Scalable and Precise Taint Analysis for Android

Wei Huang[1], Yao Dong[2], Ana Milanova[2], and Julian Dolby[3]

[1]Google Inc., USA, huangwe@google.com

[2]Rensselaer Polytechnic Institute, USA, {dongy6,milana2}@rpi.edu

[3]IBM Research, USA, dolby@us.ibm.com

ABSTRACT

We propose a type-based taint analysis for Android. Concretely, we present DFlow, a context-sensitive information flow type system, and DroidInfer, the corresponding type inference analysis for detecting privacy leaks in Android apps. We present novel techniques for error reporting based on CFL-reachability, as well as novel techniques for handling of Android-specific features, including libraries, multiple entry points and callbacks, and inter-component communication. Empirical results show that our approach is scalable and precise. DroidInfer scales well in terms of time and memory and has false-positive rate of 15.7%. It detects privacy leaks in apps from the Google Play Store and in known malware.

Categories and Subject Descriptors

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

Keywords

Taint analysis, Android, information flow, CFL-reachability

1. INTRODUCTION

Android is the most popular platform on mobile devices. As of the second quater of 2014, Android has reached 84.4% share of the global smartphone market [19]. Android's success is partly due to the enormous number of applications available at the Google Play Store, as well as other third-party app stores. However, Android apps often collect sensitive data such as location and phone state, usually for the purpose of tracking and targeted advertising.

In this paper we consider a threat model where an app, legitimate or malicious, obtains sensitive data and leaks this data to either logs or the network. Logs are an issue, because until Android 4.0 any app that held the READ_LOGS permission could read all logs. We track *log flows*, but we emphasize *network flows* (e.g., flows of the device identifier to the Internet through an HTTP request), which present a more pertinent and challenging problem.

Taint analysis detects flows from sensitive data *sources* (e.g., location, phone state) to untrusted *sinks* (e.g., logs, the Internet). Many researchers have tackled taint analysis for Android. Dynamic analyses such as Google Bouncer [10], TaintDroid [4], DroidScope [41], CopperDroid [32], and Aurasium [40] instrument the app bytecode and/or use customized execution environment to monitor the transition of sensitive data. Unfortunately, dynamic analysis slows execution and typically lacks code coverage.

Static taint analysis detects privacy leaks without running the app. There has been considerable effort on static taint analysis, with the majority of work focusing on dataflow and points-to-based approaches [24, 42, 20, 9, 22, 7, 1]. Yet a solution remains elusive.

FlowDroid, a highly-precise, context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis for Android [1], is the state-of-the-art. Unfortunately, FlowDroid is computationaly- and memory-intensive. Further, while it reports numerous log flows in apps from the Google Play Store, it reports no network flows [1]. This is surprising, given the knowledge that apps track their users pervasively.

We propose *type-based* taint analysis for Android leveraging previous work on type-based taint analysis for web applications [15]. Our approach is *modular* and *compositional*. It can analyze any given set of classes. Modular analysis is particularly suitable for Android apps because 1) the Android app is an "open" program with multiple entry points through callbacks, and 2) it uses large libraries that can be suitably handled with conservative defaults. The analysis requires annotations only on *sources* and *sinks*. Once the sources and sinks are built into annotated libraries, Android apps are analyzed *without* any input from the user.

Concretely, we propose DFlow, a context-sensitive information flow type system and DroidInfer, the corresponding type inference analysis. DroidInfer is as precision as, but much more scalable than FlowDroid. DroidInfer is lightweight and runs in ≈ 2 minutes on average, within a memory footprint of 2GB. It uncovers numerous network flows in apps from the Google Play Store and in known malware. DroidInfer posts an F-measure of 0.88 on DroidBench [7], the standard for evaluating static taint analysis for Android.

DroidInfer scales because it completely avoids points-to analysis. It is precise because in essence it is a CFL-reachability computation, a highly-precise analysis technique [33]. An important contribution of our work is that it *explains* source-sink flows *intuitively* in terms of CFL-reachability paths.

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA

```
1  public class WallpapersMain extends Activity {
2    private String BASE_URL, deviceId;
3    private int pageNum, catId;
4    private DisplayMetrics metrics;
5    private WebView browser1;
6    protected void onCreate(Bundle b) {
7      start();
8    }
9    protected void onActivityResult(int rq, int rs, Intent i){
10     navigate();
11   }
12   private void start() {
13     BASE_URL = "getWallpapers_Android2/";
14     TelephonyManager mgr =
15      (TelephonyManager)this.getSystemService("phone");
16     deviceId = mgr.getDeviceId(); // source
17   }
18   private void navigate() {
19     String str = BASE_URL + pageNum + "/" + catId + "/"
           + deviceId + "/" + metrics.widthPixels + "/" +
           metrics.heightPixels;
20     browser1.loadUrl(str); // sink
21   }
22 }
```

Figure 1: WallpapersMain **leaks the device identifier (the source at line 16) to a content server (the sink at line 20) in a URL.**

This paper makes the following contributions:

- DFlow, a context-sensitive information flow type system and DroidInfer, the corresponding worst-case cubic inference analysis which amounts to CFL-reachability. DroidInfer works on Application Package files (APKs).
- Effective handling of Android-specific features: "open" programs with multiple entry points, callbacks and large libraries, and a technique that improves precision in the handling of inter-component communication.
- An extensive empirical evaluation on three sets of Android apps: 1) DroidBench [7, 1], 2) 22 malware apps from the Contagio website [25] and 3) 144 apps from the Google Play Store [11]. DroidInfer achieves the same F-measure as FlowDroid on DroidBench. It uncovers all network flows in the Contagio apps, as well as numerous network flows across 40 Play Store apps.

The rest of the paper is organized as follows. Sect. 2 gives a motivating example and a brief discussion of our type-based approach. Sect. 3 presents the DFlow type system and the inference analysis. Sect. 4 outlines the connection with CFL-reachability and the reporting technique. Sect. 5 describes the handling of Android-specific features. Sect. 6 presents the empirical evaluation. Sect. 7 discusses related work and Sect. 8 outlines the limitations and concludes.

2. OVERVIEW

We begin with a motivating example that shows a privacy leak in an Android app and proceed to outline our approach.

2.1 Motivating Example

The example shown in Fig. 1 is refactored from one of our benchmarks, **Backgrounds HD Wallpapers** version 2.0.1 from the Google Play Store. The WallpapersMain activity first obtains the device identifier by calling the getDeviceId method and stores it into a field deviceId when it is created (onCreate). Then it appends the deviceId into a search URL url, which is sent to a content server in the navigate method. Finally, the navigate method is called in callback method onActivityResult, resulting in a privacy leak.

This example illustrates several challenges. Unlike Java programs, Android apps do not have a single entry point. Instead, each callback method is a potential entry point as it could be called by the Android framework. In WallpapersMain, both onCreate and onActivityResult are callback methods that are implicitly called by the Android framework.

Multiple entry points and callbacks by the framework are a significant challenge to traditional points-to-based static analyses, which usually require whole-program analysis.

2.2 Type Qualifiers

In our type-based approach, each variable is typed by a *type qualifier*. There are two basic qualifiers in DFlow: tainted and safe.

- tainted: A variable x is tainted, if there is flow from a sensitive source to x. In the WallpapersMain example, the return value of TelephonyManager.getDeviceId is typed as tainted.
- safe: A variable x is safe if there is flow from x to an untrusted sink. For example, the parameter url of WebView.loadUrl(String url) is a safe sink.

Note that our analysis for Android is actually a *confidentiality analysis*. We keep the term *taint analysis* and qualifiers tainted and safe only in deference to previous work [4, 7].

In order to disallow flow from tainted sources to safe sinks, DFlow enforces the following subtyping hierarchy:

safe <: tainted[1]

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2. (q is also a subtype of itself $q <: q$.) Therefore, it is allowed to assign a safe variable to a tainted one:

```
safe String s = ...;
tainted String t = s;
```

However, it is not allowed to assign a tainted variable to a safe one:

```
tainted String t = ...;
safe String s = t; // type error!
```

In the WallpapersMain example, the return value of getDeviceId is typed as tainted and the url parameter of loadUrl is typed as safe, as they are a source and a sink, respectively. The field deviceId is tainted and so is the local variable str since it contains the value of deviceId. Because it is not allowed to assign a tainted str to the safe parameter of loadUrl, the program results in a type error, signaling the leak.

