fully understand their privacy implications, nor are they able to fully control which data these libraries process. Common libraries distill private information that identifies a person for targeted advertisement such as unique identifiers (e.g., IMEI, MAC-address, etc.), country or location information.

Taint analyses address this problem by analyzing applications and presenting potentially malicious data flows to human analysts or to automated malware-detection tools which can then decide whether a leak actually constitutes a policy violation. These approaches track sensitive "tainted" information through the application by starting at a pre-defined source (e.g. an API method returning location information) and then following the data flow until it reaches a given sink (e.g. a method writing the information to a socket), giving precise information about which data may be leaked where. The analyses can inspect the app both dynamically and statically. Dynamic program analyses, though, require many test runs to reach appropriate code coverage. Moreover, current malware can recognize dynamic monitors as the analyzed app executes, causing the app to pose as a benign program in these situations.

While static code analyses do not share these problems, they run the risk of being imprecise, as they need to abstract from program inputs and to approximate runtime objects. The precise modeling of the runtime execution is particularly challenging for Android apps, as those apps are no stand-alone applications but are actually plugins into the Android framework. Apps consist of different components with a distinct lifecycle. During an app's execution, the framework calls different callbacks within the app, notifying it of system events, which can start/pause/resume/shutdown the app etc. [17]. To be able to effectively predict the app's control flow, static analyses must not only model this lifecycle, but must also integrate further callbacks for system-event handling (e.g., for phone sensors like GPS), UI interaction, and others. As we show in

this work, recognizing callbacks is anything but trivial and requires dedicated algorithms. Another challenge is posed by sources of sensitive information such as password fields in the user interface. The respective API calls returning their contents cannot be detected based on the program code alone. Instead, their detection requires a model of auxiliary information stored in the manifest and layout XML files. Last but not least, like any application written in Java, Android apps also contain aliasing and virtual dispatch constructs. Typical static analyses for Java handle these problems through some degree of context and object sensitivity. The framework nature of Android makes this problem harder than usual, as we found it to expose extraordinarily deep aliasing relationships.

Past data-flow analysis approaches for Android [14, 15, 24, 40] handle the above challenges in an unsatisfactory manner using coarse-grained over- as well as under-approximations. Under-approximations, usually caused by the lack of a faithful lifecycle model, can cause these analyses to miss important data flows. In practice even worse, though, the tools' over-approximations can cause many false warnings, easily overwhelming security analysts to the point at which they stop using the analysis tools entirely.

In this work, we therefore present FLOWDROID, a novel static taint-analysis system specifically tailored to the Android platform, and based on novel on-demand algorithms that yield high precision while maintaining acceptable performance. FLOWDROID analyzes the apps' bytecode and configuration files to find potential privacy leaks, either caused by carelessness or created with malicious intention. Opposed to earlier analyses, FLOWDROID is the first static taint-analysis system that is fully context, flow, field and object-sensitive while precisely modeling the complete Android lifecycle, including the correct handling of callbacks and user-defined UI widgets within the apps. This design maximizes precision and recall, i.e., aims at minimizing the number of missed leaks and false warnings. To obtain deep context and object sensitivity while maintaining acceptable performance, FLOWDROID uses a novel on-demand alias analysis. The analysis algorithm is inspired by Andromeda [37] but improves over Andromeda's in terms of precision. We have open-sourced FLOWDROID in summer 2013. The tool has already been picked up by several research groups and we are in contact with a leading producer of anti-virus tools, who plans to use FLOWDROID productively in the analysis backend.

For us and others to be able to measure scientific progress in this important field of research it is required that researchers are able to conduct comparative studies of Android taint-analysis tools. Unfortunately, up until now there exist no benchmarks that would allow for systematic studies. As another contribution of this work we thus make available DROIDBENCH, a novel open-source micro-benchmark suite for comparing the effectiveness of taint analyses for Android. We have made DROIDBENCH available online in spring 2013 and know of several research groups who have used it already to measure and improve the effectiveness of their Android analysis tools [19]. A first group of external researchers has already agreed to contribute further micro benchmarks to the suite [35].

FLOWDROID can be used to secure in-house developed Android apps as well as assist in the triage of Android malware. Both use cases demand not a perfect but yet a reasonably low rate of false positives and false negatives. A set of experiments with SecuriBench Micro, DROIDBENCH and some well-known apps containing data leaks shows that FLOWDROID finds a very high fraction of data leaks while keeping the rate of false positives low. On DROIDBENCH 1.0, FLOWDROID achieves 93% recall and 86% precision, greatly outperforming the commercial tools AppScan Source [2] and Fortify SCA [3]. Further experiments with real apps confirm FLOWDROID's utility in practice.

To summarize, this work presents the following original contributions:

- FLOWDROID, the first fully context, field, object and flow-sensitive taint analysis which considers the Android application lifecycle and UI widgets, and which features a novel, particularly precise variant of an on-demand alias analysis;
- a full open-source implementation of FLOWDROID,
- DROIDBENCH, a novel, open and comprehensive micro benchmark suite for Android flow analyses,
- a set of experiments confirming superior precision and recall of FLOWDROID compared to the commercial tools AppScan Source and Fortify SCA, and
- a set of experiments applying FLOWDROID to over 500 apps from Google Play and about 1000 malware apps from the VirusShare project [1].

We make available online our full implementation as an open source project, along with all benchmarks and scripts to reproduce our experimental results:

`http://sseblog.ec-spride.de/tools/flowdroid/`

Space limitations preclude us from including some details necessary to fully reproduce our approach. We thus publish an accompanying Technical Report, [13] which formalizes FLOWDROID's transfer functions and gives additional details on the implementation.

The paper continues as follows. Section 2 gives a motivating example and explains the necessary background on Android security. Section 3 explains how FLOWDROID models the Android lifecycle while Section 4 gives important details about the actual taint analysis. In Section 5, the paper discusses implementation details and limitations, while Section 6 evaluates FLOWDROID. Section 7 discusses related work and Section 8 concludes.

2. Background and Example

We start by giving a motivating example and then explain the attacker model this work assumes. The example in Listing 1 (abstracted from a real-world malware app [42]) implements an activity, which in Android represents a screen in the user interface. The app reads a password from a text field (line 5) whenever the framework restarts the app. When the user clicks on a button of the activity, the password is sent via SMS (line 24). This constitutes a tainted data flow from the password field (the source) to the SMS API (the sink). In this example, *sendMessage()* is associated with a button in the app's UI, which is triggered when the user clicks the button. 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—the analysis must also process the metadata 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 avoid false negatives, a taint analysis must model the app lifecycle correctly, recognizing that a user may indeed hit the button after an app has restarted.

