Scalable and Precise Taint Analysis for Android

Wei Huang
Google
huangwe@google.com

Yao Dong Ana Milanova
RPI
{dongy6,milana2}@rpi.edu

Julian Dolby
IBM Research
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.

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_LOG 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 [11], 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 [23, 42, 20, 10, 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. This is surprising, given the common 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 [16]. 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. In contrast to 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, 1], the standard for evaluating static taint analysis for Android. FlowDroid's F-measure is 0.89.

DroidInfer scales because it completely avoids points-to analysis. It is precise because in essence it is 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.

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.

```
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.

- An extensive empirical evaluation on three sets of Android apps: 1) DroidBench [7, 1], 2) 22 malware apps from the Contagio website [24] and 3) 144 apps from the Google Play Store [12]. 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 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. First, 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. An Android app consists of a number of *components*, each of which can be instantiated and run within the Android framework. The Android app defines its own behaviors at different states of the component lifecyle by overriding pre-defined callback methods. Multiple entry points present a challenge to traditional points-to-based static analyses, which usually require whole-program analysis. Furthermore, control flow is interrupted by callbacks from Android. In the WallpapersMain example, control from onCreate to onActivityResult must be captured in order to detect the leak.

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:

$$\mathsf{safe} <: \mathsf{tainted}^1$$

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 (usually long) flow path from source to sink! 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

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

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

```
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.

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 and str 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. Our approach presents a 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:

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

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.

$$\begin{array}{lll} cd & ::= \mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\} & \textit{class} \\ fd & ::= t\ \mathsf{f} & \textit{field} \\ md & ::= t\ \mathsf{m}(t\ \mathsf{this},\ t\ \mathsf{x})\ \{\ \overline{t\ \mathsf{y}}\ \mathsf{s};\ \mathsf{return\ y}\ \} & \textit{method} \\ \mathsf{s} & ::= \mathsf{s};\mathsf{s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.f} & \\ & \quad \mid \mathsf{y.f} = \mathsf{x} \mid \mathsf{x} = \mathsf{y.m(z)} & \textit{statement} \\ t & ::= q\ \mathsf{C} & \textit{qualified type} \\ q & ::= \mathsf{tainted} \mid \mathsf{poly} \mid \mathsf{safe} & \textit{qualifier} \end{array}$$

Figure 3: Syntax. C and D are class names, f is a field name, m is a method name, and x, y, and z are names of local variables, formal parameters, or parameter this. As in the code examples, this is explicit. For simplicity, we assume all names are unique.

$$\text{(TNEW)}\ \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}} \qquad \text{(TASSIGN)}\ \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

$$\text{(TWRITE)}\ \frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \textit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

$$\text{(TREAD)}\ \frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \textit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}$$

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

Figure 4: 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.

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. The syntax is shown in Fig. 3. 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.

The typing rules are defined in Fig. 4. 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 $\{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$.

Consider the example in Fig. 2. At callsite 10, q^{10} is $\mathsf{tainted}$ and $q^{10} \rhd \mathsf{poly}$ is interpreted as $\mathsf{tainted}$. 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 $q^{11} \rhd \mathsf{poly}$ is interpreted as safe. 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 [25].

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

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

This entails $q_{\mathsf{this_m}} <: q_{\mathsf{this_n}}$, $q_{\mathsf{p_m}} <: q_{\mathsf{p_n}}$, and $q_{\mathsf{ret_n}} <: q_{\mathsf{ret_m}}$.

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. 4. 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. 4. 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 [14]. 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.

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

1: **procedure** SUMMARYSOLVER
2:   **repeat**
3:     **for each** c in C **do**
4:       SOLVECONSTRAINT(c)
5:       **if** c is $q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}$ **and** $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ **then** ▷ Case 1
6:         Add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ into C
7:       **else if** c is $q_{\mathsf{x}} \rhd q_{\mathsf{f}} <: q_{\mathsf{y}}$ **and** $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ **then** ▷ Case 2
8:         Add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ into C
9:       **else if** c is $q_{\mathsf{x}} <: q_{\mathsf{y}}$ **then** ▷ Case 3
10:         **for each** $q_{\mathsf{y}} <: q_{\mathsf{z}}$ in C **do** Add $q_{\mathsf{x}} <: q_{\mathsf{z}}$ to C **end for**
11:         **for each** $q_{\mathsf{w}} <: q_{\mathsf{x}}$ in C **do** Add $q_{\mathsf{w}} <: q_{\mathsf{y}}$ to C **end for**
12:         **for each** $q_{\mathsf{w}} <: q^i \rhd q_{\mathsf{x}}$ and $q^i \rhd q_{\mathsf{y}} <: q_{\mathsf{z}}$ in C **do** ▷ Case 4
13:           Add $q_{\mathsf{w}} <: q_{\mathsf{z}}$ to C
14:         **end for**
15:       **end if**
16:     **end for**
17:   **until** S remains unchanged
18: **end procedure**

Figure 5: Initially, S is the result of the set-based solution and C is the set of constraints for program statements. Cases 1 and 2 add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ to C because $q_{\mathsf{y}} \rhd \mathsf{poly}$ always yields q_{y}. Case 3 adds constraints due to transitivity. Case 4 adds constraints between actual(s) and left-hand-side(s) at calls: if there are constraints $q_{\mathsf{w}} <: q^i \rhd q_{\mathsf{x}}$ (flow from actual w to formal x) and $q^i \rhd q_{\mathsf{y}} <: q_{\mathsf{z}}$ (flow from return value y to left-hand-side z), and also $q_{\mathsf{x}} <: q_{\mathsf{y}}$ (flow from formal to return value), Case 4 adds $q_{\mathsf{w}} <: q_{\mathsf{z}}$. Line 4 calls SolveConstraint(c): the solver infers new constraints, which remove additional infeasible qualifiers from S. This process repeats until S stays unchanged. We represent equality constraints as two subtyping constraints (e.g., $q_{\mathsf{x}} <: q_{\mathsf{y}}$ and $q_{\mathsf{y}} <: q_{\mathsf{x}}$), hence no explicit mention of equality constraints.