Once the sources and sinks are given, type qualifiers are inferred automatically by our inference tool (Sect. 3.2). If there is a valid typing, then there is no flow from a source to a sink. Otherwise, the tool reports *type errors*, signaling potential privacy leaks.

A longstanding issue with type inference is *explaining* type errors [21, 39]. In general, the inference tool can issue a type error anywhere along the long flow path from source to sink!

---

[1] This is the desired subtyping. However, it is not safe when *mutable* references are involved [29, 35].

We map each type error into intuitive, humanly-readable CFL-reachability flow paths (Sect. 4). For example, the type error in Fig. 1 (roughly) maps to

$$\mathit{source} \xrightarrow{[_{\mathsf{deviceId}}} \mathsf{this}_{\mathsf{start}} \rightarrow \mathsf{this}_{\mathsf{navigate}} \xrightarrow{]_{\mathsf{deviceId}}} \mathsf{str} \rightarrow \mathit{sink}$$

meaning that the source flows into field deviceId of implicit parameter this of start, which in turn flows into this of navigate, where field deviceId is read into str, which flows to sink.

The problem is not limited to type inference. Any static analysis (e.g., [1]) faces the issue of error reporting and there are no satisfying solutions at this point. We believe that our approach is a significant step forward.

2.3 Context Sensitivity

DFlow achieves context sensitivity by using a polymorphic type qualifier, poly, and *viewpoint adaptation* [3].

- poly: poly is interpreted as tainted in some contexts and as safe in other contexts.

The subtyping hierarchy becomes

$$\mathsf{safe} <: \mathsf{poly} <: \mathsf{tainted}$$

The concrete value of poly is interpreted by the viewpoint adaptation operation. Viewpoint adaptation of a type q' from the viewpoint of another type q, results in the adapted type q''. This is written as $q \rhd q' = q''$. Viewpoint adaptation adapts fields, formal parameters, and method return values from the viewpoint of the *context* at the field access or method call. DFlow defines viewpoint adaptation below:

$$\begin{array}{cccc} \_ & \rhd & \mathsf{tainted} & = & \mathsf{tainted} \\ \_ & \rhd & \mathsf{safe} & = & \mathsf{safe} \\ q & \rhd & \mathsf{poly} & = & q \end{array}$$

The underscore denotes a "don't care" value. Qualifiers tainted and safe do not depend on the viewpoint (context). Qualifier poly depends on the viewpoint: e.g., if the viewpoint (context) is tainted, then poly is interpreted as tainted.

The type of a poly field f is interpreted from the viewpoint of the *receiver* at the field access. If the receiver x is tainted, then x.f is tainted. If the receiver x is safe, then x.f is safe. The type of a poly parameter or return value is interpreted from the viewpoint of q^i, the context at the method call. Consider the example in Fig. 2, where method id is typed as follows (code throughout the paper makes parameter this explicit when necessary):

```
poly String id(tainted Util this, poly String p)
```

This enables context sensitivity because id can take as input a tainted String as well as a safe one. poly is interpreted as tainted at callsite 10, and as safe at callsite 11.

3. TYPE SYSTEM

In this section, we define the DFlow type system and present the type inference technique.

3.1 Typing Rules

We define our typing rules over a syntax in "named form" where the results of field accesses, method calls, and instantiations are immediately stored in a variable. For space reasons, we leave the syntax implicit in the typing rules; it is specified precisely in the accompanying technical report [16].

Without loss of generality, we assume that methods have parameter this, and exactly one other formal parameter. The DFlow type system is *orthogonal* to (i.e. independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

```
class Util {
  poly String id(tainted Util this, poly String p) {
    return p;
  }
}
...
Util y = new Util();
tainted String src = ...;
safe String sink = ...;
tainted String srcId = y.id^10(src);
safe String sinkId = y.id^11(sink);
```

Figure 2: Context sensitivity example.

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\textsc{tnew}) \qquad \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\textsc{tassign})$$

$$\frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}\ (\textsc{twrite})$$

$$\frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_{\mathsf{p}} \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} <: q^i \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} <: q^i \rhd q_{\mathsf{p}} \quad q^i \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}\ (\textsc{tcall})$$

Figure 3: Typing rules. Function *typeof* retrieves the DFlow types of fields and methods, Γ is a type environment that maps variables to DFlow qualifiers. q^i is the context of adaptation at call site i.

The typing rules are defined in Fig. 3. Rules (TNEW) and (TASSIGN) enforce the expected subtyping constraints. The rules for field access, (TREAD) and (TWRITE), adapt field f from the viewpoint of *receiver* y and then enforce the subtyping constraints. Recall that the type of a poly field f is interpreted in the context of the receiver y. If the receiver y is tainted, then y.f is tainted. If the receiver y is safe, then y.f is safe.

The rule for method call, (TCALL), adapts formal parameters this and p and return value ret from the viewpoint of *callsite context* q^i, and enforces the subtyping constraints that capture flows from actual arguments to formal parameters, and from return value to the left-hand-side of the call assignment.

The callsite context q^i is a value that is not important, except that it should exist. It can be any of {tainted, poly, safe}. Consider the example in Fig. 2. At callsite 10, q^{10} is tainted and $q^{10} \rhd$ poly is interpreted as tainted. The following constraints generated at callsite 10 are satisfied[2]:

$$\mathsf{y} <: q^{10} \rhd \mathsf{tainted} \quad \mathsf{src} <: q^{10} \rhd \mathsf{poly} \quad q^{10} \rhd \mathsf{poly} <: \mathsf{tainted}$$

At callsite 11, q^{11} is safe and $q^{11} \rhd$ poly is interpreted as safe.

---

[2] For brevity and clarity, we omit q when dealing with variables from code examples, i.e., we write y instead of q_{y}.

Therefore, the constraints at callsite 11 are satisfied:

$$\mathsf{y} <: q^{11} \rhd \mathsf{tainted} \quad \mathsf{sink} <: q^{11} \rhd \mathsf{poly} \quad q^{11} \rhd \mathsf{poly} <: \mathsf{safe}$$

We compose DFlow with ReIm, a reference immutability type system [17]. This is necessary to overcome known issues with subtyping in the presence of mutable references [29, 35]. Specifically, if the left-hand-side of an assignment (explicit or implicit) is *immutable* according to ReIm, we enforce a subtyping constraint; otherwise, we enforce an equality constraint. For example, at (TASSIGN) $\mathsf{x} = \mathsf{y}$, if x is immutable, i.e. x is not used to modify the referenced object, we enforce $q_{\mathsf{y}} <: q_{\mathsf{x}}$; otherwise, we enforce $q_{\mathsf{y}} = q_{\mathsf{x}}$. The more variables are proven immutable, the more subtyping constraints there are, and hence, the more precise DFlow is [26].

Method overriding is handled by the standard constraints for function subtyping. If n overrides m, we require *typeof*(n) <: *typeof*(m) and thus

$$(q_{\mathsf{this_n}}, q_{\mathsf{p_n}} \rightarrow q_{\mathsf{ret_n}}) \;<:\; (q_{\mathsf{this_m}}, q_{\mathsf{p_m}} \rightarrow q_{\mathsf{ret_m}})$$

This entails $q_{\mathsf{this_m}} <: q_{\mathsf{this_n}}$, $q_{\mathsf{p_m}} <: q_{\mathsf{p_n}}$, and $q_{\mathsf{ret_n}} <: q_{\mathsf{ret_m}}$. Soundness of DFlow is argued as in [14].

3.2 Type Inference

Given sources and sinks, type inference derives a *valid typing*, i.e. an assignment from program variables to type qualifiers that type checks with the typing rules in Fig. 3. If type inference succeeds, then there are no leaks from sources to sinks. If it fails the app may contain leaks.

Type inference first computes a *set-based solution* S, which maps variables to sets of potential type qualifiers. Then it uses *method summary constraints*, a technique that refines the set-based solution and helps derive a valid typing.

3.2.1 Set-based Solution

The set-based solution is a mapping S from variables to sets of qualifiers. For instance, if $S(\mathsf{x}) = \{\mathsf{tainted}, \mathsf{poly}\}$, that means variable x can be $\mathsf{tainted}$ or poly, but not safe. Programmer-annotated variables, including sources and sinks, are initialized to the singleton set that contains the provided type qualifier. For example, sources and sinks from the annotated library map to $\{\mathsf{tainted}\}$ and $\{\mathsf{safe}\}$, respectively. Fields f are initialized to $S(\mathsf{f}) = \{\mathsf{tainted}, \mathsf{poly}\}$; we forgo safe fields, which makes the inference converge faster. All other variables and callsite contexts q^i are initialized to the maximal set of qualifiers, i.e. $S(\mathsf{x}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$.