To avoid false positives, an analysis of this example must be field sensitive: the *user* object contains two fields for the user name and password, but only the latter of which should be considered a private value. Object-sensitivity, while not required for this example, is essential to distinguish objects originating at different allocation sites but reaching the same code locations. In our experiments we found some cases requiring deep object sensitivity to be able to automatically dismiss false positives. This is due to the relatively deep call and assignment chains of the Android framework.

Operations such as string concatenation (line 19) require a model that defines whether and how data flows through those operations. Treating such operations as normal method calls and analyzing library methods like application code can be imprecise (because

```
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.pwdString);
6    String uname = usernameText.toString();
7    String pwd = passwordText.toString();
8    if(!uname.isEmpty() && !pwd.isEmpty())
9      this.user = new User(uname, pwd);
10 }
11 //Callback method in xml file
12 public void sendMessage(View view){
13   if(user == null) return;
14   Password pwd = user.getpwd();
15   String pwdString = pwd.getPassword();
16   String obfPwd = "";
17   //must track primitives:
18   for(char c : pwdString.toCharArray())
19     obfPwd += c + "_"; //String concat.
20
21   String message = "User: " +
22      user.getName() + " | Pwd: " + obfPwd;
23   SmsManager sms = SmsManager.getDefault();
24   sms.sendTextMessage("+44 020 7321 0905",
25     null, message, null, null);
26 }
```

Listing 1: Example Android Application

it ignores the operations' semantics) and, as we found, is often forbiddingly expensive in practice.

Attacker model FLOWDROID can be used to detect data flows in general, no matter whether they are caused by carelessness or malicious intent. For malicious cases, we assume the following attacker model. The attacker may supply an app with arbitrary malicious Dalvik bytecode. Typically, the attacker's goal would be to leak private data through a dangerously broad set of permissions granted by the user [4]. FLOWDROID makes sound assumptions on the installation environment 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 or exploiting side channels. Further, we assume that the attacker does not use implicit flows [20] to disguise data leaks. Given the current kind of available malware, this is a very reasonable assumption.

3. Precise Modelling of Lifecycle

In the following we explain FLOWDROID's precise modeling of the lifecycle, including entry points, and asynchronously executing components and callbacks.

Multiple entry points Unlike Java programs, Android applications do not have a main method. Apps instead comprise many *entry points*, i.e., methods that are implicitly called by the Android framework. The Android operating system defines a complete lifecycle for all components in an application. There are four different kinds of components an app developer can define: *activities* are single focused user actions, *services* perform background tasks, *content providers* define a database-like storage, and *broadcast receivers* listen for global events. All these components are implemented by deriving a custom class from a predefined operating system class, registering it in the `AndroidManifest.xml` file and overwriting the lifecycle methods. The Android framework calls these methods to start or stop the component, or to pause or resume it, depending on environment needs. For instance, it can stop an application because of memory depletion, and later restart it when the user returns to it [17]. In result, when constructing a call graph, Android analyses cannot simply start by inspecting a predefined "main" method. Instead, all possible transitions in the Android lifecycle must be modeled precisely. To cope with this problem, FLOWDROID constructs a custom *dummy main* method emulating the lifecycle. In the following we explain how this method is constructed.

Asynchronously executing components An application can contain multiple components, e.g., three activities and one service. Although the activities run sequentially, one cannot pre-determine their order. One activity could, for instance, be the main one initially visible to the user and then launch either one of the others depending on user input. Services run as parallel background tasks. FLOWDROID models this execution by assuming that all components (activities, services, etc.) inside an application can run in an arbitrary sequential order (including repetition). Some static analyses are path sensitive, i.e., consider each possible program path separately. In such cases, considering all possible orderings would be very expensive. FLOWDROID bases its analysis on IFDS [32], an analysis framework which is *not* path sensitive and instead joins analysis results immediately at any control-flow merge point. FLOWDROID can thus generate and efficiently analyze a dummy main method in which every order of *individual* component lifecycles and callbacks is possible; it does *not* need to traverse all possible paths.

Callbacks The Android operating system allows applications to register callbacks for various types of information, e.g., location updates or UI interactions. FLOWDROID models these callbacks in its dummy main method, for instance to recognize cases where an application stores the location data that the framework passes to the callback as a parameter, and later sends this data to the Internet when the activity is stopped. The order in which callbacks are invoked cannot generally be predicted, which is why FLOWDROID assumes that all callbacks can be invoked in any possible order. However, callbacks can only happen while the parent component (e.g. activity) is running. For precision, FLOWDROID thus associates components (activities, services, etc.) with the callbacks they register. An activity may, for instance, register callbacks that get invoked when a button is pressed. The respective callback handler would then have to be analyzed between the *onResume()* and *onPause()* events of this activity only.

There are two different ways to register callback handlers on the Android platform. Firstly, callbacks can be defined declaratively in the XML files of an activity. Alternatively, they can also be registered imperatively using well-known calls to specific system methods. FLOWDROID supports both ways. Additionally, for malware there is the risk that an attacker registers undocumented callbacks by overwriting methods of the Android infrastructure, some of which could even be called by native code. FLOWDROID recognizes such overwritten methods, handling them similar to normal callback handlers such as button clicks.