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 [16] to DFlow, is shown in Fig. 5.

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. 5). 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).

Line	Code	Constraints	Set-based solution	New constraints
1	`public class Data {`			
2	`String secret;`		$\boxed{S(\mathsf{secret}) = \{\mathsf{poly}\}}$	
3	`String get(Data this) {return this.secret;}`	$\boxed{\mathsf{this}_{\mathsf{get}} \triangleright \mathsf{secret} <: \mathsf{ret}_{\mathsf{get}}}$	$S(\mathsf{ret}_{\mathsf{get}}) = \{\mathsf{poly}, \mathsf{safe}\}$	$\boxed{\mathsf{this}_{\mathsf{get}} <: \mathsf{ret}_{\mathsf{get}}}$
4	`void set(Data this, String p){this.secret = p;}`	$\boxed{\mathsf{p} <: \mathsf{this}_{\mathsf{set}} \triangleright \mathsf{secret}}$	$S(\mathsf{p}) = \{\mathsf{tainted}, \mathsf{poly}\}$	$\boxed{\mathsf{p} <: \mathsf{this}_{\mathsf{secret}}}$
5	`}`			
6	`public class FieldSensitivity2 extends Activity {`			
7	`protected void onCreate(Bundle b) {`			
8	`Data dt = new Data();`		$S(\mathsf{dt}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$	
9	`TelephonyManager tm = (TelephonyManager) getSystemService("phone");`			
10	`String sim = tm.getSimSerialNumber();`	$\boxed{q^{10} \triangleright \mathbf{tainted} <: \mathsf{sim}}$	$S(\mathsf{sim}) = \{\mathsf{tainted}\}$	
11	`dt.set(sim);`	$\boxed{\mathsf{sim} <: q^{11} \triangleright \mathsf{p}}$		$\boxed{\mathsf{sim} <: \mathsf{dt}}$
12		$\boxed{\mathsf{dt} = q^{11} \triangleright \mathsf{this}_{\mathsf{set}}}$	$S(\mathsf{this}_{\mathsf{set}}) = \{\mathsf{tainted}, \mathsf{poly}\}$	
13	`SmsManager sms = SmsManager.getDefault();`			
14	`String sg = dt.get();`	$\boxed{\mathsf{dt} <: q^{14} \triangleright \mathsf{this}_{\mathsf{get}}}$	$S(\mathsf{this}_{\mathsf{get}}) = \{\mathsf{poly}, \mathsf{safe}\}$	
15		$\boxed{q^{14} \triangleright \mathsf{ret}_{\mathsf{get}} <: \mathsf{sg}}$		$\boxed{\mathsf{dt} <: \mathsf{sg}}$ TYPE ERROR!
16	`sms.sendTextMessage("+123",null,sg,null,null);`	$\boxed{\mathsf{sg} <: q^{16} \triangleright \mathbf{safe}}$	$S(\mathsf{sg}) = \{\mathsf{safe}\}$	
17	`}`			
18	`}`			

Figure 6: FieldSensitivity2 example refactored from DroidBench. The frame box beside each statement shows 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 {safe}, then 14 forces $\mathsf{ret}_{\mathsf{get}}$ to be {poly, safe} and then 3 forces $\mathsf{this}_{\mathsf{get}}$ to be {poly, safe} and secret to be {poly} (recall that fields are initialized to {tainted, poly}, Sect. 3.2.1). 10 forces sim to be {tainted}, which in turn forces the parameters p and $\mathsf{this}_{\mathsf{set}}$ to be {tainted, 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 $\mathsf{this}_{\mathsf{get}} \triangleright \mathsf{secret} <: \mathsf{ret}_{\mathsf{get}}$ 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, $\mathsf{p} <: \mathsf{this}_{\mathsf{set}} \triangleright \mathsf{secret}$ 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 {tainted} and sg is {safe}, these constraints cause a TYPE ERROR, detecting the leak.

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

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

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. 5) 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, which dominates the inference, has worst-case complexity of $O(n^3)$, where n is the size of the program. There are at most $O(n)$ iterations of the outer loop (line 2) because at least one of $O(n)$ variables is updated in each iteration. There are at most $O(n^2)$ iterations of the inner loop (line 3) because in the worst case every two variables can form a constraint, resulting in $O(n^2)$ constraints. Note that lines 10-12 need to run only when a new constraint is first discovered, so they contribute $O(n^3)$ over all constraints. Altogether, we have worst-case complexity of $O(n^3)$. Soundness of DFlow is argued as in [15].