The inference then creates constraints for all program statements according to the typing rules in Fig. 3. It takes into account the mutability of the left-hand-side of assignments as discussed in Sect. 3. Then the *set-based solver* iterates over constraints c and calls SOLVECONSTRAINT(c). SOLVECONSTRAINT(c) removes *infeasible* qualifiers from the set of variables in c [13]. Consider constraint c: $q_{\mathsf{y}} <: q_{\mathsf{x}}$ where $S(\mathsf{y}) = \{\mathsf{tainted}\}$ and $S(\mathsf{x}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$ before solving the constraint. The solver removes poly and safe from $S(\mathsf{x})$, because there does not exist a $q_{\mathsf{y}} \in S(\mathsf{y})$ that satisfies $q_{\mathsf{y}} <: \mathsf{poly}$ and $q_{\mathsf{y}} <: \mathsf{safe}$. In the case that the infeasible qualifier is the last element in $S(\mathsf{x})$, the solver reports a *type error*. For example, $\mathsf{y}\{\mathsf{tainted}\} <: \mathsf{x}\{\mathsf{safe}\}$ is a type error because it is not satisfiable.

The solver keeps removing infeasible qualifiers for each constraint until it reaches a fixpoint. If there are type errors, this indicates potential flows from sources to sinks.

```
procedure SUMMARYSOLVER
  repeat
    for each $c$ in $C$ do
      SOLVECONSTRAINT($c$)
      if $c$ is $q_x <: q_y \rhd q_f$ and $S(f)$ is {poly} then          ▷ Case 1
        Add $q_x <: q_y$ into $C$
      else if $c$ is $q_x \rhd q_f <: q_y$ and $S(f)$ is {poly} then    ▷ Case 2
        Add $q_x <: q_y$ into $C$
      else if $c$ is $q_x <: q_y$ then                                  ▷ Case 3
        for each $q_y <: q_z$ in $C$ do Add $q_x <: q_z$ to $C$ end for
        for each $q_w <: q_x$ in $C$ do Add $q_w <: q_y$ to $C$ end for
        for each $q_w <: q^i \rhd q_x$ and $q^i \rhd q_y <: q_z$ in $C$ do ▷ Case 4
          Add $q_w <: q_z$ to $C$
        end for
      end if
    end for
  until $S$ remains unchanged
end procedure
```

Figure 4: Initially, S is the result of the set-based solution and C is the set of constraints for program statements. See [16] for details.

3.2.2 Valid Typing

Unfortunately, even if the set-based solver terminates without type errors, a valid typing still may not exist. That is, there still may be undiscovered flows from sources to sinks.

We adapt *method summary constraints*, a technique that removes additional infeasible qualifiers, and helps arrive at a valid typing or uncover additional type errors. The algorithm, adapted from [15] to DFlow, is shown in Fig. 4.

The method summary constraints are constraints that "connect" parameters to return values. Recall the id example in Fig. 2. $\mathsf{p} <: \mathsf{ret}$ is a method summary constraint reflecting the flow from the parameter p to the return value ret.

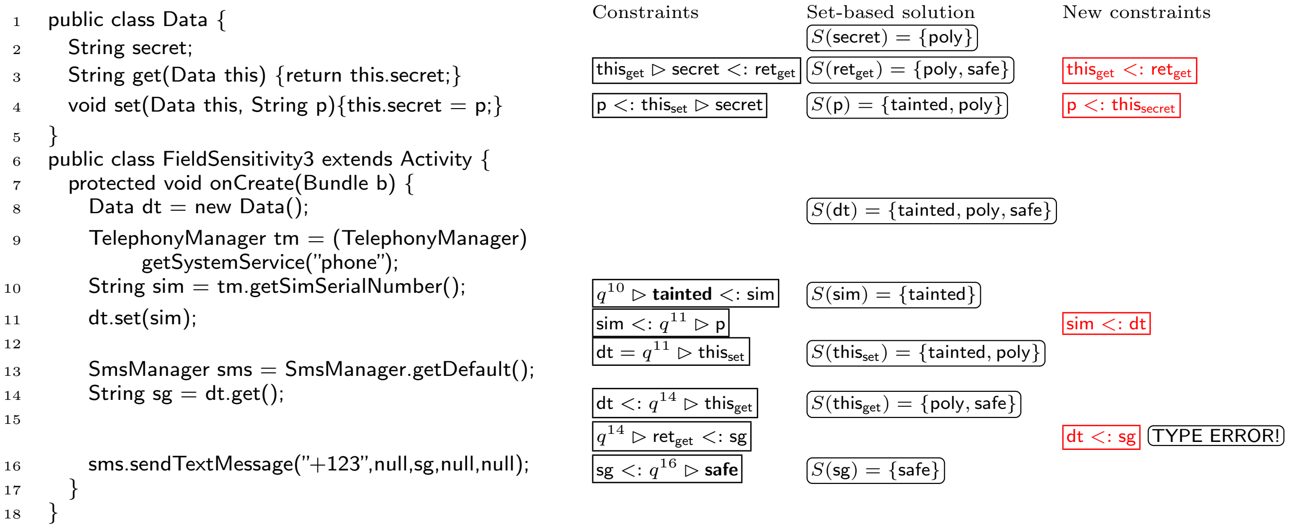
In many cases however, the flow from formal parameters to return values is "connected" indirectly. For example, the parameter p and the return value ret can be connected through two constraints: $q_{\mathsf{p}} <: q_{\mathsf{x}}$ and $q_{\mathsf{x}} <: q_{\mathsf{ret}}$. Due to transitivity, we have $q_{\mathsf{p}} <: q_{\mathsf{ret}}$. The algorithm "searches" for a subtyping chain from the formal parameter (including this) to the return value of method m (Cases 1, 2, and 3 in Fig. 4). It uses the method summary constraints to "connect" the actual argument and the left-hand-side of the call assignment at calls to m (Case 4).

Consider again the id method in Fig. 2. We have $\mathsf{p} <: \mathsf{ret}$ due to the return statement $\mathsf{return\ p}$. The inference adds constraints between actual arguments and left-hand-sides at callsites 10 and 11. First, $\mathsf{p} <: \mathsf{ret}$ implies $q^{10} \rhd \mathsf{p} <: q^{10} \rhd \mathsf{ret}$. This constraint and the constraints at callsite 10

$$\mathsf{src} <: q^{10} \rhd \mathsf{p} \quad <: \quad q^{10} \rhd \mathsf{ret} <: \mathsf{srcId}$$

entail $\mathsf{src} <: \mathsf{srcId}$. The inference adds $\mathsf{src} <: \mathsf{srcId}$, connecting the actual argument src and the left-hand side srcId at callsite 10. Similarly, the inference adds $\mathsf{sink} <: \mathsf{sinkId}$ at callsite 11. Such new constraints remove additional infeasible qualifiers, and help arrive at a valid typing or uncover new type errors.

