Technical Report

Nr. TUD-CS-2013-0113
May 8th, 2013

Highly Precise Taint Analysis for Android Applications

Authors
Christian Fritz (EC SPRIDE)
Steven Arzt (EC SPRIDE)
Siegfried Rasthofer (EC SPRIDE)
Eric Bodden (EC SPRIDE)
Alexandre Bartel (SnT, University of Luxembourg)
Jacques Klein (SnT, University of Luxembourg)
Yves le Traon (SnT, University of Luxembourg)
Damien Octeau (Penn State University)
Patrick McDaniel (Penn State University)

Highly Precise Taint Analysis for Android Application

Christian Fritz[1], Steven Arzt[1], Siegfried Rasthofer[1], Eric Bodden[1], Alexandre Bartel[2], Jacques Klein[2], Yves le Traon[2], Damien Octeau[3] and Patrick McDaniel[3]

[1]Secure Software Engineering Group, EC SPRIDE
[2]SnT, University of Luxembourg
[3]Penn State University
eric.bodden@ec-spride.de

ABSTRACT

Today's smart phones are a ubiquitous source of private and confidential data. At the same time, smartphone users are plagued by malicious apps that exploit their given privileges to steal such sensitive data, or to track users without their consent or even the users noticing. Dynamic program analyses fail to discover such malicious activity because apps have learned to recognize the analyses as they execute.

In this work we present FlowDroid, a novel and highly precise taint analysis for Android applications. A precise model of Android's lifecycle allows the analysis to properly handle callbacks, while context, flow, field and object-sensitivity allows the analysis to track taints with a degree of precision unheard of from previous Android analyses.

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, our approach finds a very high fraction of data leaks while keeping the rate of false positives low. On DroidBench, our approach achieves 93% recall and 86% precision, greatly outperforming the commercial tools AppScan Source and Fortify SCA.

Categories and Subject Descriptors

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

General Terms

Experimentation, Security, Verification

Keywords

Static analysis, Android, taint analysis

EC SPRIDE Technical Report TUD-CS-2013-0113

1. INTRODUCTION

According to a recent study [10], Android now has about 75% market share in the mobile-phone market, with a 91.5% growth rate over the past year. With Android phones being ubiquitous, they become a worthwhile target for security and privacy violations. Attacks range from broad data collection for the purpose of targeted advertisement, to targeted attacks, such as the case of industrial espionage. Attacks are most likely to be motivated primarily by a social element: a significant number of mobile-phone owners use their device both for private and work-related communication [7]. Furthermore, the vast majority of users install apps containing code whose trustworthiness they cannot judge and which they cannot effectively control.

These problems are well known, and indeed the Android platform does implement state-of-the-practice measures to impede attacks. The Android platform is built as a stack, with various layers running on top of each other [3]. The lower levels consist of an embedded Linux system and its libraries, with Android applications residing at the very top. Users typically acquire these applications through various channels (e.g., the Google Play Store[1]). The underlying embedded Linux system provides the enforcement mechanisms common to the Linux kernel, such as a user-based permission model, process isolation and secure inter-process communication.

By default, an application is not allowed to directly interact with other applications, operating system processes, or a user's private data [14]. The latter includes, for example, access to the contacts list. Android regulates access to such private data via a *permission-based* security model where, to access security-sensitive API functions, applications have to statically declare the permissions they require. An application may only be installed following the user's consent, yet users currently have little control over the installation process, as they must either grant all of the permissions that an app demands, or else forego installation. The problem is aggravated by the coarse-grained nature of Android permissions [20, 33]. For instance, an app may require internet and phone book access permissions to function correctly, in which case revoking either would not be an option. Nevertheless, one may wish to forbid the app from transmitting phone book entries over the internet. Android's existing permission system does not allow for such fine-grained restrictions on information flow. Unfortunately, practice shows that, as a result of this limitation, users grant too many permissions too often, thus running the risk to give malicious apps access to private data.

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

The industry has acknowledged the problem of malware, and has thus started to take measures through dynamic analyses. One example is Google's Bouncer [15], which test-runs apps for five minutes as they are uploaded into the Play 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 [23].