3.2.3 *Example*

Let us consider the FieldSensitivity2 example refactored from DroidBench [7] in Fig. 6. 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. 6 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 useleful. For example, DroidInfer produces the following type error at statement 10 in Fig. 6:

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

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 [26], we studied the connection between DFlow/DroidInfer and CFL-reachability [33, 6]. The key idea is that the type constraints in Fig. 4 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. 6, the field read $\mathsf{return\ this.secret}$ and its DFlow constraint $\mathsf{this_{get}} \rhd \mathsf{secret} <: \mathsf{ret_{get}}$ correspond to edge

$$\mathsf{this_{get}} \xrightarrow{]_{\mathsf{secret}}} \mathsf{ret_{get}}$$

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. 6, callsite 14 gives rise to the following constraints:

$$\mathsf{dt} <: q_{14} \rhd \mathsf{this_{get}} \quad q_{14} \rhd \mathsf{ret_{get}} <: \mathsf{sg}$$

which correspond to the following edges:

$$\mathsf{dt} \xrightarrow{(_{14}} \mathsf{this_{get}} \quad \mathsf{ret_{get}} \xrightarrow{)_{14}} \mathsf{sg}$$

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. 6 give rise to source-sink path

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

which presents an intuitive explanation of the type error we showed 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 edge $\mathsf{this} \xrightarrow{)_{11}} \mathsf{dt}$ (naturally, the flow at callsite 11 is from dt to this). This *inverse* edge is due to the mutation of $\mathsf{this_{set}}$, 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 [26]. 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. 5.) 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 $\mathsf{tainted}$. 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. 7. 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

1: **procedure** CFL-Explain
2: Add $\langle start, \epsilon, \epsilon\rangle$ to Q
3: Add $\langle start, \epsilon, \epsilon\rangle \to []$ 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 $\langle m, f{+}f', c{+}c'\rangle \to p'$ 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 7: 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., $e = (\mathsf{this_{get}}, \mathsf{ret},]_{\mathsf{secret}}, \epsilon)$) CG is a precomputed call graph. $Method(m)$ gives the enclosing method of m.

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. 6, CFL-explain takes as input the variable on the right-hand-side of the type error, sim, and produces the source-sink path we showed earlier:

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

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. 7 verifies 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 the approximation in structure-transmitted dependences employed by DroidInfer (see [26]), and CFL-Explain cannot confirm a feasible path.

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 poly, poly $\rightarrow$ 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:

```
1 public class MyListener implements LocationListener {
2   @Override
3   public void onLocationChanged(Location loc){//source
4     double lat = loc.getLatitude();
5     Log.d("History", "Latitude: " + lat); // sink
6   }
7 }
```

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 overriding constraints (Sect. 3) lead to:

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

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

DroidInfer assumes that library method Location.getLatitude() is typed as follows:

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

Multiple entry points and the ubiquitous use of callbacks in Android apps cause difficulty for traditional dataflow and points-to-based analysis. The Android app is not a closed program. Instead, it runs within the Android framework, which implicitly creates objects of the user-defined classes and calls user-defined methods in the app through 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. 8. 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.

```
1 public LocationLeak2 extends Activity implements
      LocationListener {
2   private double latitude;
3   protected void onResume() {
4     double d = this.latitude; // TYPE ERROR!
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 8: LocationLeak2 refactored from DroidBench, highlights DroidInfer's novel handling of callbacks.

If the app were a standard open library, it would be composed with user code, which would instantiate the Activity and reveal the leak. Consider this hypothetical user code:

```
1 Activity a = new LocationLeak2();
2 a.onLocationChanged(loc);
3 a.onResume();
```

When composing this code with the inferred typing for LocationLeak2, there would be a type error because Statement 2 requires a to be tainted ($\mathsf{this}_{\mathsf{onLocationChanged}}$ is mutable; thus, there is equality constraint at 2: $\mathsf{a} = q^2 \rhd \mathsf{this}_{\mathsf{onLocationChanged}}$), while Statement 3 requires a to be safe.

The Android app is not composed with user code. Instead, the Activity, as well as other component objects, are instantiated by the Android framework. Therefore, DroidInfer needs to handle the implicit instantiation of app objects. DroidInfer *creates equality constraints for all pairs of* this *parameters of callback methods in the same class*. This "connects" this of callback methods. If the app type checks, this means there is a solution for constraints

$$\begin{array}{rcl} q_{\mathsf{a}} & <: & q^{i1} \rhd q_{\mathsf{this}_{\mathsf{callback1}}} \\ q_{\mathsf{a}} & = & q^{i2} \rhd q_{\mathsf{this}_{\mathsf{callback2}}} \\ & \ldots & \end{array}$$

which correspond to the calls to callback methods in the Android framework.

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

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

$\mathsf{this}_{\mathsf{onResume}}$ becomes tainted. There is a type error at Statement 4, thus detecting the privacy leak.