For finding callbacks registered in the application code, FLOWDROID first computes one call graph per component, starting at the lifecycle methods (*onCreate()*, *onStop()*, etc.) implemented in the respective component class. This call graph is then used to scan for calls to Android system methods that use one of the well-known callback interfaces as a formal parameter type. Afterwards, the call graph is incrementally extended to include these newly discovered callbacks, and the scan is run again since callback handlers are free to register new callbacks on their own, potentially requiring FLOWDROID to re-extend the call graph and re-analyze until a fixed point it reached. While this method is more expensive than just scanning for classes implementing the callback interfaces, it delivers a more

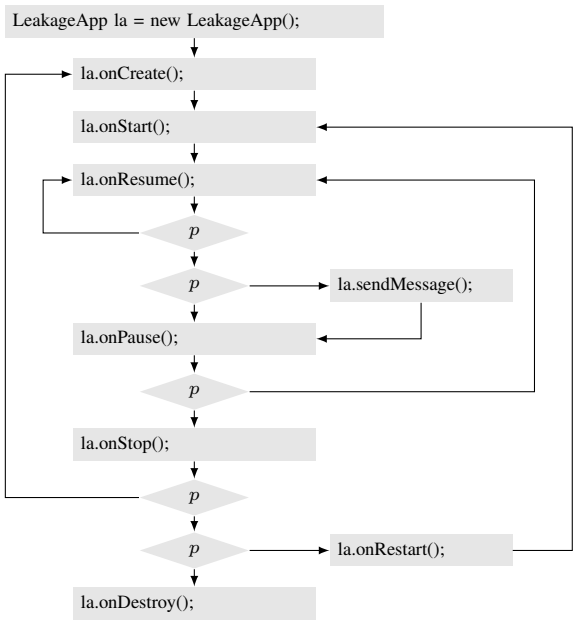

Figure 1: CFG for dummy main method

precise mapping between components and callbacks. This does not only reduce false positives, but we also found it to considerably decrease the runtime of the following taint analysis. Once the dummy main method has been constructed, FLOWDROID computes a final call graph using this method as the app's entry point.

For callbacks defined in the layout XML files, the respective XML file is mapped to one or more application components using the respective layout controls. A button-click handler, for instance, is only valid for the activity that hosts the respective button. FLOWDROID analyzes each activity to see which identifiers from the XML file it registers. This information is then used to create the mapping.

Example Note that, to gain maximal precision, FLOWDROID generates a new dummy main method for each app analyzed. Each main method will only involve the part of the lifecycle that, according to the app's XML configuration files, can actually occur at runtime. Disabled activities are automatically filtered and callback methods are only invoked in the contexts of the components to which they actually belong. A button-click handler, for instance, is only analyzed in the context of its respective activity. In Figure 1 we show the control-flow graph of the dummy main method for our previous example. The graph models a generic activity lifecycle augmented with the `sendMessage` callback. In this figure, p represents an *opaque predicate* of which we know that FLOWDROID won't be able to evaluate it statically. In result, the analysis will automatically consider on equal terms both branches for conditions involving p.

4. Precise Flow-Sensitive Analysis

One major difficulty in the analysis is how to implement high object sensitivity to resolve aliasing effectively. Figure 2 (abstracted from a real-world case) shows how FLOWDROID combines a forward-taint analysis and an on-demand backward-alias analysis to deduce that `b.f` is tainted at the sink. In step ①, the tainted variable `w` is propagated forward, tainting the heap object `x.f`. Step ② continues the taint tracking for `w` and `x.f`. The important step is ③: Whenever a heap object gets tainted, the backward analysis searches upwards for aliases of the respective object (`x.f` in this case). At ⑦, the alias `b.f` is found and then propagated forward as a normal taint.

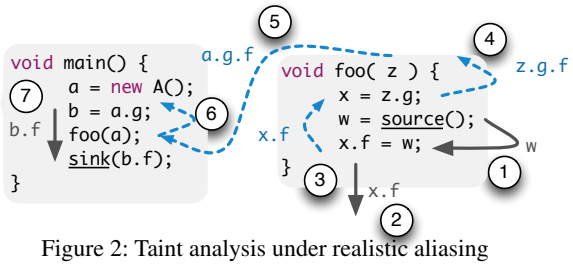

Figure 2: Taint analysis under realistic aliasing

FLOWDROID models the taint-analysis problem within the IFDS [32] framework for inter-procedural distributive subset problems. Section 4.1 explains the transfer functions that the analysis uses. Most functions are relatively standard. There is one important situation, however, in which FLOWDROID's analysis differs from standard taint-analysis algorithms, namely at statements at which tainted values are assigned to the heap, i.e., to fields or arrays. This situation will cause the backward alias analysis to be called, details of which we will explain in Section 4.2. Due to space restrictions we keep the description of flow functions on an informal level. To allow others to reproduce our approach, the accompanying Technical Report contains a complete formalization [13].

4.1 Taint analysis

Both the forward and backward analysis propagate access paths. An access path is of the form $x.f.g$ where x is a local variable or parameter and f and g are fields. Access paths can have different lengths up to a user-customizable maximal length (5 by default). An access path of length 0 is a simple local variable or parameter, e.g., x. In FLOWDROID, an access path implicitly describes the set of all objects reachable through this path, e.g., $x.f$ includes taints $x.f.g$, $x.f.h$, $x.f.g.h$ and so on.

The transfer function for assignments taints the left-hand side if any of the operands on the right-hand side is tainted. Assignments to array elements are treated conservatively by tainting the entire array. Assigning a "`new`"-expression to a variable x erases all taints modeled by access paths rooted at x. Method calls translate access paths to the callee's context by replacing actual with formal parameters; the inverse translation happens at method returns, including the return value if present. As usual with IFDS-based analyses, FLOWDROID also includes a call-to-return flow function (bypassing each method call on the side of the caller). This function propagates taints not relevant for the call, generates new taints at sources, reports taints at sinks and propagates taints for native calls. Section 5 gives further information on the latter.

4.2 On-demand alias analysis

Whenever a tainted value is assigned to a heap location such as a field or an array, FLOWDROID searches backwards for aliases of the target variable to then taint them as well. In Listing 2, for now consider the first call to `taintIt` (line 3), which taints the formal parameter `in`. In line 10, this will cause the access path `x.f` to get tainted due to the assignment `x.f = in`. In this situation (generally at all assignments to the heap), FLOWDROID will initiate a backward search for aliases of `x.f`, finding `out.f` in line 9. At this point, a new forward taint propagation is started for `out.f` from this statement, which will eventually discover the leak in line 11. The backward analysis will also continue to search backwards, though, discovering the alias `p.f` in `main`, with which it then spawns a forward analysis leading to a second taint-flow report at line 4.