One possible solution to the problem are static analyses that analyze the apps' code without requiring their execution. But while static analyses for Android have made some progress over the past years, as we show, they still lack the necessary precision to be effectively useful in practice. One major challenge is the fact that Android apps are not just closed programs but run within the Android framework, which imposes a complex lifecycle on those apps, invoking a range of pre-defined or user-defined callback methods at different times during the app's execution. To be able to effectively predict the app's control flow, static analyses must model this lifecycle precisely. Another source of imprecision in previous analyses is their typically context and object-insensitive nature. A context-insensitive analysis joins together the analysis results for a method m at all call sites to m even if the arguments to m at those call sites differ. Similarly, an object-insensitive analysis lumps together analysis information for all objects that reach the same virtual call site. As previous work has shown, context and object sensitivity is key to successful static analyses [26].

In this work we thus present FlowDroid, to the best of our knowledge 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 user-defined UI widgets within the apps.

Another contribution of this work is DroidBench, a novel open micro-benchmark suite for comparing the effectiveness of taint analyses for Android. We intend to extend and maintain this suite as a community effort and hope that in the future it will be used for empirical evaluations that are more systematic and comparative than the ones that have appeared in the scientific literature to date.

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, our approach achieves 93% recall and 86% precision, greatly outperforming the commercial tools AppScan Source [2] and Fortify SCA [1]. FlowDroid thus sets a new gold standard for the static taint analysis of Android applications. 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/flowdroid/`

To summarize, this work presents the following original contributions:

- FlowDroid, the first fully context, object and flow-sensitive taint analysis to consider the Android application lifecycle and UI widgets; it can be configured easily to run with new Android versions;

- as one of few approaches in the field: an open-source implementation of the above analysis,

- DroidBench, a novel, open and comprehensive micro benchmark suite for Android flow analyses, and

- a set of experiments comparing the relative precision and recall of FlowDroid with the commercial tools AppScan Source and Fortify SCA.

2. BACKGROUND AND EXAMPLE

We start by giving a motivating example. We then give a short introduction into the IFDS framework on which FlowDroid bases its analysis and explain our attacker model.

The example code shown below reads a password from a text field (line5) 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 (line22). 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 is known to exist in real-world malware apps [32].

```
 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 }
```

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 an app restart.

Field-sensitivity is needed due to 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 a high 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. Operations such as string concatenation (line 17) require a security lattice that defines how data flows through those operations.

The IFDS framework

Our inter-procedural dataflow analysis problem is formulated in terms of the IFDS framework by Reps et al. [25] and can therefore be reduced to a graph-reachability problem. IFDS solves inter-procedural, finite, distributive subset problems and works by creating a so-called *exploded supergraph* based on flow functions associated with the program statements. A flow function defines the impact of a statement on a set of flow facts. For example, the statement $x = y$ would be associated with a 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).

Function calls are modeled using summary functions, which makes the approach precise, yet very 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 very call site (thus yielding full context sensitivity). Section 4 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 within this graph. Because all method calls have been abstracted through summary functions, those queries are fully intra-procedural and therefore highly efficient.

Attacker model

We assume an attacker that can supply an app with arbitrary malicious bytecode, obfuscated or not. In the scenario treated in this paper, the attacker's goal is to leak private data through a dangerously broad set of permissions granted by the user [4]. Our analysis 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. 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 [9], however their approach requires access to a load-time instrumentation API, which is something that Android does not currently support. We are in contact with Google to see if such an API can be supported in the future.

3. PRECISE MODELLING OF LIFECYCLE

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 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 system calls these methods at runtime to start or stop the component, or pause or resume it, depending on environment needs. It can, e.g., stop an application because of low memory, and later restart it when the user returns to it. Figure 1 shows the lifecycle for an activity. 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. However, the lifecycle is more complex than depicted and explicitly documented. There are additional methods for saving and restoring state, as well as callbacks that notify the app about additional state changes. If not taken into account, a malicious application could exploit these methods and send out secret information without being noticed.