```
1   public class SmsReceiver extends BroadcastReceiver {
2     public void onReceive(Context c, Intent i) {
3       Bundle bundle = intent.getExtras();
4       Object[] pdusObj = (Object[]) bundle.get("pdus");
5       StringBuilder sb = new StringBuilder();
6       for (int i = 0; i < pdusObj.length; i++) {
7         SmsMessage msg = SmsMessage.createFromPdu((byte
                []) pdusObj[i]); // source
8         String body = msg.getDisplayMessageBody();
9         sb.append(body);
10      }
11      Intent it = new Intent(c, TaskService.class);
12      it.putExtra("data", sb.toString());
13      startService(i);
14    }
15  }
16  public class TaskService extends Service {
17    public void onStart(Intent it, int d) {
18      String body = it.getSerializableExtra("data");
19      List list = new LinkedList();
20      list.add(body);
21      HttpClient client = ....getHttpClient();
22      HttpPost post = new HttpPost();
23      post.setURI(URI.create("http://103.30.7.178/getMotion.
            htm"));
24      Entity e = new UrlEncodedFormEntity(list, "UTF8");
25      post.setEntity(e); // sink
26      client.execute(post);
27    }
28  }
```

Figure 9: 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.

5.3 Inter-Component Communication (ICC)

Android components (activity, service, broadcast receiver and content provider) interact through ICC objects — mainly *Intents*. Communication can happen across applications as well, to allow functionality reuse. There are two forms of Intent in Android: 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.

Capturing data flow through Intents is important for detecting privacy leaks in Android. Consider the example refactored from a real malware app, **Fakedaum**[3] in Fig. 9, where the return value of SmsMessage.createFromPdu is a source and the parameter of HttpPost.setEntity 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. If the communication between the broadcast receiver SmsReceiver and the background service TaskService is not captured, there is no way to detect the privacy leak.

We propose a technique to 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 the data carried by the Intent using placeholders. DroidInfer replaces the Intent with a "typed"

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

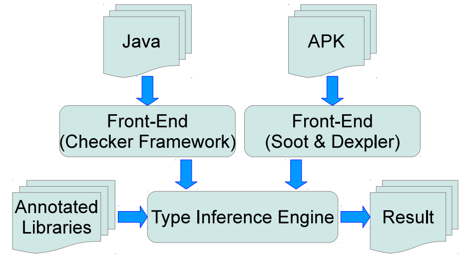


Figure 10: Architecture of the inference framework.

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. 9 (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.

For explicit Intents whose target class is not specified by a constant string, a string analysis, which we leave for future work, is required to determine the target. DroidInfer makes the worst-case assumption for such explicit Intents, 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. Suppose it2 refers to an implicit Intent carrying current location information, then there is a type error at statement startSerivce(it2) because startSerivce requires a safe argument, but it2 is tainted as it contains tainted data.

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]. The architecture of our inference framework is shown in Fig. 10. 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

Tool Name	**AppScan Source**	**Fortify SCA**	**FlowDroid**	**DroidInfer**
Sum, Precision and Recall–excluding implicit flows				
$\surd$, higher is better	14	17	26	28
$\times$, lower is better	5	4	4	8
$\bigcirc$, lower is better	14	11	2	0
Precision $p = \surd/(\surd + \times)$	74%	81%	86%	78%
Recall $r = \surd/(\surd + \bigcirc)$	50%	61%	93%	100%
F-measure $2pr/(p+r)$	0.60	0.70	0.89	0.88

Figure 11: Summary of comparison on DroidBench [7] with other taint analysis tools ($\surd$ = correct warning, $\times$ = false warning, $\bigcirc$ = missed flow).

Appendix. They are the union of the sources and sinks of DroidBench [7, 1] and the network sinks of Contagio [24]. The sources are various phone sata 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. The type inference framework, including DFlow and DroidInfer, is publicly available at `http://code.google.com/p/type-inference/`.

We build 0-CFA callgraphs 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[9], 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 info to the network, in known malicious apps and in Google Play Store apps.

(H3) Scalability. DroidInfer scales to large apps.

We run DroidInfer on three sets of apps: 1) DroidBench [7], 2) 22 apps from the Contagio website [24], known to contain leaks, and 3) 144 popular apps from the Google Play store, including the top 30 apps at the time of writing.

6.2 DroidBench

We run DroidInfer on DroidBench, 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 [13], and FlowDroid [7, 1], using the results presented by Fritz et al. [7]. Fig. 11 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. In our experience with real-world apps however, flow sensitivity will not help. Overall, the F-measures for FlowDroid and DroidInfer are essentially the same. This strongly supports hypothesis **H1**.

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

AwesomeJokes (4)
DeviceId→URLConnection
Backflash (3)