Maintaining context sensitivity Algorithms 1 and 2 show the main loops of both the forward and the backward analysis solver in

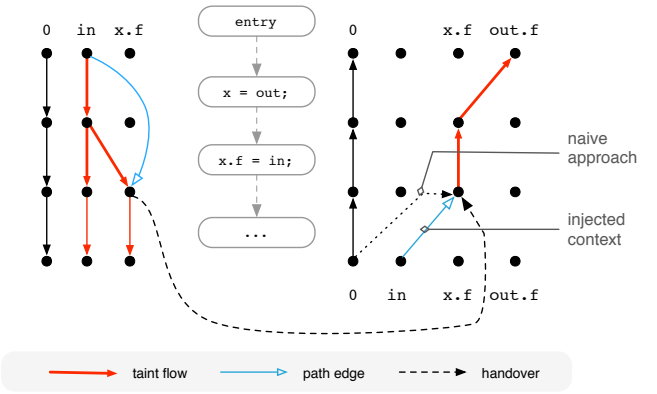

Figure 3: Analysis handover with context injection in `taintIt`

Algorithm 1 Main loop of forward solver

1: **while** $WorkList_{FW} \neq \emptyset$ **do**
2: pop $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ off $WorkList_{FW}$
3: **switch** (n)
4: **case** n is call statement:
5: **if** summary exists for call **then**
6: apply summary
7: **else**
8: map actual parameters to formal parameters
9: **end if**
10: **case** n is exit statement:
11: install summary $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$
12: map formal parameters to actual parameters
13: map return value back to caller's context
14: **case** n is assignment $lhs = rhs$:
15: $d_3 :=$ replace rhs by lhs in d_2
16: insert $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$ into $WorkList_{BW}$
17: extend path-edges via the *propagate*-method of the classical IFDS algorithm
18: **end while**

Algorithm 2 Main loop of backward solver

1: **while** $WorkList_{BW} \neq \emptyset$ **do**
2: pop $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ off $WorkList_{BW}$
3: **switch** (n)
4: **case** n is call statement:
5: **if** summary exists for call **then**
6: apply summary
7: **else**
8: map actual parameters to formal parameters
9: **end if**
10: extend path-edges via the *propagate*-method of the classical IFDS algorithm
11: **case** n is method's first statement:
12: install summary $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$
13: insert $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ into $WorkList_{FW}$
14: do *not* extend path-edges via the *propagate*-method of the classical IFDS algorithm, killing current taint d_2
15: **case** n is assignment $lhs = rhs$:
16: $d_3 :=$ replace lhs by rhs in d_2
17: insert $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$ into $WorkList_{FW}$
18: extend path-edges via the *propagate*-method of the classical IFDS algorithm
19: **end while**

```
1  void main() {
2    Data p = new ..., p2 = new ...
3    taintIt(source(), p);
4    sink(p.f);
5    taintIt("public", p2);
6    sink(p2.f);
7  }
8  void taintIt(String in, Data out) {
9    x = out;
10   x.f = in;
11   sink(out.f);
12 }
```

Listing 2: Example for context injection

pseudo code. The algorithmic representation assumes the reader to be familiar with the algorithmic description of the original IFDS algorithm [32]. Both solvers operate on their own worklist, containing so-called path-edges that summarize the data-flows computed so far up to the current statement/node n. An edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ effectively states that the analysis concluded that d_2 holds at n if d_1 holds at the start point s_p of n's procedure p. In our particular implementation, the abstract-domain values d_i are effectively access paths describing references to tainted values. The handover between both analyses is quite non-trivial. If coordinated in a naive fashion, one will easily obtain two independent analyses that each on their own may be context sensitive, but would in combination produce analysis information for unrealizable paths along conflicting contexts. For instance, note that in the example from Listing 2 the aliases of `x.f` become tainted only if `in` was tainted previously. In particular, the analysis should *not* report a leak at line 6, whose corresponding `taintIt`-call only propagated the string `"public"`.

Figure 3 shows both how a naive implementation could cause such a false positive, and how FLOWDROID handles the problem by injecting contexts from one analysis to another. The figure assumes some familiarity with the typical notation [32] for flow functions within the IFDS framework. Here the black nodes represent data-flow facts before/after the respective statement and the black and red edges represent data flows. The fact **0** is the tautological fact that is always true, which is why one **0** node always connects to the next. The left-hand side of the figure shows how the forward taint analysis determines `x.f` to be tainted. When processing the assignment to `x.f`, the forward analysis spawns an instance of the backward alias analysis, shown on the right-hand side. The naive way to spawn this analysis would be to initialize it with an edge from **0** to `x.f` (dotted line). This implementation, although straightforward, leads to imprecision, as its semantics state that aliases of `x.f` are tainted no matter what. In Listing 2, this could cause the analysis to incorrectly report a taint violation even for `p2.f`. The correct way is thus to inject into the backwards analysis the context of the forward analysis: FLOWDROID consults the "path edge" to `x.f`, which the IFDS algorithm stores as a side-effect of its summary computation. It then injects that entire edge into the backward solver. (see Algorithm 1, line 16) Context injection happens both ways. At line 9 in the example, when the backward analysis spawns a forward analysis for `out.f`, it injects into the forward analysis the original context `in`. (see Algorithm 2, line 17) Semantically, for the example this implies that all taints that both analyses discover for `taintIt` are conditional w.r.t. `in` being tainted initially.

A second problem is to avoid false positives due to unrealizable paths: FLOWDROID needs to prevent the backwards analysis to return into contexts not analyzed by the forward analysis (and vice versa). To implement this constraint, the backward analysis in FLOWDROID actually never returns into the caller at all. Instead,

```
1 Data p = new ..., p2 = p;
2 sink(p2.f);
3 p.f = source();
4 sink(p2.f);
```

Listing 3: Example for activation statements

whenever finding an alias it triggers the forward analysis on that alias, such as for `out.f` in line 9. It is then the task of the forward analysis to map back any relevant taints into the caller's context. In the example, the forward analysis knows the calling context it originated from, which is why it can easily make sure to map back the taints into the right context only. In the example, the forward pass would map `out.f` to `p.f` in line 3 only, not to `p2.f` in line 5. In essence, the backwards analysis can descend into callees, but never returns back into callers; all returns are handled by the forward analysis. When the backwards analysis descends into a call, it will eventually spawn a forward analysis when reaching the method header. (see Algorithm 2, line 13) The forward analysis can then make sure to only return into the right caller because its context is injected by the backward analysis. (Technically, its incoming-set [26] is injected.) Whenever the forward analysis maps back to the caller a taint associated with a heap object, it spawns a new alias search inside the caller.