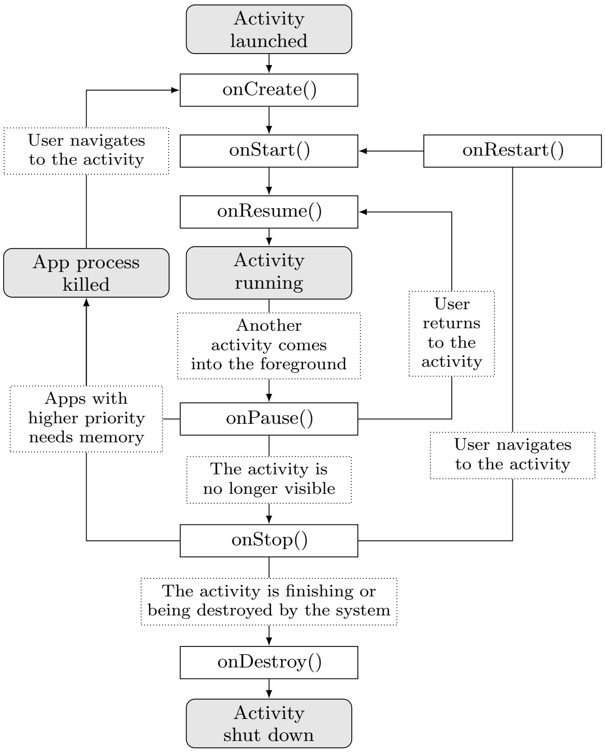

Activity launched
onCreate()
User navigates to the activity
onStart()
onRestart()
onResume()
App process killed
Activity running
User returns to the activity
Another activity comes into the foreground
Apps with higher priority needs memory
onPause()
User navigates to the activity
The activity is no longer visible
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 1: Android Activity Lifecycle

Asynchronously executing components. An application can contain multiple components, e.g., three activities and one service. The activities run sequentially, however, one cannot pre-estimate 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 in parallel. FlowDroid thus conservatively assumes that all components (activities, services, etc.) inside an application to run in an arbitrary sequential order. *Within* a component, the analysis is fully flow-sensitive, though. Additionally, the Android operating system allows applications to register callbacks for various types of information, e.g., location updates or UI interactions. Callbacks implement predefined interfaces declaring methods that are then called asynchronously by the operating system. FlowDroid models these callbacks in its dummy main method, for instance to recognize cases where an application stores to the heap the location data that is passed into 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 conservatively as-

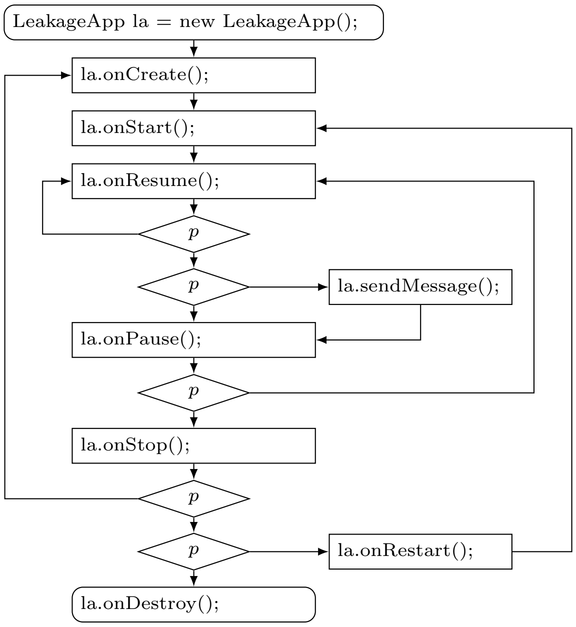


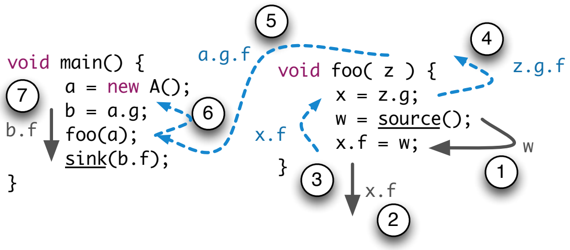
Figure 2: CFG for dummy main method

sumes that all callbacks can be invoked in any possible order. Some static analysis are path-sensitive, i.e., consider each possible program paths separately. In such cases, considering all possible orderings would be very expensive. Fortunately, FlowDroid bases its analysis on IFDS, which is *not* path-sensitive and instead joins analysis results immediately at any control-flow merge point. For FlowDroid can thus generate a main method in which every order of *individual* component lifecycles and callbacks is possible, it does *not* need to simulate all possible paths.