SmsMessage→URLConnection
BatteryDoctor (5)
Location→HttpEntity
DeviceId→HttpEntity
DeviceId→WebView
BatteryImprove (1)
DeviceId→URLConnection
Beita (4)
DeviceId→HttpEntity
Location→HttpEntity
DroidKungFu (1)
DeviceId→HttpEntity
FakeBanker (7)
PhoneNumber→HttpEntity
SimSerialNumber→HttpEntity
SmsMessage→HttpEntity
PhoneNumber →HttpEntity
Fakedaum (3)
SimSerialNumber→HttpEntity
PhoneNumber→HttpEntity
SmsMessage→HttpEntity
FakeTaobao (2)
PhoneNumber→HttpEntity
DeviceId→HttpEntity
Godwon (5)
DeviceId→HttpEntity
PhoneNumber→HttpEntity
SmsMessage→HttpEntity
Jollyserv (2)
DeviceId→sendTextMessage
PhoneNumber→HttpEntity
Kmin (2)
SubscriberId→URLConnection
DeviceId→URLConnection
Loozfon (4)
PhoneNumber→HttpEntity
DeviceId→HttpEntity
Contact→HttpEntity
Roidsec (4)
PhoneNumber
→Socket OutputStream
Locaion→Socket OutputSteam
Contact→Socket OutputStream
DeviceId→Socket OutputStream
Scipiex (1)
Contact→Socket OutputStream
Simhosy (3)
DeviceId→URLConnection
SubscriberId→URLConnection
Skullkey (1)
DeviceId→HttpEntity
Uranai (5)
DeviceId→HttpEntity
PhoneNumber→HttpEntity
SimSerialNumber→HttpEntity
Contact→HttpEntity
Zertsecurity (3)
DeviceId→HttpEntity
SmsMessage→HttpEntity

Figure 12: Leaks detected in Malware.

6.3 Contagio

We analyzed all 22 apps tagged as "infostealer" on the contagio website [24]. Fig. 12 summarizes the analysis result. 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. 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 errors on **Phospy**. (**Phospy** appears to 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

2048 Number puzzle game_6.06 (8)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

AccuWeather_3.2.14.1 (8)
PhoneData→Log Location→Log
PhoneData→Network Location→Intent
Location→Network

Amazon_2.9.7 (2)
Location→Network

Backgrounds HD Wallpapers_2.0.1 (3)
PhoneData→Log Contact→Log
PhoneData→Network

Chase Mobile_3.16 (2)
Contact→Intent

Clash of Mafias_1.0.45 (9)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Clean Master_5.4.0 (6)
PhoneData→Network Location→Intent
Location→Log Contact→Intent
Location→Network

CSI: Hidden Crimes_1.6.0 (3)
PhoneData→Log Location→Log

Cut the Rope 2_1.3.0 (13)
PhoneData→Log Location→Network
Location→Log

DealMoon_3.4.0 (4)
PhoneData→Log PhoneData→Network

Dice With Buddies_3.3.5 (14)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Dictionary.com_4.4 (11)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Don't Tap The White Tile_2.2.3 (22)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Dumb Ways to Die 2_1.1.1 (1)
PhoneData→Intent

ebay_2.6.0.98 (3)
Location→Log

ES File Explorer_3.1.3 (5)
PhoneData→Log Location→Log
PhoneData→Network

ES Task Manager_1.4.2 (1)
PhoneData→Intent

ESPN Fantasy Football_1.1.6 (1)
PhoneData→Log PhoneData→Network

Expedia_3.6 (2)
PhoneData→Network Location→Log

Facebook_4.0.0.26.3 (2)
Location→Log

Facebook Messenger_5.0.0.16.1 (2)
Location→Log

Flow Free: Bridges_1.7 (3)
PhoneData→Log

GO SMS Pro_5.44 (14)
PhoneData→Log PhoneData→Intent
PhoneData→Network Contact→Intent

Groupon_3.1.3273 (6)
PhoneData→Log Location→Log
PhoneData→Network

GUNSHIP BATTLE_1.3.0 (3)
Location→Log

HotPads_3.1 (2)
Location→Log

iHeartRadio_4.11.0 (5)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Iron Force_1.5.2 (6)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Job Search_2.3 (1)
Location→Log

Kakao Talk_4.6.8 (5)
PhoneData→Log Contact→Log
PhoneData→Intent Contact→Intent
Location→Log

LINE_4.3.0 (8)
PhoneData→Intent Contact→Intent

LinkedIn_3.4.3 (2)
PhoneData→Network Contact→Log

Looney Tunes Dash_1.45.11 (11)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Make It Rain: Love of Money_1.2 (5)
Location→Log

Mint_3.1.1 (1)
Location→Log

MITBBS reader_8.5 (7)
Location→Log

Multi Touch Painting Demo_2.6.0 (3)
PhoneData→Log Location→Log

My Talking Angela_1.1 (4)
PhoneData→Log Location→Log
PhoneData→Intent

My Talking Tom_2.1.1 (5)
PhoneData→Log PhoneData→Intent
PhoneData→Network Location→Log

Netflix_3.9.1 (2)
PhoneData→Log PhoneData→Network

Next Radio_2.0.757 (6)
PhoneData→Log Location→Network
Location→Log

Noom Weight Loss Coach_4.0.5 (4)
PhoneData→Log Location→Network
PhoneData→Network

NYTimes_3.6.3 (3)
Location→Log

ooVoo_2.1.3 (7)
PhoneData→Log PhoneData→Network

Pandora_5.4 (2)
Location→Log Location→Network

Paperama_1.1.0 (3)
PhoneData→Log Location→Log

Pool Billiards Pro_2.49 (1)
PhoneData→Log PhoneData→Network

Powerboat Racing_1.1 (1)
PhoneData→Log

Priceline_2.8.21 (5)
PhoneData→Log Location→Log
PhoneData→Network

Real Fingerprint Scanner_3.4 (3)
PhoneData→Network

Skype_5.1.0.57240 (6)
PhoneData→Log Location→Log

Smash Hit_1.3.3 (3)
PhoneData→Log Location→Log

Snapchat_5.0.34.10 (3)
PhoneData→Log Location→Log

Solitaire_3.0.3 (11)
PhoneData→Log Location→Intent
PhoneData→Network Location→Network
Location→Log

Sound Cloud_14.10.27 (2)
Location→Intent

Spotify Music_1.0.0.82 (1)
PhoneData→Log

StudyBlue_5.4.2 (3)
PhoneData→Log Location→Log
PhoneData→Network

Subway Surfers_1.33.0 (8)
PhoneData→Log Location→Log

Super-Bright LED FlashLight_1.0.3 (21)
PhoneData→Log Location→Log
PhoneData→Network Location→Network

Swype_1.7.3.28966 (4)
Location→Log