Maintaining flow sensitivity Andromeda [37] is another taint-analysis tool that inspired FLOWDROID's on-demand alias analysis. Andromeda's analysis, however, can lead to flow-insensitive results. In the example in Listing 3, the analysis would report two leaks at lines 2 and 4, even though the first call to `sink` definitely happens before `p2.f` becomes tainted. In fact, the very same problem would hold also for FLOWDROID's analysis as we described it above: the backward analysis would discover the tainted alias `p2.f` at line 1 and trigger a forward pass with that value, causing a taint to be reported henceforth anywhere where `p2.f` is leaked.

FLOWDROID addresses this problem by keeping track of what we call *activation statements*. Whenever spawning an instance of the backwards alias analysis, the respective access path is augmented with the current statement, the alias' activation statement. Also, the tainted alias is marked as *inactive*. Semantically, only active taints cause leaks when reaching a sink. Inactive taints are aliases to memory locations which have not yet been tainted. Whenever the backward analysis spawns the forward analysis again, and when then the forward analysis propagates the aliased taint over its activation statement, the taint becomes activated and thus gains its ability to actually cause leaks to be reported. In the example, the activation statement is at line 3, which thus causes the analysis to only report a leak at the succeeding line 4, avoiding the false alarm at line 2.

In general, activation statements are representatives of call trees. Assume for a moment that the heap assignment in Listing 3 was contained inside a method call such as it was the case in the assignment at line 10 of Listing 2, which occurs within method calls to `taintIt`. In that example, when the forward analysis processes the return edge back into line 3 of `main`, the analysis globally associates the call to `taintIt` (line 3) with the activation statement since whenever this call has completed, the activation statement also has been processed and thus the taint will be active. In other words, activation statements are used for looking up the call trees in which they occur to translate them back into (transitive) callers.

To the best of our knowledge, FLOWDROID is the first approach to implement an on-demand analysis that fully maintains context and flow sensitivity. In the future we plan to investigate to what extent the principles explained here can be reused outside the scope of taint analysis, ideally yielding a rather generic extension of IFDS.

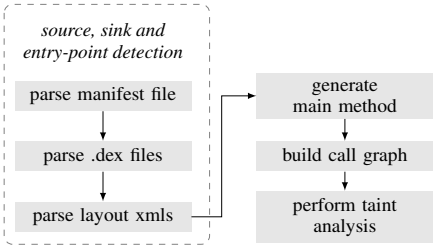

Figure 4: Overview of FLOWDROID

Further algorithmic details Our implementation of IFDS uses the extensions explained by Naeem and Lhoták [26]. With this extension, the IFDS implementation computes the program's super graph on the fly, which means that in our case we compute taint information only for those variables/access paths that are indeed tainted. Our implementation uses two instances of the IFDS solver, each of which with slight adjustments as explained in Algorithms 1 and 2. Each instance contains a separate table of summary functions that, as in the original IFDS algorithm, is used to avoid re-computation for the same callees under the same contexts d_i.

5. Implementation

FLOWDROID extends the Soot framework [21] which provides important prerequisites for a precise analysis, in particular the three-address code intermediate representation Jimple and the accurate call-graph analysis framework Spark [22]. A plugin called Dexpler [5] allows FLOWDROID to convert Android's Dalvik bytecode into Jimple. On top of Soot and Dexpler, FLOWDROID further uses Heros [7], a scalable, highly multi-threaded implementation of the IFDS framework [32]. We next explain FLOWDROID's architecture, while the subsequent sections explain important implementation details and FLOWDROID's current limitations.

Architecture Figure 4 shows FLOWDROID's architecture. Android applications are packaged in *apk* files (Android Packages), which are essentially zip-compressed archives. After unzipping an archive, FLOWDROID searches the application for lifecycle and callback methods as well as calls to sources and sinks. 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. Next, FLOWDROID generates the dummy main method from the list of lifecycle and callback methods. This main method is then used to generate a call graph and an inter-procedural control-flow graph (ICFG). Starting at the detected sources, the taint analysis then tracks taints by traversing the ICFG as explained in Section 4. FLOWDROID is configured with sources and sinks inferred by our SuSi project [30], by far the most comprehensive one available. The concrete lists of sources and sinks are available from the FLOWDROID website. At the end, FLOWDROID reports all discovered flows from sources to sinks. The reports include full path information. To obtain this information, the implementation links data-flow abstraction objects to their predecessors and to their generating statements. This allows FLOWDROID's reporting component to fully reconstruct a graph of all relevant assignment statements that might have caused a taint violation at the given sink.

Defining shortcuts Including the full JRE or Android platform runtime in the analysis not only requires a lot of analysis time and memory, but, due to approximations performed during the library's analysis, can also lead to undesired imprecision. FLOWDROID therefore comprises an interface for external library models. The tool supports a simple textual file format for defining certain "shortcut

rules". Predefined rules handle collection classes, string buffers and similar commonly used data structures, e.g., specifying that adding a tainted element to a set taints the entire set. Technically, the shortcuts are implemented using the call-to-return edge. If a library call has no associated rule then it is fully analyzed.

Native Calls Both Java and the Android platform support the invocation of native methods written in C or other unmanaged languages. For a Java-based analysis, such methods are black boxes which cannot be analyzed. FLOWDROID comes with explicit taint-propagation rules for the most common native methods, such as `System.arraycopy`. In this example, the rule defines the third argument (the output array) to become tainted if the first argument (the input array) is tainted before the call. For native methods without an explicit rule, FLOWDROID assumes a sensible default: 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 likely the best practical approximation in a black-box setting.

Inter-Component Communication FLOWDROID over-approximates explicit inter-component communication by regarding method which send intents as sinks and callbacks which receive intents as sources. Android also supports implicit intent-based communication, e.g., by setting the result value of a called activity which is then automatically passed back to the calling activity by the operating system. Supporting such behavior together with a more accurate inter-component connection model is left to future work. In particular, we are working on integrating FLOWDROID with EPICC [27], a novel static analysis that uses Soot and Heros to perform a String analysis to resolve inter-app communication more precisely.

Limitations Although FLOWDROID is generally aiming for a sound analysis, it does share some inherent limitations with most other static-analysis tools. For instance, FLOWDROID resolves reflective calls only if their arguments are string constants, which is not always the case. On the Java platform, reflection-analysis tools such as TamiFlex [8] can be used to make static analysis tools aware of reflective calls issued at runtime. Such tool require load-time instrumentation through `java.lang.instrument`, though, which the Android platform does not currently support. Unsoundness can also arise in case the Android lifecycle contains callbacks we are not aware of, or through native methods that our rules model incorrectly. At the moment FLOWDROID is also oblivious to multi-threading: it assumes threads to execute in an arbitrary but sequential order, which is generally unsound as well. Fully incorporating sound support for multi-threading is a big challenge in its own right, which we thus leave to future work.