When SUMMARYSOLVER (Fig. 4) terminates without type errors, the inference derives a concrete typing by picking up the *maximal* element of $S(\mathsf{x})$ according to the ranking $\mathsf{tainted} > \mathsf{poly} > \mathsf{safe}$. Such *maximal typing* almost always type-checks, which guarantees that there is no unsafe flow from sources to sinks. Even in the rare cases when the maximal typing does not type check, there is no unsafe flow [27]. In contrast, the maximal typing derived from the set-based solution before running SUMMARYSOLVER, usually does not type check. SUMMARYSOLVER is $O(n^3)$ [14].



```
1  public class Data {
2    String secret;
3    String get(Data this) {return this.secret;}
4    void set(Data this, String p){this.secret = p;}
5  }
6  public class FieldSensitivity3 extends Activity {
7    protected void onCreate(Bundle b) {
8      Data dt = new Data();
9      TelephonyManager tm = (TelephonyManager)
            getSystemService("phone");
10     String sim = tm.getSimSerialNumber();
11     dt.set(sim);
12
13     SmsManager sms = SmsManager.getDefault();
14     String sg = dt.get();
15
16     sms.sendTextMessage("+123",null,sg,null,null);
17   }
18 }
```

	Constraints	Set-based solution	New constraints
2		$S(\mathsf{secret}) = \{\mathsf{poly}\}$	
3	$\mathsf{this}_{\mathsf{get}} \rhd \mathsf{secret} <: \mathsf{ret}_{\mathsf{get}}$	$S(\mathsf{ret}_{\mathsf{get}}) = \{\mathsf{poly}, \mathsf{safe}\}$	$\mathsf{this}_{\mathsf{get}} <: \mathsf{ret}_{\mathsf{get}}$
4	$\mathsf{p} <: \mathsf{this}_{\mathsf{set}} \rhd \mathsf{secret}$	$S(\mathsf{p}) = \{\mathsf{tainted}, \mathsf{poly}\}$	$\mathsf{p} <: \mathsf{this}_{\mathsf{secret}}$
8		$S(\mathsf{dt}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$	
10	$q^{10} \rhd \mathbf{tainted} <: \mathsf{sim}$	$S(\mathsf{sim}) = \{\mathsf{tainted}\}$	
11	$\mathsf{sim} <: q^{11} \rhd \mathsf{p}$		$\mathsf{sim} <: \mathsf{dt}$
12	$\mathsf{dt} = q^{11} \rhd \mathsf{this}_{\mathsf{set}}$	$S(\mathsf{this}_{\mathsf{set}}) = \{\mathsf{tainted}, \mathsf{poly}\}$	
14	$\mathsf{dt} <: q^{14} \rhd \mathsf{this}_{\mathsf{get}}$	$S(\mathsf{this}_{\mathsf{get}}) = \{\mathsf{poly}, \mathsf{safe}\}$	
15	$q^{14} \rhd \mathsf{ret}_{\mathsf{get}} <: \mathsf{sg}$		$\mathsf{dt} <: \mathsf{sg}$ TYPE ERROR!
16	$\mathsf{sg} <: q^{16} \rhd \mathbf{safe}$	$S(\mathsf{sg}) = \{\mathsf{safe}\}$	

Figure 5: FieldSensitivity3 example refactored from DroidBench. The frame boxes beside each statement show the corresponding constraints the statement generates. We omitted uninteresting constraints. The oval boxes show propagation during the set-based solution. 16 forces sg to be $\{\mathsf{safe}\}$, then 14 forces $\mathsf{ret}_{\mathsf{get}}$ to be $\{\mathsf{poly}, \mathsf{safe}\}$ and then 3 forces $\mathsf{this}_{\mathsf{get}}$ to be $\{\mathsf{poly}, \mathsf{safe}\}$ and secret to be $\{\mathsf{poly}\}$ (recall that fields are initialized to $\{\mathsf{tainted}, \mathsf{poly}\}$, Sect. 3.2.1). 10 forces sim to be $\{\mathsf{tainted}\}$, which in turn forces the parameters p and $\mathsf{this}_{\mathsf{set}}$ to be $\{\mathsf{tainted}, \mathsf{poly}\}$. There are no type errors in the initial set-based solution. The red frame boxes in the fourth column (New constraints) show the constraints due to SummarySolver. Since field secret is poly, constraint $\mathsf{this}_{\mathsf{get}} \rhd \mathsf{secret} <: \mathsf{ret}_{\mathsf{get}}$ leads to method summary constraint $\mathsf{this}_{\mathsf{get}} <: \mathsf{ret}_{\mathsf{get}}$, which in turn leads to $\mathsf{dt} <: \mathsf{sg}$ due to the call at 14. Similarly, $\mathsf{p} <: \mathsf{this}_{\mathsf{set}} \rhd \mathsf{secret}$ leads to $\mathsf{p} <: \mathsf{this}_{\mathsf{set}}$, which in turn leads to $\mathsf{sim} <: \mathsf{dt}$ due to the call at 11. Since sim is $\{\mathsf{tainted}\}$ and sg is $\{\mathsf{safe}\}$, these constraints cause a TYPE ERROR, detecting the leak.

DroidInfer is fully context-sensitive in the call-transmitted dependences (i.e., it uses infinitely deep context). It approximates in the field-transmitted dependences by defaulting to field insensitivity in some cases (see [27] for details). DroidInfer remains precise for two reasons: (1) Android apps rarely trigger default to field insensitivity, and (2) even when they do trigger default, CFL-Explain (described in the following section) restores fields sensitivity.

3.2.3 *Example*

Let us consider the FieldSensitivity3 example refactored from DroidBench [7] in Fig. 5. The return of TelephonyManager.getSimSerialNumber (line 10) is a source and the parameter msg of SmsManager.sendTextMessage (line 16) is a sink. The serial number of the SIM card is obtained and stored into a Data object. Later, it is retrieved from the Data object and sent out through an SMS message without user consent. Fig. 5 demonstrates the analysis.

4. EXPLAINING TYPE ERRORS

Type inference produces type errors whenever there may be flow from a source to a sink. Unfortunately, type errors by themselves are rarely useful. For example, DroidInfer produces the following type error at Statement 10 in Fig. 5:

$$q^{10} \rhd \mathsf{ret}_{\mathsf{getSimSerialNumber}}\{\mathsf{tainted}\} <: \mathsf{sim}\{\mathsf{safe}\}$$

meaning that the right-hand-side of the call assignment is tainted while the left-hand-side is inferred safe. The challenge is to map each type error into a concise and intuitive source-sink path that explains the error.

In recent work [27], we studied the connection between DFlow/DroidInfer and CFL-reachability [33, 6]. The key idea is that the type constraints in Fig. 3 correspond to edges in an annotated dependence graph, and that type inference amounts to CFL-reachability computation over the graph.

Field access constraints correspond to field-annotated edges (those constraints account for structure-transmitted dependences in Reps' terminology [34]). In the example in Fig. 5, the field read return this.secret and its DFlow constraint $\mathsf{this}_{\mathsf{get}} \rhd \mathsf{secret} <: \mathsf{ret}_{\mathsf{get}}$ correspond to edge

$$\mathsf{this}_{\mathsf{get}} \xrightarrow{]_{\mathsf{secret}}} \mathsf{ret}_{\mathsf{get}}$$

As it is standard in CFL-reachability, the open bracket $[_{\mathsf{f}}$ denotes a *write* to field f, and the close bracket $]_{\mathsf{f}}$ denotes a *read* of f. Similarly, callsite constraints correspond to callsite-annotated edges (those account for call-transmitted dependences). In the example in Fig. 5, callsite 14 gives rise to the following constraints:

$$\mathsf{dt} <: q^{14} \rhd \mathsf{this}_{\mathsf{get}} \quad q^{14} \rhd \mathsf{ret}_{\mathsf{get}} <: \mathsf{sg}$$

which correspond to the following edges:

$$\mathsf{dt} \xrightarrow{(_{14}} \mathsf{this}_{\mathsf{get}} \quad \mathsf{ret}_{\mathsf{get}} \xrightarrow{)_{14}} \mathsf{sg}$$

Again standard in CFL-reachabililty, the open parenthesis $(_i$ denotes a call at callsite i, and the closed parenthesis $)_i$ denotes a return at callsite i.

The constraints in Fig. 5 give rise to source-sink path

$$source \rightarrow \mathsf{sim} \xrightarrow{(_{11}} \mathsf{p} \xrightarrow{[_{\mathsf{secret}}} \mathsf{this}_{\mathsf{set}} \xrightarrow{)_{11}} \mathsf{dt} \xrightarrow{(_{14}} \mathsf{this}_{\mathsf{get}} \xrightarrow{]_{\mathsf{secret}}} \mathsf{ret} \xrightarrow{)_{14}} \mathsf{sg} \rightarrow sink$$

which gives an intuitive explanation of the type error at the beginning of this section: the source flows into local variable sim, which in turn flows to formal parameter p at callsite 11, where in turn p is written into field secret of this, etc. Perhaps the only unintuitive part is the inverse edge $\mathsf{this}_{\mathsf{set}} \overset{)_{11}}{\rightarrow} \mathsf{dt}$ (naturally, the flow at callsite 11 is from dt to $\mathsf{this}_{\mathsf{set}}$). This edge is due to the mutation of $\mathsf{this}_{\mathsf{set}}$, which amounts to a *return* from set at 11.