Tango Messenger_3.6.84179 (10)
PhoneData→Log Location→Log
PhoneData→Intent Contact→Intent

Temple Run 2_1.9 (1)
PhoneData→Log

textPlus_5.9.1.4671 (12)
PhoneData→Log Location→Log
PhoneData→Network Location→Network
PhoneData→Intent

The Weather Channel_4.2.5 (6)
Location→Log

Tinder_3.3.2 (6)
PhoneData→Log Location→Log
PhoneData→Intent

Trigger_8.9.590 (2)
Location→Log Location→Intent

Trivia Crack_1.9.3 (10)
PhoneData→Log Location→Network
Location→Log

TuneIn Radio_12.0 (3)
Location→Log

Twitter_5.32.0 (4)
PhoneData→Log Location→Intent

Uber_2.7.15 (5)
Location→Log

Venmo_6.4.2 (8)
PhoneData→Log Contact→Intent
Location→Log

Viber_4.3.1.21 (18)
PhoneData→Intent Location→Network

Virtual Pet Care_1.52 (2)
Location→Log Location→Network

Walmart_1.7.2 (1)
Location→Log

WeatherBug_3.4.33 (10)
Location→Log Location→Network

WhatsApp Messenger_2.11.238 (16)
PhoneData→Log Contact→Intent

Words_7.1.4 (17)
PhoneData→Log Location→Log
PhoneData→Network Location→Network
PhoneData→Intent

World of Battleships_1.0.05 (3)
PhoneData→Log Location→Log
PhoneData→Network

Yahoo Mail_3.1.3 (8)
PhoneData→Network

Yik Yak_2.0.002 (2)
PhoneData→Log

Yo_1.110640.50 (4)
PhoneData→Log Location→Log

ZEDGE Ringtones Wallpapers_4.3.1 (6)
Location→Log Contact→Intent

Zillow_5.7.257 (5)
PhoneData→Log Location→Log
PhoneData→Network

Figure 13: Actual leaks (i.e., non-false-positives) in Google Play Store shown as Source→Sink pairs. The number in parentheses (n) is the number of type errors reported by DroidInfer for the app (some of these n errors are false positives and do not have Source→Sink pairs). Sometimes several errors correspond to one Source→Sink pair. Some flows happen in advertising libraries such as InMobi, Millenial Media and Flurry, called from the apps. 5 of the 88 apps with type errors had only false positives or all type errors were refuted by CFL-Explain, and are not included here.

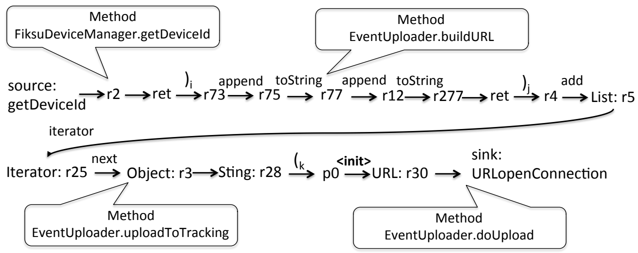


Figure 14: A source-sink path in Fiksu. When a flow is triggered by a library call, CFL-Explain labels the edge with the corresponding library method. When types change due to library calls, we show the new type at the target (e.g., List r5). We keep the identifiers exactly as they appear in the Jimple code.

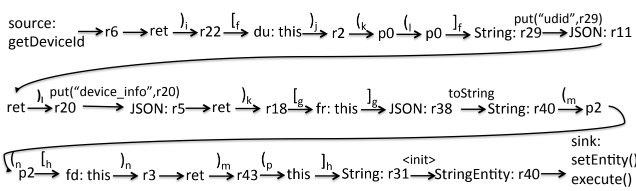


Figure 15: A source-sink path in Tremorvideo.

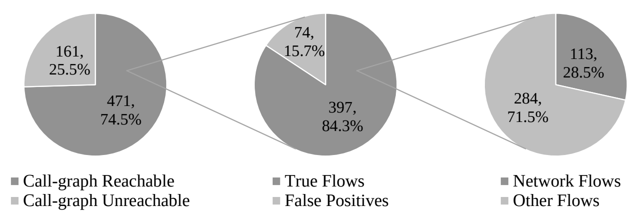


Figure 16: Results.

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 set to 2GB.) 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. 16 summarizes our results. Of those 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 and CFL-Explain prints source-sink paths instantly. This timings strongly support hypothesis **H3**.

Fig. 13 summarizes the leaks we found. In contrast to the FlowDroid researchers [1], who report no network flows, we uncover many network flows. Almost *one third* of all apps and almost *one 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.

We examined several representative network flows. 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. 14. 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 argument to method doUpload. DroidInfer reports an interesting type error in the Tremorvideo video ad library (com.tremorvideo.sdk.android.videoad.*) included in the **accuweather** app. The source-sink path reported by CFL-Explain is shown in Fig. 15. Note that the path has interleaving parentheses and brackets and yet the flow is quite obvious. The deviceId source is returned at callsite i and written into field f of the du object (names of classes and methods in Tremorvideo appear to have been obfuscated.) The du flows through several calls, until it's f field is retrieved into String r29 which is then put into a JSON object to form the uiud-deviceId key-value pair. This complex and yet feasible path attests to the power of DroidInfer and CFL-Explain.

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 would log so much sensitive info, usually in clear text, given that malicious apps may read the logs and retrieve the info (until Android 4.0, any app that held the READ_LOGS permission could read the logs).

One 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 r4 $\rightarrow$ r5 in Fig. 14, constructed from DroidInfer constraint r4 <: 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., *source* $\rightarrow$ r1 $\xrightarrow{[_f}$ UserActivity : r2 $\xrightarrow{\mathsf{getPackageName}}$ *sink*. We assume that the library method does not retrieve sensitive information and count these 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 of the reason discussed above, i.e., that 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,

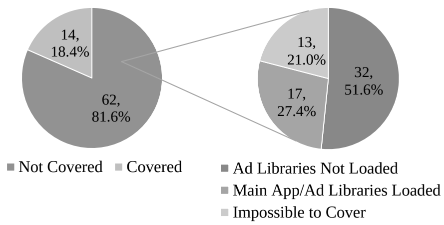


Figure 17: Runtime results.

as well as an undergraduate research assistant with minimal knowledge of program analysis.

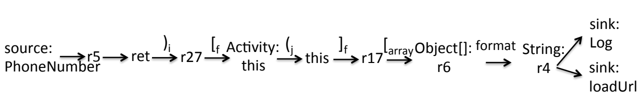
6.4.2 Runtime Results

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% of all errors. These errors span 8 apps and expose flows of DeviceId to both logs and the network. The flows are obvious tracking, as in Fig. 14, which is covered.