6. Experimental Evaluation

Our evaluation addresses the following research questions:

RQ1 How does FLOWDROID compare to commercial taint-analysis tools for Android in terms of precision and recall?

RQ2 Can FLOWDROID find all privacy leaks in InsecureBank, an app specifically designed by others to challenge vulnerability-detection tools for Android [28], and what is its performance?

RQ3 Can FLOWDROID find leaks in real-world applications and how fast is it?

RQ4 How well does FLOWDROID perform when being applied to taint-analysis problems related to Java, not Android, both in terms of precision and recall?

The next sections address each research question in detail. Section 6.5 explains why, unfortunately, we were unable to directly compare FLOWDROID to other academic Android analysis tools.

6.1 RQ1: Commercial taint-analysis tools

While there are benchmark suites for analyzing web applications or specifically for detecting different kinds of Java vulnerabilities [23], there is no Android-specific analysis benchmark suite at the moment. This is problematic because the generic Java test suites do not cover aspects like the Android lifecycle, callbacks or interactions with UI elements like password fields. Thus, they cannot be used for assessing the practical effectiveness of Android analysis tools.

DroidBench Specifically for this work, we therefore developed an Android-specific test suite called DROIDBENCH. In this evaluation we consider version 1.0, which contains 39 hand-crafted 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. Our Technical Report [13] gives additional information about the individual apps. We have made available online DROIDBENCH in spring 2013 and know of several research groups [19] who have used it already to measure and improve the effectiveness of their Android analysis tools. A first group of external researchers has already agreed to contribute further micro benchmarks to the suite [35].

Table 1 presents the analysis results for FLOWDROID and two commercial analysis tools (explained in the following) when applied to DROIDBENCH. As the results show, FLOWDROID achieves 93% recall and 86% precision.[1] As explained before, for performance reasons, 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 interesting research question in its own [10]. *Button2* causes a false positive because FLOWDROID does not currently support strong updates. In result, it cannot kill taints for certain button combinations. Allowing strong updates would require a (probably quite expensive) must-alias analysis. Incorporating such an analysis into FLOWDROID is out of the scope of this work. *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. Determining exactly where such initializers can execute at runtime is an interesting research question. We plan to add better support in the future.

Comparison with IBM AppScan Source We compared FLOWDROID with IBM AppScan Source [2] version 8.7, on all tests from DROIDBENCH. AppScan Source distinguishes three different categories of findings: vulnerabilities, exceptions of type 1 and exceptions of type 2. Like reports by FLOWDROID, vulnerabilities include a complete path from source to sink. For a type 1 exception, there is a flow from source to sink as well, but the semantics of some methods along the propagation path is unknown (e.g. possible sanitization). Since FLOWDROID does not support sanitization at the moment, we consider both vulnerabilities and type 1 exceptions as findings. For type 2 exceptions on the other hand, there is no trace. These reports are generated when certain code constructs (e.g. writing a variable value into the log file) are detected. As these findings are highly imprecise and completely disregard data flow, we do not count them as findings. As Table 1 shows, AppScan Source finds only about 50% of all leaks. Major problems occur with the

[1] We exclude the analysis of implicit flows [20] caused through control-flow dependencies as none of the tools, including FLOWDROID was designed to analyze such flows.

⊛ = 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	Fortify	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			
Sum, Precision and Recall			
⊛, 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 1: DROIDBENCH test results

handling of callbacks and the Android component. It appears like the advertised support for Android is mostly restricted to AppScan being configured with some appropriate sources and sinks. AppScan shows a relatively decent precision of 74%.

Comparison with Fortify SCA Fortify SCA [3] by HP is another commercial tool widely used by security analysts. Similar to IBM AppScan Source, 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. All tests were carried out using version 5.14. As can be seen in Table 1, Fortify SCA shows problems similar to those of IBM AppScan, like the handling of the Android component lifecycle and callbacks. Figure 1 shows that Fortify detects 4 out of 6 data leaks for the lifecycle tests, but closer inspection shows that this only happens by chance. In these tests, the data source involves a static field, which Fortify apparently treats in a special way that coincidentally causes a leak to be reported. When removing the static modifier, which does not change the semantics of the test case, Fortify does not detect the leak any longer. Fortify's precision measures as 81%.

Conclusion From our experiments we conclude that, to not overburden the user with false positives, AppScan Source and Fortify SCA aim for relatively high precision while sacrificing recall, thus risking to miss actual privacy leaks. In comparison, FLOWDROID shows a significantly higher recall and even a slightly improved precision.

6.2 RQ2: Performance on InsecureBank

InsecureBank [28] 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. Analyzing the application takes about 31 seconds on a laptop computer with an Intel Core 2 Centrino CPU and 4 GB of physical memory running on Windows 7 with Oracle's Java Runtime version 1.7 (64 bit) in its default settings. FLOWDROID finds all seven data leaks which we all verified by hand. There are no false positives nor false negatives.

6.3 RQ3: Performance on Real-World Applications

For assessing FLOWDROID on real applications, we applied it to the 500 most popular Android applications from Google Play.[2] Fortunately, despite FLOWDROID's high recall, the analysis did not reveal any leaks hinting at truly malicious behavior. Nevertheless, the majority of apps was reported to—probably accidentally—leak sensitive information like the IMEI (a unique ID) or location data into logs and preference files.

Samsung's Push Service, for instance, logs the phone's IMEI. Logs are problematic, as the OS does not impose the same access restrictions on logs as it does on files: for devices running Android 4.0 or lower, all logs are readable by any app that has the READ_LOGS permission. The problem is deemed so important that the mechanism was changed in Android 4.1. Since this version, logs are only privately visible, unless the app is run with debugging enabled. Additionally, Samsung's Push Service also broadcasts an Android intent containing the IMEI. All other applications can simply subscribe to this intent and get the broadcast IMEI, thereby circumventing the Android permission system for this data item.

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 again how important a precise environment model is to reduce the number of false positives. Future tools should thus model the respective APIs more precisely.

For most examined apps FLOWDROID terminated in under a minute. The instance that took the longest to complete was Samsung's Push Service which took about 4.5 minutes to analyze.

