I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis

Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon
SnT
University of Luxembourg
firstName.lastName@uni.lu

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

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

ABSTRACT

Android applications may leak privacy data carelessly or maliciously. In this work we perform inter-component dataflow analysis to detect privacy leaks between components of Android applications. Unlike all current approaches, our tool, called IccTA, propagates the context between the components, which improves the precision of the analysis. IccTA outperforms all other available tools by reaching a precision of 95.0% and a recall of 82.6% on DroidBench. Our approach detects 147 inter-component based privacy leaks in 14 applications in a set of 3000 real-world applications with a precision of 88.4%. With the help of ApkCombiner, our approach is able to detect inter-app based privacy leaks.

1. INTRODUCTION

With the growing popularity of Android, thousands of applications (also called apps) emerge every day on the official Android market (Google Play) as well as on some alternative markets. As of May 2013, 48 billion apps have been installed from the Google Play store, and as of September 3, 2013, 1 billion Android devices have been activated [1]. Researchers have shown that Android apps frequently send the user's private data outside the device without the user's prior consent [29]. Those applications are said to *leak* private data. Android applications are made of different components; most of the privacy leaks are simple and operate within a single component. More recently, cross-component and also cross-app privacy leaks have been reported [26]. Analyzing components separately is not enough to detect such leaks. Therefore, it is necessary to perform an inter-component analysis of applications. Android app analysts could leverage such a tool to identify malicious apps that leak private data. For the tool to be useful, it has to be highly precise and minimize the false positive rate when reporting applications leaking private data.

Privacy leaks. In this paper, we use a static taint analysis technique to find privacy leaks, i.e., paths from sensitive data, called *source*s, to statements sending the data outside the application or device, called *sink*s. A path may be within a single component or cross multiple components and/or applications.

State-of-the-art approaches using static analysis to detect privacy leaks on Android apps mainly focus on detecting intra-component sensitive data leaks. CHEX [18], for example, uses static analysis to detect component hijacking vulnerabilities by tracking taints between sensitive sources and sinks. DroidChecker [9] uses inter-procedural Control-Flow Graph (CFG) searching and static taint checking to detect exploitable data paths in an Android application. FlowDroid [4] also performs taint analysis within single components of Android applications but with a better precision. In this paper, we not only focus on intra-component leaks, but we also consider Inter-Component Communication (ICC) based privacy leaks, including Inter-Application Communication (IAC) leaks.

Other approaches use dynamic tracking to find privacy leaks. For instance, TaintDroid [12] leverages Android's virtualized execution environment to monitor Android apps at runtime in which it tracks how application leaks private information. CopperDroid [20] dynamically observes interactions between the Android components and the underlying Linux system to reconstruct higher-level behavior.

A dynamic approach must send input data to the app at runtime to trigger code execution. The input data may be incomplete and thus not execute all parts of the code. Furthermore, some code may only be executed if precise conditions are met at runtime such as a data. In this paper, we focus on static analysis to avoid these drawbacks. The counterpart of static analysis is that it may yield an over-approximation since it analyzes all code even the one that could never be executed.

Static taint analysis for Android is difficult. Despite the fact that Android applications are mainly programmed in Java, off-the-shelf static taint analysis tools for Java do not work on Android applications. The tools need to be adapted mainly for three reasons. The first reason is that, as already mentioned, Android applications are made of components. Communications between components involve two main artifacts: *Intent Filter* and *Intent*. An *Intent Filter* is attached to a component and "filters" *Intents* that can reach the component. An *Intent* is used to start a new component by first dynamically creating an *Intent* instance, and then by calling a specific method (e.g. *startActivity*, *startService*) with the *intent* previously created as parameter. The *intent* is used either explicitly by specifying the new component to

call, or implicitly by for instance only specifying the action[1] to perform. The launch of a component is performed by the Android system which "resolves" the matching between *Intent* and *Intent Filter* at runtime. This dynamic resolution done by the Android system induces a discontinuity in the control-flow of Android applications. This specificity makes static taint analysis challenging by requiring pre-processing of the code to resolve links between components.

The second reason is related to the user-centric nature of Android applications, in which a user can interact a lot through the touch screen. The management of user inputs is mainly done by handling specific callback methods such as the *onClick* method which is called when the user clicks on a button. Static analysis requires a precise model that stimulates users' behavior.