Let $L(F)$ denote the context-free language of balanced open and closed brackets, and let $L(C)$ denote the analogous language of balanced open and closed parentheses. For example, string $[_{\mathsf{f}}\ ]_{\mathsf{f}}\ [_{\mathsf{g}}\ ]_{\mathsf{g}}$ is in $L(F)$ but $[_{\mathsf{f}}\ ]_{\mathsf{g}}$ is not in $L(F)$. For precise treatment, we refer the interested reader to [27]. A *feasible* source-sink path is a path where the field string belongs to $L(F)$ and the call string belongs to $L(C)$. The above path is feasible because its field string $[_{\mathsf{secret}}\ ]_{\mathsf{secret}} \in L(F)$ and its call string $(_{11}\ )_{11}\ (_{14}\ )_{14} \in L(C)$. Our goal is to map each type error to one or more feasible source-sink paths.

We run DroidInfer with the option that pushes type errors towards sources. (This can be done with a prioritization of the constraints in SummarySolver in Fig. 4.) The result is that when DroidInfer terminates, the safe sinks have affected the set-based solution of *each* variable that flows to a sink. More precisely, if x flows to a sink, then $\mathsf{tainted} \notin S(\mathsf{x})$. Thus, we can construct the dependency graph from the constraints for program statements, *omitting* all nodes whose set-based solution contains tainted. The resulting graph can be viewed as a backward slice that excludes the parts of the program unaffected by the sinks. This significantly reduces the size of the dependency graph and renders CFL-reachability reasoning practical.

For each type error, we run CFL-explain, which prints feasible paths from the source at the type error, to all reachable sinks. CFL-Explain, a breadth-first-search (BFS) augmented with CFL-reachability, is described in detail in Fig. 6. Note that one must restrict the keys of map M to ensure termination. Currently we distinguish keys by the last two open parentheses and the last two open brackets. This means that if CFL-Explain has recorded in M a path to x with a call string, say, that ends at $(_i\ (_j$, and it later arrives at a different path to x, whose call string also ends at $(_i\ (_j$, the latter path will not be recorded.

Continuing with the example in Fig. 5, CFL-Explain takes as input sim, and produces the source-sink path:

$$\mathsf{sim} \overset{(_{11}}{\rightarrow} \mathsf{p} \overset{[_{\mathsf{secret}}}{\rightarrow} \mathsf{this}_{\mathsf{set}} \overset{)_{11}}{\rightarrow} \mathsf{dt} \overset{(_{14}}{\rightarrow} \mathsf{this}_{\mathsf{get}} \overset{]_{\mathsf{secret}}}{\rightarrow} \mathsf{ret} \overset{)_{14}}{\rightarrow} \mathsf{sg} \rightarrow \mathit{sink}$$

Type inference and CFL-reachability inherently provide a *data-flow guarantee* but lack a *control-flow guarantee*. In other words, in order for the flow from source to sink to happen, control must reach the statements on the path in the particular order specified by the path. But does control reach the path? DroidInfer takes as input the entire APK and infers types and source-sink paths across the entire APK, even though some classes and methods may be unreachable.

To provide a (degree of) control-flow guarantee, we incorporate a conservative call graph. Concretely, line 9 in Fig. 6 ensures that the target node m appears in a method, which is live in the call graph CG.

CFL-Explain can refute a type error reported by DroidInfer for one of two reasons: 1) one or more methods on the path from source to sink is unreachable on the call graph and 2) the type error is a false positive due to field insensitivity (see [27]), and CFL-Explain cannot confirm a feasible path.

```
 1: procedure CFL-Explain
 2:   Add $\langle start, \epsilon, \epsilon \rangle$ to $Q$
 3:   Add $\langle start, \epsilon, \epsilon \rangle \rightarrow []$ to $M$
 4:   while $Q$ is not empty do
 5:     dequeue next node $\langle n, f, c \rangle$ from $Q$
 6:     if $n$ is a sink node then
 7:       print the path in $M$ associated with $\langle n, f, c \rangle$
 8:     end if
 9:     for each edge $e = \langle n, m, f', c' \rangle$ s.t. $Method(m) \in CG$ do
10:       Let $p$ be the path in $M$ associated with $\langle n, f, c \rangle$
11:       Let $p'$ be the path formed by appending $e$ to $p$
12:       if $f{+}f' \in L(F) \wedge c{+}c' \in L(C) \wedge \langle m, f{+}f', c{+}c' \rangle \notin M$ then
13:         Add $\langle m, f{+}f', c{+}c' \rangle \rightarrow p'$ to $M$
14:         Add $\langle m, f{+}f', c{+}c' \rangle$ to $Q$
15:       end if
16:     end for
17:   end while
18: end procedure
```

Figure 6: CFL-Explain is a BFS augmented with CFL-reachability. M maps graph nodes n, augmented with field-access strings $f \in L(F)$ and call strings $c \in L(C)$, to paths in the graph. f' is a field write $[_{\mathsf{f}}$, a field read $]_{\mathsf{f}}$ or ϵ. Similarly, c' is $(_i$, $)_i$ or ϵ. For each edge, f' or c' is empty (e.g., $e = (\mathsf{this}_{\mathsf{get}}, \mathsf{ret},]_{\mathsf{secret}}, \epsilon)$). CG is a precomputed call graph. $Method(m)$ gives the enclosing method of m.

5. ANDROID-SPECIFIC FEATURES

In this section, we discuss our techniques for handling Android-specific features, including libraries, multiple entry points and callbacks, and inter-component communication.

5.1 Libraries

Libraries are ubiquitous in Android apps. An effective analysis should keep track of flows through library method calls. Unfortunately, analyzing the Android library is a significant challenge. Computing safe summaries for the Android library is an open problem (to the best of our knowledge). Analyzing library calls on-demand, i.e., using some form of reachability analysis faces challenges due to callbacks and reflection, which are pervasive in Android. The most popular solution appears to be manual summaries for common library methods [20, 7], which is clearly unsatisfying.

DroidInfer inserts annotations (type qualifiers) into the Android library for sources (e.g. location access, phone state) and for sinks (e.g., Internet access) by using the Stub Generation Tool and the Annotation File Utility from the Checker Framework [31]. DroidInfer *uses conservative defaults for all unknown library methods.* For any unanalyzed library method m, it assumes the typing $\mathsf{poly}, \mathsf{poly} \rightarrow \mathsf{poly}$. This typing conservatively propagates a tainted receiver/argument to the left-hand side of the call assignment. Similarly, it propagates a safe left-hand-side to the receiver/arguments. Consider the following code snippet:

```
public class MyListener implements LocationListener {
  @Override
  public void onLocationChanged(Location loc){//source
    double lat = loc.getLatitude();
    Log.d("History", "Latitude: " + lat); // sink
  }
}
```

LocationListener.onLocationChanged(tainted Location l) is a callback method. Parameter l is a tainted source that must propagate throughout the overriding user-defined method MyListener.onLocationChanged(Location loc). The method

```
1  public LocationLeak2 extends Activity implements LocationListener {
2    private double latitude;
3    protected void onResume() {
4      double d = this.latitude;
5      Log.d("Latitude", "Latitude: " + d); // sink
6    }
7    public void onLocationChanged(Location loc) {
8      double lat = loc.getLatitude(); // loc is a source
9      this.latitude = lat;
10   }
11 }
```

Figure 7: LocationLeak2 refactored from DroidBench, highlights DroidInfer's novel handling of callbacks.

overriding constraints (Sect. 3) lead to:

$$\begin{gathered} \mathit{typeof}(\mathsf{MyListener.onLocationChanged(Location\ loc)}) \\ <: \\ \mathit{typeof}(\mathsf{LocationListener.onLocationChanged(tainted\ Location\ l)}) \end{gathered}$$

This entails l <: loc, forcing loc to be tainted as well.

DroidInfer types library method Location.getLatitude()

poly double getLatitude(**poly** Location this)

and creates the following constraints at Statement 4:

$$\mathsf{loc} <: q^4 \rhd \mathsf{poly} \quad q^4 \rhd \mathsf{poly} <: \mathsf{lat}$$

Because loc is tainted, the callsite context q^4 is inferred as tainted. Consequently, lat is inferred as tainted as well, which leads to a type error because Statement 5 requires a safe argument. (Here the parameter msg of Log.d(String tag, String msg) is a safe sink.)