We also ran FLOWDROID on about 1000 known malware samples from the VirusShare project [1]. The average runtime was 16 seconds since the Malware samples seem to be comparatively small. The minimum runtime was 5 seconds, the maximum was 71 seconds which only happened for a single, comparatively large application. Most of the apps contained two data leaks (1.85 leaks per application on average), usually with identification information like the

[2] For legal reasons we are unable to provide these applications online. To be able to reproduce our results, though, researchers may email the first author to obtain a copy of those applications.

Test-case group	TP	FP
Aliasing	11/11	0
Arrays	9/9	6
Basic	58/60	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	0
StrongUpdates	0/0	0
Sum	117/121	9

Table 2: SecuriBench Micro test results

IMEI being sent to a remote server to register the phone, or sent as part of an SMS message, sometimes to a premium-rate number. Some malware applications were even found to receive data through broadcast receivers and to then send out this data in SMS messages. This can allow other applications to send SMS messages indirectly, without requiring the respective permission on their own.

6.4 RQ4: SecuriBench Micro

FLOWDROID was specifically designed for Android, and in this space 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 FLOWDROID is set up for this use case, we evaluated FLOWDROID against Stanford SecuriBench Micro [23] version 1.08, a common set of 96 J2EE micro benchmarks originally intended for web-based applications. For each of the benchmarks in the suite, we manually 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. We omitted from our experiments test cases involving sanitization, reflection, predicates and multi-threading. 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.

Table 2 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 Basic, for instance, FLOWDROID found 58 out of 60. The **FP** column shows the number of *false positives*, i.e., the finding that 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 in which FLOWDROID reports 6 false positives. Again, those are caused by the failure to model array indices precisely.

6.5 Comparison with Other Tools

We also tried to compare FLOWDROID to a number of other tools from the scientific literature, namely TrustDroid [41], LeakMiner [40], and the tool by Batyuk et al. [6]. Unfortunately none of those tools are available online, nor did the respective authors reply to our inquiries.

We tried to run DROIDBENCH on SCanDroid [14], 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 and both sides eventually gave up. The authors of AndroidLeaks [15] promised to run their tool on DROIDBENCH but never delivered. We also contacted the authors of CHEX [24], but they were unable to provide the tool or any benchmark results due to intellectual property claimed by NEC. Starostin [25] declined to participate in the experiment as his tool ignores aliasing, making any comparison meaningless. The authors of ScanDal [19] could not provide their tool due to intellectual property claimed by Samsung, but used our benchmark suite and feedback to improve their analysis. According to the authors, on DROIDBENCH, the analysis now reports results similar to FLOWDROID.

In result, we were unable to successfully evaluate even a single scientific taint-analysis tool for Android on our own. This is quite unfortunate, as it restricts us to comparing to those tools only based on the available publications. We hope that the availability of FLOWDROID and DROIDBENCH will greatly improve this situation in the future. After all, publishing irreproducible results hinders progress in the field and is considered unacceptable in most other sciences.

7. Related Work

There are several approaches to static analysis of Android applications differing in precision, runtime, scope and focus.

One of the most sophisticated ones is CHEX [24], a tool to detect component hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources or sinks. Although not built for the task, CHEX can, in principle, be used for taint analysis. CHEX does not analyze calls into 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 precision and performance. Users can thus omit the model entirely and still be sure not to lose taints. CHEX's entry-point model requires an enumeration of all possible "split orderings" which is not necessary in FLOWDROID. Furthermore, CHEX is limited to at most 1-object-sensitivity, while FLOWDROID's demand-driven alias analysis allows for contexts of arbitrary lengths (using a default of 5). We found 1-object-sensitivity to be too imprecise in practice.

LeakMiner [40] appears similar to our approach from a technical point of view: like FLOWDROID, it is based on Soot, uses Spark for call-graph generation, it implements the Android lifecycle, and the paper states that an app can be analyzed in 2.5 minutes on average. However, the analysis is not context-sensitive which precludes the precise analysis of most test cases in DROIDBENCH.

AndroidLeaks [15] 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, but is not as precise as FLOWDROID, because it taints the whole object if tainted data is stored in one of its fields, i.e., is neither field nor object sensitive. This precludes the precise analysis of many practical scenarios.

SCanDroid [14] 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. This is much less precise than FLOWDROID's treatment; FLOWDROID applies this default rule only for *native* calls not modeled explicitly. FLOWDROID currently models intent sending as sink and intent reception as source, yielding a sound treatment of inter-app communication. In the future, we plan to integrate FLOWDROID with EPICC [27], a novel static analysis that uses String analysis to precisely resolve inter-app communication.

Other approaches like CopperDroid [31] dynamically observe interactions between the Android components and the underlying Linux system to reconstruct higher-level behavior. Special stimulation techniques are used for exercising the application 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 [29]. Alternatively, data leaks might only occur after a certain runtime threshold. Aurasium [38] and DroidScope [39] largely suffer from the same shortcomings with respect to static leak detection.

TaintDroid [11] is one of the most sophisticated Android taint-tracking systems to date. As a dynamic approach, however, it yields some quite different tradeoffs compared to FLOWDROID. For instance, TaintDroid has no problem tracking taints through reflective method calls, as TaintDroid is implemented as an extension to the execution environment, for which it does not matter whether methods are invoked through reflection or not. On the other hand, if used for triaging malware before installation time, then TaintDroid can successfully detect malware only if paired with a dynamic testing approach that yields decent code coverage. Static ahead-of-time analyses like FLOWDROID do not share this shortcoming because they cover all execution paths. Secondly, a dynamic approach such as TaintDroid can be fooled by a malicious apps that recognize that it is being analyzed in which case the app could simply refrain from performing any malicious activities [29]. While this is not problematic if the dynamic analysis is installed on the end user's mobile phone (in that case, the malware would effectively be tamed), it is problematic if the dynamic analysis is only used for ahead-of-time triaging of malware that could then later on be installed on a system not protected by the dynamic analysis (in which case the app could resume its malicious activities). Static approaches such as FLOWDROID do not share this particular shortcoming as they never actually execute the app.