The third and last reason is related to the lifecycle management of the components. There is no *main* method as in a traditional java program. Instead, the Android system switches between states of a component's lifecycle by calling callback methods such as *onStart*, *onResume* or *onCreate*. However, these lifecycle methods are not directly connected in the code. Modeling the Android system allows to connect callback methods to the rest of the code.

Our Proposal. The above challenges will unavoidably cause some discontinuities in the control-flow graph. To overcome these issues, we present an Inter-component communication Taint Analysis tool named IccTA[2]. IccTA allows a sound and precise detection of ICC and IAC links. This approach is generic and can be used for any data-flow analysis. In this paper we focus on using IccTA to detect privacy leaks.

IccTA is based on three software artifacts: Epicc-IccTA, FlowDroid-IccTA and ApkCombiner.

Epicc-IccTA extends Epicc [19] which computes ICC links between Android components. Epicc-IccTA leverages Epicc to incrementally store the computed ICC links to a database for conveniently analyzing a large set of apps. FlowDroid-IccTA extends FlowDroid [4]. FlowDroid only finds privacy leaks within single components of Android applications but not between components.

FlowDroid-IccTA uses ICC links computed by Epicc to improve FlowDroid. Based on these computed links, FlowDroid-IccTA modifies Android applications' code to directly connect components to enable data-flow analysis between components. By doing this, we build a complete control-flow graph of the whole Android application. This allows propagating the context between Android components and yielding a highly precise data-flow analysis. To the best of our knowledge, this is the first approach that precisely connects components for data-flow analysis.

Finally, ApkCombiner helps analyzing multiple Android applications by combining multiple apps into one when there exist data flows between these apps. This results in having a complete control-flow graph of the combined apps. This allows to propagate the context not only between components of a single app but also between components of different apps.

To verify our approach, we developed 26 apps containing ICC based privacy leaks. We have added these applications to DroidBench [2], an open test suite for evaluating the effectiveness and accuracy of taint analysis tools specifically for Android apps. The 26 apps cover the top 8 used ICC methods illustrated in Table 1.

Table 1: The top 8 used ICC methods[†]

ICC Method	Counts(#.)	Used Apps(#.)
startActivity	55802 (61.44%)	2765 (92.2%)
startActivityForResult	11095 (12.21%)	1980 (66.0%)
query	6606 (7.27%)	1601 (53.4%)
startService	3942 (4.34%)	1077 (35.9%)
sendBroadcast	3472 (3.82%)	790 (26.3%)
insert	2100 (2.31%)	615 (20.5%)
bindService	1515 (1.67%)	644 (21.5%)
delete	1238 (1.36%)	350 (11.7%)
Other ICC Methods	5058 (5.57%)	-
Total	90828 (100%)	-

[†] Methods with higher counts are selected when overload methods exist

Contributions. To summarize, we present the following original contributions in this paper:

- A novel methodology to resolve the ICC problem by directly connecting the discontinuities of Android apps at the code level.
- IccTA, a tool for inter-component data-flow analysis.
- An improved version of DroidBench with 26 new apps to evaluate tools detecting ICC based privacy leaks.
- An empirical study to evaluate IccTA over an augmented version of the DroidBench test suite (available online[3]) and 3000 real-world Android applications.

2. BACKGROUND

2.1 Android ICC Methods

An Android application is made of basic units, called components, described in a special file, called *Manifest*, stored in the application. There are four types of components: a) Activities that represent the user interface and are the visible part of Android applications; b) Services which execute tasks in background; c) Broadcast Receivers that receive messages from other components or the system, such as incoming calls or text messages; and d) Content Providers which act as the standard interface to share structured data between applications.

Some specific Android system methods are used to trigger inter-component communication. We call them Inter-Component Communication (ICC) methods. Those methods take as parameter a special kind of object, called *Intent*, which specifies the target component(s). We perform a short study to compute the usage rate of ICC methods. We analyzed 3000 Android applications randomly selected from Google Play and other third party markets. Table 1 shows the top 8 most used ICC methods. The third column represents the number of apps using at least once the corresponding ICC method. The most used ICC method is `startActivity`, used to launch a new Activity component, which accounts for 59.2% of the total detected ICC methods.