Fig. 17 summarizes the results. Of the 62 type errors we did not cover, 13 are beyond our runtime analysis. For example, there are several type errors reported as flows to the network. However, there are no logs around the network call and we cannot judge if the flow is covered or not.

This experiment shows that 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 never loaded 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.

We conclude this section with one of the most interesting flows we observed. When the user requests the details of a mortgage quote, the **Zillow** app grabs the phone number (and other info) stores it into an Object array, then proceeds to format the array into a URL string, log the string and send it to a loadUrl. The CFL-Explain paths are as follows:



The URL string with the phone number appears in the log!

6.4.3 Comparison with FlowDroid

We ran FlowDroid 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. DroidInfer's scales well because it avoids points-to analysis, while FlowDroid uses computationally expensive field- and object-sensitive points-to analysis. This result strongly supports hypothesis **H3** well.

[5]Eric Bodden: personal communication.

Of the remaining 114 apps, FlowDroid succeeds on 48. It reports more than 4000 flow paths over the 50 apps. We examined a random 21 of those apps and compared the results with DroidInfer. 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 "classical" flows in all 21 apps. FlowDroid reports thousands of flows from Bundle, Intent and Context, as it is overly-conservative in its handling of inter-process communication. These results are consistent with Artz et al. [1]. It is unclear why FlowDroid reports no network flows — it does specify DeviceId and Location as sources and 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. 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 [10] 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 [23] 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, which will provide more channels for privacy leaks.

In previous work we built SFlow [16], 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. Consider again the example in Fig. 2. SFlow generates these constraints at callsite 10: $\mathsf{y} <: \mathsf{y} \rhd \mathsf{tainted}$ $\quad \mathsf{src} <: \mathsf{y} \rhd \mathsf{poly}$ $\quad \mathsf{y} \rhd \mathsf{poly} <: \mathsf{srcId}$. Because $\mathsf{src} = \mathsf{tainted}$, y must be tainted. However, y being tainted does not satisfy the SFlow constraints at callsite 11: $\mathsf{y} <:$

$y \triangleright \mathsf{tainted} \quad \mathsf{sink} <: y \triangleright \mathsf{poly} \quad y \triangleright \mathsf{poly} <: \mathsf{sinkId}$
where both sink and sinkId are safe. This is because $y \triangleright \mathsf{poly} = \mathsf{tainted}$ is not a subtype of $\mathsf{sinkId} = \mathsf{safe}$. As a result, SFlow rejects this program, even though there is no flow from the source to the sink. In contrast, DFlow accepts this program as shown in Sect. 3.1. 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 therefore it can analyze any real-world Android app. Last but not least, the extensive evaluation on Google Play Store apps is a major contribution over our previous work. IFC [5] is another recent type-based taint analysis. It also works only on source and therefore cannot analyze real-world apps. Furthermore, it requires 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 DFlow, a context-sensitive information flow type system, and DroidInfer, the corresponding inference tool for detecting privacy leaks in Android apps. Empirical evaluation has shown that our approach is effective.

9. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. 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] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications.

[10] 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.

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

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

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

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

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

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

[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] 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.

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

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

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

[27] A. Milanova, W. Huang, and Y. Dong. Cfl-reachability and context-sensitive integrity types. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 99–109, New York, NY, USA, 2014. ACM.

[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.

APPENDIX

A. LISTS OF SOURCES AND SINKS

PhoneData

In package android.telephony:
TelephonyManager.getDeviceId()
TelephonyManager.getLine1Number()
TelephonyManager.getSimSerialNumber()
TelephonyManager.getSubscriberId()
SmsManager.getAllMessagesFromIcc()
SmsMessage.createFromPdu(byte[] arg0)
SmsMessage.createFromPdu(byte[] arg0, String arg1)
SmsMessage.newFromCMT(String[] arg0)