We apply these conservative defaults to the Java and Android libraries. We can apply these defaults to any third-party library we do not wish to analyze.

5.2 Multiple Entry Points and Callbacks

DroidInfer is type-based and modular. Therefore, it can analyze any given set of classes. However, the analysis of an Android app is different from the analysis of an open library and it requires special consideration.

Roughly speaking, we need to capture the "connections" among callback methods, or DroidInfer might miss privacy leaks through fields. Consider the LocationLeak2 example refactored from DroidBench in Fig. 7. The tainted lat of the current location, obtained in callback method onLocationChanged, flows through field latitude and reaches the safe parameter of Log.d in another callback method, onResume. Local variables lat and d are tainted and safe, respectively. If DroidInfer analyzed the app as a standard open library (e.g., as in [17]), it would infer this of onResume as safe. This is because of (TREAD) constraint $\mathsf{this}_{\mathsf{onResume}} \rhd \mathsf{latitude} <: \mathsf{d}$ where $S(\mathsf{latitude}) = \{\mathsf{tainted}, \mathsf{poly}\}$ and $S(\mathsf{d}) = \{\mathsf{safe}\}$. Due to this constraint, $S(\mathsf{latitude})$ would be updated to $\{\mathsf{poly}\}$. Further, DroidInfer would infer this of onLocationChanged as tainted, because of (TWRITE) constraint $\mathsf{lat} <: \mathsf{this}_{\mathsf{onLocationChanged}} \rhd \mathsf{latitude}$ where $S(\mathsf{lat}) = \{\mathsf{tainted}\}$. The inferred typing would type check and the leak through field latitude would be missed.

In Android, the Activity, as well as other component objects, are instantiated by the Android framework. DroidInfer handles the implicit instantiation by *creating equality constraints for all pairs of* this *parameters of callback methods in the same class.* Intuitively, the constraints "connect" callback methods of implicitly instantiated objects.

```
public class SmsReceiver extends BroadcastReceiver {
  public void onReceive(Context c, Intent i) {
    Bundle bundle = intent.getExtras();
    Object[] pdusObj = (Object[]) bundle.get("pdus");
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < pdusObj.length; i++) {
      SmsMessage msg = SmsMessage.createFromPdu((byte[]) pdusObj[i]); // source
      String body = msg.getDisplayMessageBody();
      sb.append(body);
    }
    Intent it = new Intent(c, TaskService.class);
    it.putExtra("data", sb.toString());
    startService(i);
  }
}
public class TaskService extends Service {
  public void onStart(Intent it, int d) {
    String body = it.getSerializableExtra("data");
    List list = new LinkedList();
    list.add(body);
    HttpClient client = ....getHttpClient();
    HttpPost post = new HttpPost();
    post.setURI(URI.create("http://103.30.7.178/getMotion.htm"));
    Entity e = new UrlEncodedFormEntity(list, "UTF8");
    post.setEntity(e); // sink
    client.execute(post);
  }
}
```

Figure 8: SMS message stealing in Fakedaum. The SMS message is intercepted in SmsReceiver and passed to TaskService via Intent. Finally, the message is sent out to the Internet using HTTP post method, resulting in a message leak.

In the LocationLeak2 example, the inference creates an equality constraint for the this parameters of onResume and onLocationChanged:

$$\mathsf{this}_{\mathsf{onResume}} = \mathsf{this}_{\mathsf{onLocationChanged}}$$

This causes a type error, thus detecting the privacy leak.

5.3 Inter-Component Communication (ICC)

Android components (activity, service, broadcast receiver and content provider) interact through ICC objects — mainly *Intents*. There are two forms of Intent: 1) **Explicit Intents** have an explicit target component — the exact target class of the Intent is specified, and 2) **Implicit Intents** do not have a target component, but they include enough information for the system to implicitly determine the target component.

Consider the example refactored from a real malware app, **Fakedaum**[3] in Fig. 8, where the return value of SmsMessage.createFromPdu is a source and the Http request is a sink. The broadcast receiver SmsReceiver intercepts the SMS messages, then puts the messages into an Intent and starts the background service TaskService with the Intent. Then TaskService sends the messages to the Internet without user consent. We must capture the communication between broadcast receiver SmsReceiver and background service TaskService.

We improve analysis precision in the presence of ICC through Intents. For an explicit Intent whose target class is specified by a final or constant string, DroidInfer connects

---

[3] `http://contagiominidump.blogspot.com/2013/11/fakedaum-vmvol-android-infostealer.html`

Tool Name	**AppScan Source**	**Fortify SCA**	**FlowDroid**	**DroidInfer**
Sum, Precision and Recall–excluding implicit flows				
√	14	17	26	27
×	5	4	4	7
◯	14	11	2	1
Precision	74%	81%	86%	79%
Recall	50%	61%	93%	96%
F-measure	0.60	0.70	0.89	0.87

Figure 9: Comparison on DroidBench 1.0 [7]. √ = correct warning (higher is better), × = false warning (lower is better), ◯ = missed flow (lower is better), Precision $p = \surd/(\surd + \times)$, Recall $r = \surd/(\surd + \bigcirc)$, F-measure $= 2pr/(p+r)$.

the data carried by the Intent using placeholders. DroidInfer replaces the Intent with a "typed" Intent at both the sender and the receiver components. In addition, each putExtra and getExtra are treated as writing and reading a field in the "typed" Intent, respectively. Since the target class of Intent it in Fig. 8 (line 11) is specified by constant TaskService.class, DroidInfer transforms the program into:

```
10  ...
11  TaskService_Intent it = new TaskService_Intent();
12  TaskService_Intent.data = sb.toString();
13  ...
18  String body = TaskService_Intent.data;
```

As a result, the intercepted message is connected to the post data via placeholder data of TaskService_Intent. The leak is captured by DroidInfer.

DroidInfer makes the worst-case assumption for explicit Intents whose target class is not specified by a constant string, as well as for implicit Intents carrying sensitive data, as their content can be intercepted by any, possibly malicious, component. This is achieved by annotating as safe the Intent parameter of library methods that start new components, such as startActivity and startService. For example, if it2 refers to an implicit Intent carrying location information, then there is a type error due to statement startSerivce(it2) because startSerivce requires a safe argument, but it2 is tainted.

6. EMPIRICAL RESULTS

We have built a type inference and checking framework and we have instantiated the framework with several type systems and their corresponding inferences. Initially, the framework had one front-end, built on top of the Checker Framework [31] (CF). CF takes as input the Java source code, which unfortunately is not available for most Android apps, as they are usually delivered as Android Package Files (APKs). Therefore, we extended our type inference framework by building an Android constraint generation front-end, based on Soot [37] and Dexpler [2]. It is worth noting that we came upon DFlow and DroidInfer as instances of our framework and they proved very effective.

The Android front-end takes as input the Jimple files transformed by Soot and Dexpler and outputs the constraints generated according to the typing rules in Sect. 3. Next, the generated constraints along with the annotated libraries where sources and sinks are defined, are supplied to the type inference engine, which computes the set-based solution then either extracts a valid typing or reports type errors for the analyzed program. All sources and sinks are listed in the technical report [16]. They are the union of the sources and sinks of DroidBench 1.0 [7, 1] and the network sinks of Contagio [25]. The sources are various phone state and location. The sinks include the expected log sinks and the following Internet sinks: WebView.loadUrl, URL.openConnection, and Http request. The sinks include Intents, which are necessary for the flows in DroidBench 1.0. The type inference framework, including DFlow and DroidInfer, is publicly available at `http://code.google.com/p/type-inference/`.

We build 0-CFA call graphs using WALA[4]. Recall that we use the set of reachable methods from the call graph to check that the finding of DroidInfer occurs entirely within those methods (Sect. 4). We use support in WALA, contributed by SCanDroid[8], to build call graphs of APKs.

All experiments run on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM. The maximal heap size is set to 2 GB. The software environment consists of Oracle JDK 1.6 and the Soot 2.5.0 nightly build.

6.1 Hypotheses

We evaluate the DroidInfer system along three hypotheses:
(H1) High recall and precision. DroidInfer misses few true flows and reports few false positive flows.
(H2) Network flows. DroidInfer detects leaks of phone or location data to the network.
(H3) Scalability. DroidInfer scales to large apps.

We run DroidInfer on three sets of apps: 1) DroidBench 1.0 [7], 2) 22 apps from the Contagio website [25], known to contain leaks, and 3) 144 popular apps from the Google Play Store, including the top free 30 apps at the time of writing.