All ICC methods[4] take at least one *Intent* in their parameters to specify the target component(s). There are two

[1]Such as `android.intent.action.VIEW` or `.CALL` or `.EDIT`

[2]Our experimental results and IccTA itself are available at https://sites.google.com/site/icctawebpage.

[3]github.com/secure-software-engineering/DroidBench

[4]Except `Content Provider` related methods such as `query` or `insert`

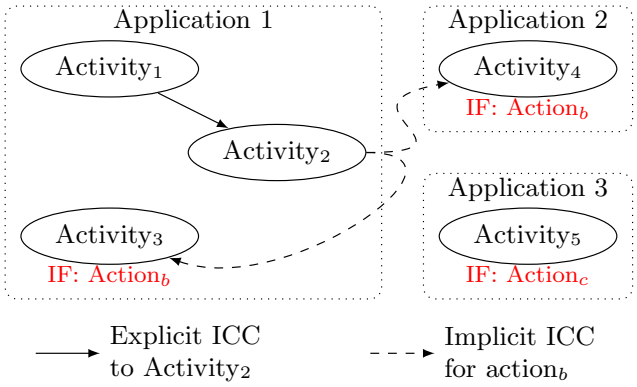

Figure 1: Explicit and Implicit ICC between Components of Android Applications.

ways to specify ICC method's target components. The first one is by explicitly specifying them by setting the name of the target components through an *Intent*. The second one is by implicitly specifying them by setting the *action*, *category* and *data* fields of an *Intent*. In order to receive implicit *Intents*, target components need to specify an *Intent Filter* in their application's manifest file. Note that *Intents* can transfer data between components.

Again, we performed a short study on the 3000 apps to compute the ratio between explicit and implicit *Intents* for the `startActivity` ICC method. Among the 55,802 `startActivity` method calls, 27978 use explicit *intents* and 27824 use implicit *Intents*.

Figure 1 represents three Android apps made of Activity components. There is an explicit ICC from $\texttt{Activity}_1$ to $\texttt{Activity}_2$ in `Application 1`. There are two implicit ICCs from $\texttt{Activity}_2$ to $\texttt{Activity}_3$ in `Application 1` and from $\texttt{Activity}_2$ to $\texttt{Activity}_4$ between `Application 1` and `Application 2`. Note that the target components of implicit ICC, $\texttt{Activity}_3$ and $\texttt{Activity}_4$, have an *Intent Filter* with the same action and category value as the *Intent* used in $\texttt{Activity}_2$. Each time there is an ICC, there may be a flow of data between components and potentially a privacy leak.

2.2 FlowDroid

FlowDroid [4] is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications. FlowDroid is based on Soot [16] and Heros [8]. The context-, flow-, field-, object-sensitives of FlowDroid are guaranteed by the precise call-graph of Soot and the IFDS/IDE [21, 23] based data-flow analysis of Heros. A special main method, which considers all combinations of lifecycles, callbacks and entry points of Android components is generated to model data flows within the application. The sources and sinks used by FlowDroid are provided by SuSi [3], also an open sourced tool used to fully automatically classify and categorize Android sources and sinks. FlowDroid achieves 93% recall and 86% precision when detecting data leaks on DroidBench. FlowDroid has been mainly used on single component. However, with slight modifications, FlowDroid could also be used when multiple components are involved, i.e., for ICC analyses. Indeed, it's possible to use FlowDroid to compute paths for all individual components and then combines all those paths together, whatever there is a real link or not between these components. A major drawback is that this naive approach yields many false positives.

2.3 Epicc

Epicc [19] is a tool, also based on Soot and Heros, to identify ICC links. In other words it finds links from ICC methods to their target components. Epicc reduces the discovery of ICC in Android to an instance of the Interprocedural Distributive Environment (IDE) problem [23]. It uses data flow analysis to compute *Intent* values at every ICC method call statements. Experiments show that Epicc identifies 93% of all ICC links and finds ICC vulnerabilities with far fewer false positives than the next best tool.