Example. In Figure 2 we show the control-flow graph of the dummy main method for our example. In this figure, p represents an *opaque predicate* of which we know that FlowDroid won't be able to evaluate it statically (for instance, a check involving an environment variable). In result, the analysis will automatically consider on equal terms both branches for conditions involving p. Note that, to gain maximal precision, we generate an individual dummy main method for each app analyzed. Each main method will only involve the fraction of the lifecycle that, according to the app's XML configuration files, 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. A button click handler is for instance only analyzed in the context of its respective activity.

4. CONTEXT, OBJECT, FIELD AND FLOW-SENSITIVE ANALYSIS

One major difficulty in the analysis is how to handle aliasing with high object sensitivity. Figure 3 shows how FlowDroid combines a forward-taint analysis and an on-demand backward-alias analysis to decide that `b.f` is tainted at the sink. The important step is ③ , where the forward analysis assigns a taint to `x.f`. On such assignments to the heap, the backward analysis searches upwards for aliases of `x.f`, an idea borrowed from Andromeda [27]. At ⑦ , the alias `b.f` is found and then propagated forward.



```
void main() {
    a = new A();
    b = a.g;
    foo(a);
    sink(b.f);
}

void foo( z ) {
    x = z.g;
    w = source();
    x.f = w;
}
```

Figure 3: Taint analysis under realistic 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 1: Formalization Domains

To increase the reproducibility of our approach, 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 heap model. Section 4.1 explains our general taint analysis, while Section 4.2 explains our on-demand alias analysis.

FlowDroid leverages IFDS to propagate each tainted value individually, using a multi-threaded implementation. When processing any given statement, the set $\mathbf{T}$ of incoming taints are transformed into a set of outgoing taints. We define the abstract analysis semantics with the standard notation used similarly by Tripp et al. [27]. Table 1 shows our abstract domain. For representing the effects of program statements, FlowDroid uses the semantics defined in Table 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

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 2: Program Semantics

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

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

$source(s) : Stmt \rightarrow \mathcal{P}\,(VarId)$ 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

$nativeTaint(s) : Stmt \rightarrow \mathcal{P}\,(VarId)$ 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.

$nativeAlias(s) : Stmt \rightarrow \mathcal{P}\,(VarId)$ returns possible aliases of tainted values before the native call $s \in Stmt$ (backwards analysis)

To model deep object sensitivity, FLOWDROID does not simply propagate simple fields like $x.f$ but instead so-called *access paths* [27] 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, $\mathbf{T}$ is actually a set of currently tainted access paths.

A concrete state is a program state extended by the set of tainted access paths $\mathbf{T}$ 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 4.2 explains how we use access paths to deal with aliasing.

4.1 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: normal, call, return and call-to-return.

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.

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

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 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*.

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

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 functions 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. 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 $b = c.m(a_0, \ldots, a_n)$:

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

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 5). Again consider a call $b = c.m(a_0, \ldots, a_n)$:

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

4.2 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 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 [27]. As Figure 3 shows, 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.

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

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

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

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

Call flow function. Since the backward analysis traverses the control-flow graph backwards, its call flow function propagates information from call sites to *return* sites in the callee. Consider again a call $b = c.m(a_0, \ldots, a_n)$ and a return site returning a variable r:

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

Return flow function. In the backwards analysis, return flow functions propagate from the callee's *start* point to just before the call site. The return flow function thus maps the callee's aliases at back to the caller while taking into account renaming from formal parameters p_i to actual parameters a_i. For a call site $c.m(a_0, \ldots, a_n)$ we obtain:

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

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

$$A \xrightarrow{s} \begin{cases} A \cup nativeAlias(s) & native(s) \\ & \wedge (a_i.f^m \in A \vee c.f^n \in A) \\ A \setminus \{b.f^n\} & b.f^n \in A \\ A & \text{otherwise} \end{cases}$$

5. IMPLEMENTATION

FlowDroid extends the Soot framework [16] 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 intermediate representation which allows us to implement our analysis for both targets. FlowDroid further uses Heros [8] as an implementation of IFDS on top of Soot and Dexpler. We next explain FlowDroid's architecture, while the subsequent sections explain interesting implementation details and FlowDroid's current limitations.