6.2 DroidBench

We run DroidInfer on DroidBench 1.0, which is a suit of 39 Android apps designed by Fritz et al. [7, 1]. DroidBench exercises many difficult flows, including flows through fields and method calls, as well as Android-specific flows. DroidBench is the standard evaluating taint analyses for Android. We compare with three other taint analysis tools – AppScan Source [18], Fortify SCA [12], and FlowDroid [7, 1], using the results presented by Fritz et al. [7]. Fig. 9 summarizes the comparison. DroidInfer outperforms AppScan Source and Fortify SCA, which miss substantial amount of flows. The low recall contributes to the slightly higher precision reported by Fortify SCA. FlowDroid is slightly more precise than DroidInfer because it uses a flow-sensitive analysis. DroidBench tests for flow sensitivity and our analysis, which is flow-insensitive, misses those tests. Overall, the F-measures for FlowDroid and DroidInfer are essentially the same. This strongly supports hypothesis **H1**.

6.3 Contagio

We analyzed all 22 apps tagged as "infostealer" on the contagio website [25]. DroidInfer detects that 19 out of the 22 apps send out phone state (e.g. DeviceId, SimSerialNumber, and PhoneNumber), SMS messages, and/or location information through HTTP or text messages, or write into a socket. The list of apps and detailed leaks can be found in the technical report [16]. DroidInfer detects no leaks for the remaining 3 apps. For two of the APK files, **FakePlay** and **Repane**, Soot/Dexpler did not generate Jimple files and DroidInfer in turn did not generate constraints. DroidInfer reports zero type error on **Phospy**. (**Phospy** appears to

---

[4] `http://wala.sourceforge.net`

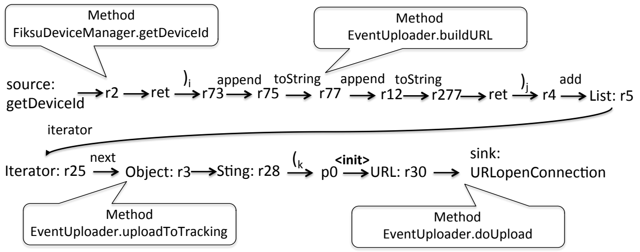


Figure 10: A source-sink path in Fiksu. When a flow is triggered by a library call, CFL-Explain labels the edge with the corresponding library method. When types change, e.g., due to library calls, we show the new type at the target (e.g., List r5). We keep the identifiers exactly as they appear in the Jimple code.

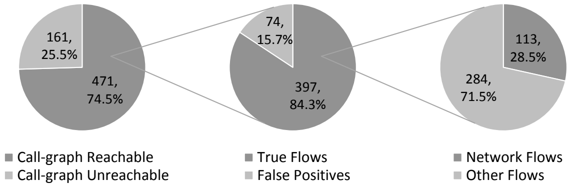


Figure 11: Results.

steal jpg and mp4 files, and such sources are not included in DroidInfer at this point). All type errors on these apps are explained and there are no false positives. These results strongly support hypotheses **H1** and **H2**.

6.4 Google Play Store

We analyze 144 free Android apps from the official Google Play Store. These include the top 30 free apps (as of Jan 5th 2015, the time of writing) as well as other popular apps from the Editor's Choice list, and cover at least 24 categories. DroidInfer throws an Internal error in Dexpler on 1 app and an Out-of-memory error on 5 apps. (Recall that the max heap size is 2 GB.) It analyzes all other 138 apps successfully.

6.4.1 Results

DroidInfer identifies sources and sinks in 111 apps and reports 632 type errors over 88 apps. Two authors of the paper inspected all type errors with CFL-Explain.

Fig. 11 summarizes our results. Of the 632, 161 type errors are refuted by CFL-Explain. Almost all of the refutations are due to the call graph. The false positive rate is 15.7%, which is well within the accepted bounds. (The reason false positives happen will be explained shortly.) 113 true flows (29%), spanning 40 apps, are network flows (i.e., Location or DeviceId flows to the Internet). The remaining flows are flows of Location or DeviceId to Logs and to a lesser extent to Intent. This strongly supports hypothesis **H2**.

DroidInfer takes 139 seconds per app on average. It takes less than 3 minutes on 99 of the 138 apps, between 3 and 5 minutes on 31 apps, and between 5 and 8 minutes on 6 apps. The 2 outliers run in 18 and 19 minutes. The call graphs are built in, on average, 90 seconds per APK, with a range of 6s to 373s. CFL-Explain prints source-sink paths instantly. This timings strongly support hypothesis **H3**.

In contrast to the FlowDroid researchers [1], who report no network flows, we uncover many network flows. Almost *a third* of all apps and almost *a half* of the apps with errors collect sensitive data and send this data over the network. In numerous cases, the DeviceId is sent over the network as part of the URL string. The detailed list of apps and leaks can be found in the technical report [16].

We show one representative network flow. DroidInfer reports the following type error in the Fiksu tracking library (com.fiksu.asotracking.*) included in the **Zillow** app:

$$q^{i} \triangleright \mathsf{ret}_{\mathsf{getDeviceId}}\{\mathsf{tainted}\} <: \mathsf{r2}\{\mathsf{safe}\}$$

The source-sink path reported by CFL-Explain is shown in Fig. 10. Source DeviceId is returned from method getDeviceId into method buildUrl, which forms a URL string "https://...&deviceId=...&uiud=...". buildUrl adds this string to a list of saved URLs; subsequently it iterates over the list, retrieves each URL string and sends the string as an argument to method doUpload. Other examples of complex flows can be found in the technical report [16].

Similarly to the FlowDroid researchers [1], we uncover many flows of DeviceId and Location to logs. In one interesting case, the **Whatsapp** app dumps the SMS message body into the log when a certain IOException occurs. In the majority of cases the logs appear for debugging purposes (to the best of our understanding.) It is unclear why apps log so much sensitive info, usually in clear text, given that malicious apps may read the logs (until Android 4.0, any app that held the READ_LOGS permission could read the logs).

The reader may wonder why false positives occur given that CFL-Explain filters out infeasible paths. Recall that the DroidInfer system does not analyze libraries. Thus, constraints due to library calls result in "local" edges by CFL-Explain, that is, edges connecting two local variables, with no field or call annotations. Edge $\mathsf{r4} \rightarrow \mathsf{r5}$ in Fig. 10, constructed from DroidInfer constraint $\mathsf{r4} <: \mathsf{r5}$ is an example of such local edge. These edges subsume the field accesses and method calls that happen inside the library.

In rare cases, these edges cause infeasible paths. The most common case writes sensitive data (e.g., DeviceId) into a field, then calls a library method on the object: e.g., $\mathit{source} \rightarrow \mathsf{r1} \xrightarrow{[_{\mathsf{f}}} \mathsf{UserActivity}\colon \mathsf{r2} \xrightarrow{\mathsf{getPackageName}} \mathit{sink}$. We assume that the library method does not access the sensitive data stored in fields, and count such cases as false positives.

We conclude this section with a brief discussion of the usability of the system. DroidInfer is completely automatic. CFL-Explain requires users to enter an identifier and examine the paths, because, as discussed above, library calls may give rise to false positive paths. In our experience, it takes less than 1 minute to vet the flow paths for a given type error, 2 minutes in rare cases. The tool was used successfully by two of the authors of this paper, as well as an undergraduate research assistant with no knowledge of program analysis.

6.4.2 Runtime Results

To gauge the usefulness of the static results, we run 10 random apps and collect and analyze their logs using Android Device Monitor. There are 76 type errors reported as true flows across the 10 apps. Despite short runs, we covered 14 type errors, or almost 20%. These errors span 8 apps and expose flows of DeviceId to both logs and the network. The flows are obvious tracking, as in Fig. 10, which is covered.

Fig. 12 summarizes the results. Of the 62 type errors we did not cover, 13 are impossible to cover with our technique.

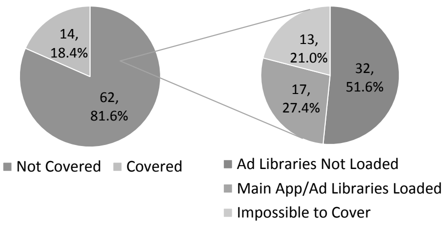


Figure 12: Runtime results.