3. MOTIVATING EXAMPLE

This section motivates our approach and illustrates the problem we solve through a concrete example. This example is detailed in Listing 1, which presents code of `Application 1` introduced in Figure 1. The app has three Activity components represented by $\texttt{Activity}_1$, $\texttt{Activity}_2$ and $\texttt{Activity}_3$ classes. It also features `ButtonOnClickListener` a listener class used to handle button click events. $\texttt{Activity}_1$ registers a button listener for the *to2* button (lines 6-11) and $\texttt{Activity}_2$ registers one for the *to3* button (line 15).

```
1  //TelephonyManager telMnger; (default)
2  //SmsManager sms; (default)
3  class Activity1 extends Activity {
4   void onCreate(Bundle state) {
5    Button to2 = (Button) findViewById(to2a);
6    to2.setOnClickListener(new OnClickListener(){
7     String id = telMnger.getDeviceId();
8     Intent i = new
        Intent(Activity1.this,Activity2.class);
9     i.putExtra("sensitive", id);
10    Activity1.this.startActivity(i);
11   });}}
12 class Activity2 extends Activity {
13  void onCreate(Bundle state) {
14   Button to3 = (Button) findViewById(to3a);
15   to3.setOnClickListener(new
       ButtonOnClickListener(this));
16   Intent i = getIntent();
17   String s = i.getStringExtra("sensitive");
18   sms.sendTextMessage(number,null,s,null,null);
19  }
20  void onActivityResult(int,int,Intent){
21   //log all the Extras of Intent
22  }}
23 class Activity3 extends Activity {
24  void onCreate(Bundle state) {
25   Intent i = getIntent();
26   String s = i.getStringExtra("sensitive");
27   sms.sendTextMessage(number,null,s,null,null);
28  }}
29 class ButtonOnClickListener extends
       OnClickListener{
30  //Activity act; (construct)
31  void onClick(View view) {
32   String id = telMnger.getDeviceId();
33   Intent i = new Intent();
34   i.setAction("test.ACTION"); //Action b
35   i.putExtra("sensitive", id);
36   act.startActivityForResult(i, 1);
37  }}
```

Listing 1: A Motivating Example Code

When button *to2* and *to3* are clicked, the `onClick` method is executed and the user interface will change to $\texttt{Activity}_2$ and to $\texttt{Activity}_3$, respectively. In both cases, an `Intent` containing the device ID (lines 7 and 32), considered as sensitive data, is sent between two components by first attach-

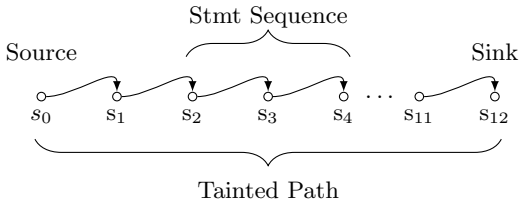

Figure 2: Representation of Statements, Source, Sink, Statement Sequence and Tainted Path.

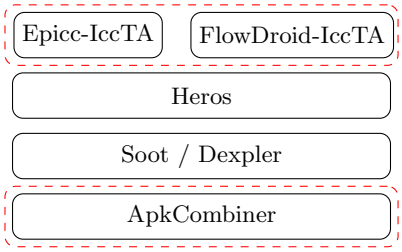

Figure 3: The architecture of IccTA

ing the data to the intent with the `putExtra` method (lines 9, and 35) and then by invoking either `startActivity` or `startActivityForResult` (lines 10 and 36). Note that Listing 1 exemplifies both the use of explicit and implicit intents. At line 8, the intent is created by explicitly specifying the target class ($\texttt{Activity}_2$). At line 34, only the intent action is specified with no explicit reference to the target.

In this example, `sendTextMessage` is directly executed when $\texttt{Activity}_2$ or $\texttt{Activity}_3$ is loaded since `onCreate` is the first method in the lifecycle of an `Activity`. It sends the data retrieved from the `Intent` as a SMS to the specified phone number.

In this code, two privacy leaks occur: one when button *to2* is clicked, the other when button *to3* is clicked. When *to2* is clicked, the device ID is transferred from $\texttt{Activity}_1$ to $\texttt{Activity}_2$ (line 10) and then $\texttt{Activity}_2$ sends it outside the application (line 18).

When *to3* is clicked, the device ID is transferred (line 36) from $\texttt{Activity}_2$ to $\texttt{Activity}_3$[5]. Actually, the device ID (the source) is retrieved in class `ButtonOnClickListener` instantiated by $\texttt{Activity}_2$. Finally, $\texttt{Activity}_3$ sends the device ID outside the application (line 27).