SmsMessage.newFromParcel(Parcel arg0)
SmsMessage.createFromEfRecord(int arg0, byte[] arg1)

Location

In package android.location:
LocationListener.onLocationChanged(Location arg0)
LocationManager.onLocationChanged(Location arg0)
LocationManager.getLastLocation()
LocationManager.getLastKnownLocation(String arg0)
ILocationListener.onLocationChanged(Location arg0)
In package com.millennialmedia:
android.MMRequest.getUserLocation()

Contact

android.provider.Contacts.android.net.Uri.CONTENT_URI
android.provider.ContactsContract.android.net.Uri.CONTENT_URI

Figure 18: Sources in DroidInfer: Location, PhoneData and Contact. In each category, the specific source is underlined for each item. If the source is a parameter of a method, the parameter is underlined, e.g. onLocationChanged(Location arg0). If the source is the return value, the method name is underlined, e.g. getLastLocation(). If the source is a field of a class, the field name is underlined, e.g. Uri.CONTENT_URI.

Log

In package android.util:
Log.d(String arg0, String arg1)
Log.d(String arg0, String arg1, Throwable arg2)
Log.e(String arg0, String arg1)
Log.e(String arg0, String arg1, Throwable arg2)
Log.i(String arg0, String arg1)
Log.i(String arg0, String arg1, Throwable arg2)
Log.println(int arg0, String arg1, String arg2)
Log.println_native(int arg0, int arg1, String arg2, String arg3)
Log.v(String arg0, String arg1)
Log.v(String arg0, String arg1, Throwable arg2)
Log.w(String arg0, String arg1)
Log.w(String arg0, String arg1, Throwable arg2)
Log.w(String arg0, Throwable arg1)
Log.wtf(String arg0, String arg1)
Log.wtf(String arg0, String arg1, Throwable arg2)

Network

android.telephony.SmsManager.sendDataMessage(String arg0, String arg1,
short arg2, byte[] arg3, PendingIntent arg4, PendingIntent arg5)
android.telephony.SmsManager.sendTextMessage(String arg0,
String arg1, String arg2, PendingIntent arg3, PendingIntent arg4)
android.webkit.WebView.loadUrl(String arg0, Map<String,String> arg1)
android.webkit.WebView.loadUrl(String arg0)
android.webkit.WebView.postUrl(String arg0, byte[] arg1)
com.millennialmedia.android.MMRequest.setUserLocation(Location arg0)
java.net.URL.openConnection(Proxy proxy) this
java.net.URL.getOutputStream() this
java.net.Socket.getOutputStream() this
org.apache.http.client.methods.HttpPost.setEntity(HttpEntity arg0)
org.apache.http.client.methods.HttpEntityEnclosingRequestBase
.setEntity(HttpEntity arg0)

Intent

In package android.app:
Activity.setResult(int arg0, Intent arg1)
Activity.startActivities(Intent[] arg0)
Activity.startActivities(Intent[] arg0, Bundle arg1)
Activity.startActivity(Intent arg0)
Activity.startActivity(Intent arg0, Bundle arg1)

Intent (continued)

In package android.app:
Activity.startActivityAsUser(Intent arg0, Bundle arg1, UserHandle arg2)
Activity.startActivityAsUser(Intent arg0, UserHandle arg1)
Activity.startActivityForResult(Intent arg0, int arg1)
Activity.startActivityForResult(Intent arg0, int arg1, Bundle arg2)
Activity.startActivityFromChild(Activity arg0, Intent arg1, int arg2)
Activity.startActivityFromChild(Activity arg0, Intent arg1, int arg2, Bundle arg3)
Activity.startActivityFromFragment(Fragment arg0, Intent arg1, int arg2)
Activity.startActivityFromFragment(Fragment arg0, Intent arg1,
int arg2, Bundle arg3)
Activity.startActivityIfNeeded(Intent arg0, int arg1)
Activity.startActivityIfNeeded(Intent arg0, int arg1, Bundle arg2)
Activity.startIntentSender(IntentSender arg0, Intent arg1, int arg2,
int arg3, int arg4)
Activity.startIntentSender(IntentSender arg0, Intent arg1, int arg2,
int arg3, int arg4, Bundle arg5)
Activity.startIntentSenderForResult(IntentSender arg0, int arg1,
Intent arg2, int arg3, int arg4, int arg5)
Activity.startIntentSenderForResult(IntentSender arg0, int arg1,
Intent arg2, int arg3, int arg4, int arg5, Bundle arg6)
Activity.startIntentSenderFromChild(Activity arg0, IntentSender arg1,
int arg2, Intent arg3, int arg4, int arg5, int arg6)
Activity.startIntentSenderFromChild(Activity arg0, IntentSender arg1,
int arg2, Intent arg3, int arg4, int arg5, int arg6, Bundle arg7)
Activity.startNextMatchingActivity(Intent arg0)
Activity.startNextMatchingActivity(Intent arg0, Bundle arg1)
In package android.content:
Context.startActivity(Intent arg0)
Context.startActivityAsUser(Intent arg0, UserHandle arg1)
Context.startActivity(Intent arg0, Bundle arg1)
Context.startActivityAsUser(Intent arg0, Bundle arg1, UserHandle arg2)
Context.startActivities(Intent[] arg0)
Context.startActivities(Intent[] arg0, Bundle arg1)
Context.startActivitiesAsUser(Intent[] arg0, Bundle arg1, UserHandle arg2)
Context.startIntentSender(IntentSender arg0, Intent arg1, int arg2,
int arg3, int arg4)
Context.startIntentSender(IntentSender arg0, Intent arg1, int arg2,
int arg3, int arg4, Bundle arg5)
Context.startService(Intent arg0)
Context.startServiceAsUser(Intent arg0, UserHandle arg1)

Figure 19: Sinks in DroidInfer: Log, Network and Intent. The specific sink is underlined for each item. If the sink is a parameter, the parameter is underlined, e.g. Log.d(String arg0, String arg1). If the sink is the return value, the method name is underlined, e.g. getOutputStream().