For example, several type errors are flows to the network. However, there is no log around the network call and we cannot confirm the call.

The analysis reports a substantial number of type errors that reveal true, dangerous flows. In the same time, it reports many "difficult" errors, i.e., type errors that are likely true flows, but are difficult to trigger with runtime analysis. A lot of the uncovered type errors are in ad libraries that do not load during our runs. Yet we found it impossible to trigger a specific ad library. For example, in **Cut the Rope 2** we observed ads from AdMarvel and other libraries in unrecorded runs. (Our tool reported several type errors in AdMarvel.) Unfortunately, when recording the logs, we observed ads only from Unity3D until the app stopped serving ads altogether.

6.4.3 Comparison with FlowDroid

We ran FlowDroid [1] on the top 30 free apps from the Google Play Store with max heap size set to 6 GB. FlowDroid threw an Out-of-memory error on 28 of the apps (we confirmed with the developers that FlowDroid indeed requires more than 6 GB of memory[5]). In contrast, DroidInfer runs with a max heap size of 2 GB and succeeds on 28 of the 30 apps. This result strongly supports hypothesis **H3**.

FlowDroid succeeds on 50 of the remaining 114 apps. It reports more than 4000 flows over the 50 apps. We examined a random 21 apps and compared the results with DroidInfer. FlowDroid reports thousands of flows from Bundle, Intent and Context, as it is overly-conservative in its handling of inter-process communication. In only 6 apps does FlowDroid report "classical" flows: there are 4 log flows (DeviceId or Location to log) and 2 network flows (DeviceId or Location to Internet). In contrast, DroidInfer reports only "classical" flows, 14 network, in all 21 apps. These results are consistent with Artz et al. [1]. It is unclear why FlowDroid reports so few log flows and virtually no network flows — like DroidInfer, it does specify DeviceId and Location as sources, and logs, URL.openConnection and Http request as sinks.

7. RELATED WORK

There is a large body of work on Android malware analysis, both dynamic and static. We focus the discussion on static analyses, excluding FlowDroid. LeakMiner [42] is a points-to based static analysis for Android. It models the Android lifecycle to handle callback methods. However, LeakMiner is context-insensitive which may lead to false positives. It is unclear whether LeakMiner supports ICC. Cassandra [23] is a type-based information- flow analysis for Android apps. It is not evaluated on real-world apps. SCANDAL [20] is a static analyzer that detects privacy leaks in Android apps. It directly processes Dalvik bytecode. SCANDAL is limited by high false positive rate — the average false positive rate is about 55%, primarily due to the *unknown paths*, which make up more than half of the total paths [20]. AndroidLeaks [9] finds potential leaks of private information in Android apps. It uses WALA to construct a context-sensitive System Dependence Graph (SDG) and a context-insensitive overlay for tracking heap dependencies in the SDG. CHEX [24] can automatically vet Android apps for component hijacking vulnerabilities. It models the vulnerabilities from a data-flow analysis perspective and detects possible hijack-enabling flows and data leakage. Unfortunately, these tools are not publicly available and we cannot compare with DroidInfer. Fritz et al. have contacted the authors of these tools, but still, they were unable to compare due to various reasons [7].

SCanDroid [8] focuses on ICC. It formalizes the data flows through and across components in a core calculus. Epicc [30] discovers ICC for Android apps by identifying a *specification* for every ICC source and sink, including the ICC Intent action, data type, category, etc. We plan to integrate Epicc in DroidInfer to provide more channels for privacy leaks.

In previous work we built SFlow [15], a type-based taint analysis for Java web applications, which can also analyze Java source of Android apps. Although we build upon this work, this paper has several substantial contributions. First, we interpret type errors in terms of CFL-reachability, which is a major step towards usability of type-based tools. Second, we incorporate control-flow guarantees via call graphs; SFlow provides no such guarantees, which means that many type errors may be unreachable. Another key difference is that SFlow uses the receiver, while DFlow uses the callsite context at method calls. Thus, DFlow is more precise than SFlow and accepts more programs (see [16] for details). In addition, SFlowInfer, the inference tool of SFlow, only works on Java source, which is not available for most Android apps. DroidInfer works on both Java source and Android APKs and it can analyze any real-world Android app. The extensive evaluation on Google Play Store apps is a major contribution over our previous work. Ernst et al. [**?**] present a type-based taint analysis system similar to ours. However, they target source code and therefore do not analyze Google Play Store apps. Furthermore, they require user annotations, while DroidInfer requires *no user annotations*. Earlier work on type-based taint analysis comes from Shankar et al. [36] who present a type system for detecting string format vulnerabilities in C. Classical work on type-based information flow control includes the type systems by Volpano et al. [38] and Myers [28]. DFlow and DroidInfer are substantially simpler and thus more practical.

8. CONCLUSION

We have presented DroidInfer a system for detecting privacy leaks in Android apps. Empirical evaluation has shown that our approach is effective.

One limitation of our approach is that CFL-Explain may be difficult to use by non-experts on static analysis. Furthermore, while CFL-Explain reduces imprecision (by reversing field insensitivity), it may introduce unsoundness. We will investigate approaches that mitigate these limitations.

9. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel.

---

[5] Eric Bodden: personal communication.

FlowDroid: Precise context-, flow-, field-, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI, to appear*, 2014.

[2] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *SOAP*, pages 27–38, 2012.

[3] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[4] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. Mcdaniel, and A. N. Sheth. TaintDroid : An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 1–6, 2010.

[5] M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. Technical Report UW-CSE-14-04-02, University of Washington Department of Computer Science and Engineering, Apr. 2014.

[6] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *ACM Conference on Programming Languages Design and Implementation*, pages 253–263, 2000.

[7] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android application. Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013.

[8] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid : Automated security certification of Android applications. unpublished.

[9] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST*, pages 273–290, 2012.

[10] Google. Android and Security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, 2012.

[11] Google. Google Play Store. `https://play.google.com`, 2014.

[12] HP. HP Fortify Static Code Analyzer. `http://www8.hp.com/us/en/software-solutions/application-security/`, 2013.

[13] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.

[14] W. Huang, Y. Dong, and A. Milanova. Type-based Taint Analysis for Java Web Applications. Technical report, Rensselaer Polytechnic Institute, 2013.

[15] W. Huang, Y. Dong, and A. Milanova. Type-based taint analysis for Java web applications. In *FASE*, pages 140–154, 2014.

[16] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. Technical report, Department of Computer Science, Rensselaer Polytechnic Institute.

[17] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, pages 879–896, 2012.

[18] IBM. IBM Security AppScan. `http://www-03.ibm.com/software/products/en/appscan`, 2013.

[19] IDC. Smartphone OS Market Share, Q3 2014. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, 2014.

[20] J. Kim, Y. Yoon, and K. Yi. SCANDAL: Static analyzer for detecting privacy leaks in Android applications. In *MoST*, 2012.

[21] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 425–434, New York, NY, USA, 2007. ACM.

[22] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, and P. Aldous. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *SPSM*, pages 21–32, 2013.

[23] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104. ACM, 2014.

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

[25] Mila. contagio mobile. `http://contagiominidump.blogspot.com`, 2014.

[26] A. Milanova and W. Huang. Composing polymorphic information flow systems with reference immutability. In *FTfJP*, pages 5:1–5:7, 2013.

[27] A. Milanova, W. Huang, and Y. Dong. In *ACM Conference on Programming and Practice of Programming in Java*, pages 99–109, 2014.

[28] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[29] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, pages 132–145, 1997.

[30] D. Octeau, P. Mcdaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security*, pages 543–558, 2013.

[31] M. M. Papi, M. Ali, T. L. Correa Jr, J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.

[32] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In *EuroSec*, 2013.

[33] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:5–19, 1998.

[34] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162—-186, 2000.

[35] A. Sampson, W. Dietl, and E. Fortuna. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[36] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, pages 201–220, 2001.

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON*, pages 13–23, 1999.

[38] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, pages 167–187, 1996.

[39] M. Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 38–43, New York, NY, USA, 1986. ACM.

[40] R. Xu, H. Saïdi, R. Anderson, and H. SaÄśdi. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security*, pages 539–552, 2012.

[41] L. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, pages 569–584, 2012.

[42] Z. Yang and M. Yang. LeakMiner: Detect information leakage on Android with static taint analysis. *2012 Third World Congress on Software Engineering*, pages 101–104, Nov. 2012.