The sensitive data leaks described above crosses two components: they cannot directly be detected since there is no real code connection between `startActivity` and `onCreate` (lines 10 and 13) or between `startActivityForResult` and `onCreate` (lines 36 and 24). Section 5 describes our approach to connect components to analyze paths between components and even between applications.

4. DEFINITIONS

In order to better describe our approach, some android and taint analysis related concepts need to be defined.

Control-Flow Graph (CFG) We detect data leaks by analyzing control-flow graphs of Android applications. An application CFG consists of a collection of method CFGs linked together according to how they call one another.

Source Method. A source method returns data considered as private from the user's point of view into the application code. For example, method `getDeviceId` (line 7[6]) is a source method returning the device ID.

Sink Method. A sink method sends data out of the application. For example, method `sendTextMessage` (line 27) is a sink method sending data to another phone using SMS. We use sources and sinks computed for Android by the SuSi tool [3].

[5]As illustrated in Figure 1, $\texttt{Activity}_3$ has the appropriate *Intent Filter* to catch the implicit *Intent*

[6]All the line numbers described in this section is referring to Listing 1

ICC Method. An ICC method is used to trigger communication between two components. For example, method `startActivity` (line 10) is an ICC method which triggers component communication from $\texttt{Activity}_1$ to $\texttt{Activity}_2$.

Tainted Stmt. A tainted statement contains at least one tainted piece of data. For example, `i.putExtra("sensitive",id)` (line 9) is a statement containing the tainted data `id`.

Tainted Stmt Sequence. A tainted stmt sequence is a flow-sensitive sequence of tainted stmt. For instance statements at line 9 and 10 form a tainted statement sequence.

Tainted Path. A tainted path is a tainted stmt sequence where 1) More than one stmt exist in the tainted path; 2) The first stmt contains a source method; 3) The last stmt contains a sink method. Tainted Stmt, Tainted Stmt Sequence and Tainted Path are illustrated in Figure 2.

There are three types of tainted stmt paths in Android: Intra-Component Communication, Inter-Component Communication (ICC) and Inter-Application Communication (IAC) based tainted paths.

Intra-Component Tainted Path. An intra-component tainted path is a tainted path only happening within a component. In our motivating example, there is no intra-component tainted path. But if the `startActivity` call was replaced with a call to `sendTextMessage` which sends the device id out of the application, there would be an intra-component tainted path (line 7-10).

ICC based Tainted Path. An ICC based tainted path is a tainted path among two or more components, i.e., there is at least one ICC method in the path. In our motivating example, there is an ICC based tainted path from source method `getDeviceId` in $\texttt{Activity}_1$ to sink method `sendTextMessage` in $\texttt{Activity}_2$ through the `startActivity` ICC method (line 10).

IAC based Tainted Path. An IAC based tainted path is a tainted path between two or among more applications, i.e., it has at least one ICC method between two components of different applications. There is no IAC based tainted path in our motivating example. But if the $\texttt{Activity}_4$ in Figure 1 sends the device id transferred from $\texttt{Activity}_2$ out of the application, then there is an IAC based tainted path from `Application 1` to `Application 2`.

Privacy Leaks. If a tainted path is detected, it means that a privacy leak has been found. In other words, some private data obtained from a *source* method can flow through the tainted path to a *sink* method.

5. ICCTA

In this Section we describe IccTA, our tool to detect privacy leaks in Android applications. It uses static taint analysis to detect privacy leaks. The main challenge for this is to

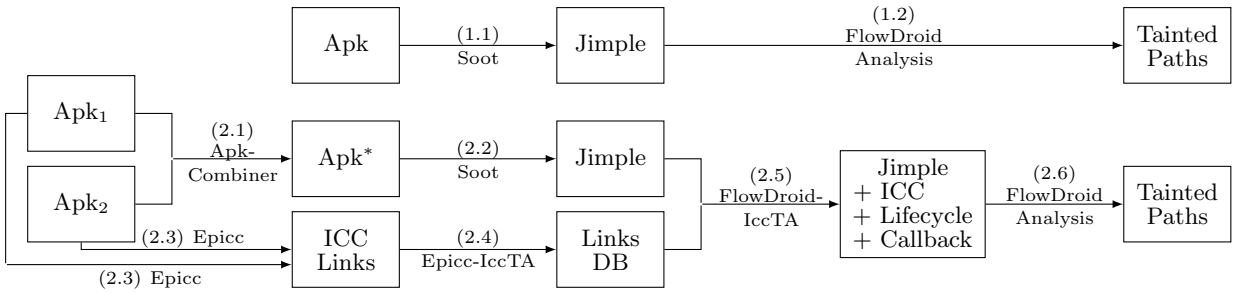

Figure 4: Overview of IccTA (down) and FlowDroid (up).

solve the discontinuities problem introduced by the Android system.