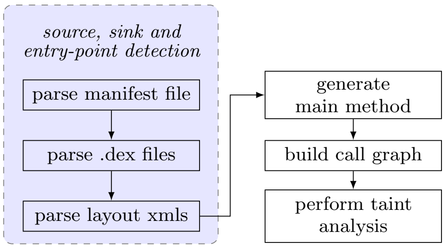


Figure 4: Overview of FlowDroid

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 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. Next, FlowDroid generates the main method from the list of lifecycle and callback methods (see the Paragraphs *Callbacks* and *Substitution Classes* for more information). 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. *Native Calls* require a special treatment which is described below along with a performance optimization called *Taint Wrapping*. At the end, FlowDroid reports all discovered flows from sources to sinks.

UI Interactions. UI elements can be taint sources, e.g., if an application prompts the user for a password and then sends it out to the Internet. 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 3), 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. Next, FlowDroid generates a customized main method for each activity, taking the respective discovered callbacks into account.

Substitution Classes. J2EE and Android applications sometimes operate on interface types or abstract class types that at runtime receive objects instantiated within the J2EE or

Android framework. FlowDroid's main-method generator cannot easily be aware of those concrete subtypes. FlowDroid thus allows the analysis designer 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 the designer aims for higher precision or scalability. For Android, substitution classes are required, for instance, for the abstract class *android.context.Context.*

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. We thus defined explicit taint propagation rules for the most common native methods, such as `arraycopy` in `java.lang.System`. In this example, third 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 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.

Taint Wrapping. Including the full JRE or Android platform runtime in the analysis requires a lot of time and memory and can lead to unwanted imprecisions. Thus, while explicit taint-propagation rules are required for native methods, they can be useful for regular Java methods as well: their use can prevent the taint propagation from having to analyze the library's internals. FlowDroid supports a simple textual file format for defining such "shortcut rules". Predefined rules handle collection classes, string buffers and similar commonly used data structures.

Limitations. 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 [9], as reflective call targets are often determined by external configuration files. On the Java platform, reflection-analysis tools such as TamiFlex [9] can be used to make static analysis tools aware of reflective calls. Such tool 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, but we plan to include them in the near future. FlowDroid also ignores probabilistic and possibilistic leaks caused by multi-threading [13].

6. EXPERIMANTAL EVALUATION

Our evaluation addresses the following research questions:

RQ1 How does FlowDroid compare to commercial taint-analysis tools for Android, both 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 [21], 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 academic Android analysis tools published elsewhere.

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 [17], at the moment there is no Android-specific analysis benchmark suite. 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 containing 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 3 contains a list of all apps in the suite together with their respective names and short descriptions. The table also indicates whether an analysis tool must model the Android lifecycle or callback infrastructure to be able to successfully analyze the app. We 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 from our project website.

Table 5 presents the analysis results for FlowDroid and two commercial analysis tools explained in the following. As the results show, FlowDroid generally performs quite well. 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 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. We plan to add better support in the future. As most known taint-analysis tools, FlowDroid currently disregards implicit flows caused through control-flow dependencies.

Comparison with IBM AppScan Source. We compared FlowDroid against 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. 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 5 shows, AppScan Source finds only about 38% of all leaks (50% if ignoring implicit flows). Major problems occur with the handling of callbacks, the Android component lifecycle and implicit flows.

Comparison with Fortify SCA. Fortify SCA [1] 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 5, Fortify SCA shows problems similar to those of IBM AppScan (cf. Section 6.1), like the handling of the Android component lifecycle, the callbacks and implicit flows. Figure 5 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 which coincidentally causes a leak to be reported. When removing the static modifier, which does not the semantics of the test case, Fortify does not detect the leak any longer.

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 higher precision with a significantly higher recall.

6.2 RQ2: Performance on InsecureBank

InsecureBank [21] 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 found all seven data leaks which we all verified by hand. There were no false positives nor false negatives.

6.3 RQ3: Real-World Applications

The above experiments give a very good indication that FlowDroid yields correct and precise results, not just for small test case but also for more realistic apps such as InsecureBank [21]. 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: 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.