F4F [36] is a framework for performing taint analysis of framework-based applications using a specification language called WAFL for describing the functional behavior of the respective framework. While originally created for web applications, it might also be extended to model the Android framework by adding a WAFL generator for Android. FLOWDROID's dummy-main generation has the big advantage to only include components and callbacks that are indeed accessed by the app. This, however, requires a semantic model of the app's manifest, the layout XML files, the compiled resources file and the app's source code, which are all interleaved. F4F could at best be used to give a coarse approximation modeling the common denominator of all possible apps.

FLOWDROID currently handles exceptional flows through a coarse over-approximation. Kastrinis and Smaragdakis have recently presented a novel and particularly efficient approach for analyzing exceptions and points-to analysis in combination [18]. It would be interesting to see whether FLOWDROID can easily benefit from an integration of some of those concepts.

Rountev et al. have proposed a way to pre-compute summaries for large libraries with the intention to speed up the repeated analysis of client code [33]. For Android apps, which come with the huge Android framework, such an approach makes a lot of sense. Rountev's work is based on the IDE framework [34], which FLOWDROID also uses internally to conduct its IFDS-based analyses. It should therefore be entirely possible to incorporate the authors' ideas into FLOWDROID.

Dillig et al. developed an approach to more precisely analyzing the contents of collections and arrays [10]. The required analysis effort is non-trivial, but given our results it is clear that FLOWDROID could increase its precision further by implementing analysis supports along those lines.

8. Conclusions

We have presented FLOWDROID, a novel and highly precise static taint-analysis tool for Android applications. Unlike previous approaches, FLOWDROID adequately models Android-specific challenges like the application lifecycle or callback methods, which helps reduce missed leaks or false positives. Novel on-demand algorithms allow FLOWDROID to maintain efficiency despite its strong context and object sensitivity. For assessing the effectiveness of analysis tools, we have proposed the Android-specific benchmark suite DROIDBENCH and used it for comparing FLOWDROID to the commercial tools AppScan Source and Fortify SCA, showing that besides finding more real leaks (93% of all leaks in total), FLOWDROID also has a higher precision (86%) resulting in fewer false positives. We hope that in the future DROIDBENCH will serve as a standard test set for Android taint analyses.

For analyzing the top 500 real-world applications from Google's Play Store, FLOWDROID only took under a minute per application and found several leaks. About 1000 malware samples were analyzed in about 16 seconds per minute, uncovering 2 leaks per sample on average. An evaluation of FLOWDROID on SecuriBench Micro shows a 96% recall with only 9 false positives.

In future work we plan to improve the support for handling reflection. Also we are interested in pre-computing library abstractions automatically.

Acknowledgements We would like to thank Stephan Huber from Fraunhofer SIT for supporting us with real-world applications from the Google Play market and Dr. Karsten Sohr from TZI Bremen for supporting us with the Fortify SCA evaluation. Thanks to Marc-André Laverdière and others for contributions to a our implementations of FLOWDROID, Soot and Heros. This work was supported by a Google Faculty Research Award, by the BMBF within EC SPRIDE and ZertApps, by the Hessian LOEWE excellence initiative within CASED, by the DFG within the project RUNSECURE, a project associated with the DFG Priority Programme 1496 "Reliably Secure Software Systems – RS^3", by the Fonds National de la Recherche (FNR), Luxembourg, under the AndroMap project, and by the National Science Foundation Grants No. CNS-1228700, CNS-0905447, CNS- 1064944 and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or any other funding partner.

References

[1] Virus share, aug 2013. `http://virusshare.com/`.

[2] IBM Rational AppScan, Apr. 2013. `http://www-01.ibm.com/software/de/rational/appscan/`.

[3] Fortify 360 Source Code Analyzer (SCA), Apr. 2013. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1214365#.UW6CVKuAtfQ`.

[4] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: an application to android. In *ASE 2012*, pages 274–277, 2012.

[5] A. Bartel, J. Klein, Y. Le Traon, and M. 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, 2012.

[6] L. Batyuk, M. Herpich, S. Camtepe, K. Raddatz, A.-D. 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, 2011.

[7] E. 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, 2012.

[8] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. 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] I. D. Corporation. Worldwide quarterly mobile phone tracker 3q12, Nov. 2012. http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.

[10] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 187–200, 2011.

[11] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In R. H. Arpaci-Dusseau and B. Chen, editors, *OSDI*, pages 393–407. USENIX Association, 2010.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM. . URL http://doi.acm.org/10.1145/2046614.2046618.

[13] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013. URL http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf.

[14] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications.

[15] C. Gibler, J. Crussell, J. Erickson, and H. 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, 2012.

[16] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM. . URL http://doi.acm.org/10.1145/2185448.2185464.

[17] G. Inc. Application fundamentals. 2013. URL http://developer.android.com/guide/components/fundamentals.html.

[18] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In R. Jhala and K. D. Bosschere, editors, *CC*, volume 7791 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013.

[19] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In H. Chen, L. Koved, and D. S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[20] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 56–70, Berlin, Heidelberg, 2008. Springer-Verlag. .

[21] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oktober 2011.

[22] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In G. Hedin, editor, *Compiler Construction*, volume 2622 of *LNCS*, pages 153–169. Springer Berlin Heidelberg, 2003. .

[23] B. Livshits. Securibench micro, Mar. 2013. http://suif.stanford.edu/~livshits/work/securibench-micro/.

[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *CCS 2012*, pages 229–240, 2012.

[25] C. Mann and A. 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, 2012.

[26] N. A. Naeem, O. Lhoták, and J. Rodriguez. Practical extensions to the ifds algorithm. In *Compiler Construction 2010*, pages 124–144, 2010.

[27] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security Symposium 2013*, Aug. 2013.

[28] Paladion. Insecurebank test app. http://www.paladion.net/downloadapp.html.

[29] N. J. Percoco and S. Schulte. Adventures in bouncerland. *Blackhat USA*, 2012.

[30] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, Feb. 2014. URL http://www.bodden.de/pubs/rab14classifying.pdf. To appear.

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

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[33] A. Rountev, M. Sharp, and G. Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *Compiler Construction*, volume 4959 of *LNCS*, pages 53–68. Springer, 2008.

[34] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95*, pages 131–170, 1996.

[35] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices, 2013.

[36] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In *OOPSLA 2011*, pages 1053–1068, 2011.

[37] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE 2013*, pages 210–225, 2013.

[38] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security 2012*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[39] L. K. Yan and H. 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.

[40] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104, 2012.

[41] Z. Zhao and F. 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.

[42] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, 2012.