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 [17] 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. For the interfaces *HTTPServletRequest* and *HTTPServletResponse* we needed to provide substitution classes (see Section 5), as we had no access to framework code that would provide their implementations. 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 4 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. This is due to an imprecision that FlowDroid shares with most other static analyses: for performance reasons, the tools 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 interesting 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.

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 4: SecuriBench Micro test results

6.5 Comparison with Other Tools

We also tried to compare FLOWDROID to a number of other tools from the scientific literature, namely TrustDroid [31], LeakMiner [30], 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 [11], 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 by the submission deadline. The authors of AndroidLeaks [12] promised to run their tool on DROIDBENCH but never delivered. We also contacted the authors of CHEX [18], but they were unable to provide the tool or any benchmark results due to intellectual property claimed by NEC. Starostin [19] declined to participate in the experiment as his tool ignores aliasing, making any comparison meaningless.

In result, we were unable to successfully evaluate even a single scientific taint-analysis tool for Android. We believe that the lack of such comparative experiments is hindering scientific progress a lot, which is why we make available our entire implementation, documentation and benchmarks as open source.

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 [18], 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. However, since CHEX looks for vulnerabilities, target applications are supposed to be benign. The derived assumptions (no obfuscation etc.) do not hold for malicious apps. Furthermore, 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 performance. CHEX's entry point model requires an enumeration of all possible *split* orderings which is not necessary in FLOWDROID.

LeakMiner [30] 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 is likely a major source of imprecision. Unfortunately we were unable to perform a systematic comparison as the authors did not respond to our inquiries.

AndroidLeaks [12] 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. As noted before, we contacted the authors but they failed to provide us any data for a comparative evaluation.

SCanDroid [11] 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 string analysis, which we leave to future work.

Other approaches like CopperDroid [24] 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. Alternatively, data leaks might only occur after a certain runtime threshold. Aurasium [28] and DroidScope [29] largely suffer from the same shortcomings with respect to static leak detection.

There are also approaches with a much broader focus like [6]. This approach not only tracks data flows, but also generate a user-friendly report and sanitizes malicious apps by replacing their sources with safe equivalents like UUID generators. The paper gives no details on the implementation, which is why we were unable to conduct a detailed comparison to our approach.

The approach by Payet et al. [22] tries to aid the developer by checking for common programming errors using different checks, for example a nullness analysis. In contrast to FLOWDROID, their approach cannot perform a dataflow analysis and is not focused on security.

8. CONCLUSIONS

We have presented FLOWDROID, a novel and highly precise static taint-analysis tool for Android applications. We have shown that many existing approaches do not adequately model Android-specific challenges like the application lifecycle or callback methods, leading to either missed leaks or false positives. 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, FLOWDROID also has a higher precision resulting in less false positives. We hope that in the future DroidBench will serve as a standard test set for Android taint analyses.

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. This work was supported by a Google Faculty Research Award, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG within the project RUNSECURE.

9. REFERENCES

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

[2] Ibm rational appscan, April 2013. `http://www-01.ibm.com/software/de/rational/appscan/`.

[3] Android. Android security overview, December 2012. `http://source.android.com/tech/security/`.

[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.-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] Bit9. Pausing google play: More than 100,000 android apps may pose security risks, November 2012. `http://www.bit9.com/pausing-google-play/`.

[8] 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.

[9] 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.

[10] International Data Corporation. Worldwide quarterly mobile phone tracker 3q12, November 2012. `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37`.

[11] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications.

[12] 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. Springer, 2012.

[13] 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.

[14] Google Inc. Permissions, December 2012. `http://developer.android.com/guide/topics/security/permissions.html`.

[15] Google Inc. Android and security, November 2012. `http://googlemobile.blogspot.de/2012/02/android-and-security.html`.

[16] 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.

[17] Ben Livshits. Securibench micro, March 2013. `http://suif.stanford.edu/~livshits/work/securibench-micro/`.

[18] 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.

[19] 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.

[20] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS '10*, pages 328–332, New York, NY, USA, 2010. ACM.

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

[22] 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.

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

[24] 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.

[25] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[26] 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.

[27] 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*, FASE'13, pages 210–225, 2013.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proc. of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107. Springer, 2011.

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.	●	○	○
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				
BroadcastReceiverLifecycle1	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 3: DroidBench 1.0 test cases

⊛ = 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: DroidBench test results