We present the architecture of IccTA in Figure 3 where new or modified component are surrounded by a dashed line. IccTA is the combination of Epicc-IccTA and FlowDroid-IccTA. Epicc-IccTA relies on Epicc to incrementally compute ICC links from Android apps. Both FlowDroid and Epicc are based on Soot [16] and Heros [8]. Soot is a framework to analyze Java based applications. It uses the Dexpler [7] plugin to convert Android Dalvik byte code to Soot's internal representation called Jimple and relies on Spark [17] to build accurate call graphs. Heros is a scalable implementation of IFDS [22] and IDE [23], two frameworks to perform data flow analysis. Analyzing multiple applications is done using ApkCombiner. It combines multiple apps to a single one to ease the analysis of IccTA.

Figure 4 is a comparison between IccTA and FlowDroid. FlowDroid (4 up) first converts the Android bytecode to Jimple in step (1.1). Then, in step (1.2), it analyzes the Jimple code to detect tainted paths in single Android components.

IccTA (4 down) can analyze one or multiple Android applications. If more than one application is analyzed, it uses ApkCombiner to merge the Android applications in a single application in step (2.1). The Android application's bytecode is then converted to Jimple in step (2.2). In parallel, Epicc-IccTA analyzes all the input applications (Apk_1 and Apk_2 in the Figure) to generate ICC Links in step (2.3) and stores the results to a database in step (2.4). IccTA uses ICC links generated by Epicc-IccTA to connect Android components in the Jimple code in step (2.5). Steps (2.2) and (2.6) correspond to FlowDroid's steps (1.1) and (1.2): the Jimple code is updated to take into account lifecycles and callbacks of components and the taint analysis is launched to generate a list of tainted paths.

5.1 FlowDroid-IccTA: Reducing the ICC problem to an Intra-Component problem

Since there is no direct code connection between two Android components, FlowDroid cannot detect ICC based privacy leaks with precision. In this section, we describe how FlowDroid-IccTA reduces the ICC problem to an intra-component problem on which FlowDroid can perform an highly precise data-flow analysis. Our approach instruments the *Jimple* code of Android applications to connect components directly in the code.

As mentioned in the introduction, there are three types of discontinuities in Android: (1) ICC methods, (2) lifecycle methods and (3) callback methods. We first describe how FlowDroid-IccTA tackles ICC methods in Section 5.1.1. Then, we detail how FlowDroid-IccTA resolves lifecycle and callback methods in Section 5.1.2. Finally, using our motivating example of Listing 1, we illustrate the code instrumentation process in Section 5.1.3.

(A)
```
// modifications of Activity1
Activity1.this.startActivity(i);
IpcSC.redirect0(i);
```

(B)
```
// creation of a helper class
class IpcSC {
 static void redirect0(Intent i) {
  Activity2 a2 = new Activity2(i);
  a2.dummyMain();
 }
}
```

(C)
```
// modifications in Activity2
public Activity2(Intent i) {
  this.intent_for_ipc = i;
}
public Intent getIntent() {
 return this.intent_for_ipc;
}
public void dummyMain() {
 // lifecycle and callbacks
 // are called here
}
```

Figure 5: Code Modifications to Handle ICC Communication between $Activity_1$ and $Activity_2$. The `startActivity` ICC method is replaced (A) by a call to code that instantiates and calls the "main" method of $Activity_2$ (B). The target component class is updated to handle Intent objects directly, by modeling the Android system behavior (C).

5.1.1 ICC Methods

As shown in Figure 4, the ICC problem is solved at step 2.5. This is where the *Jimple* code is updated by FlowDroid-IccTA to connect components. This code modification is required for all ICC methods (listed in Table 1). We detail these modifications for the two most used ICC methods: `startActivity` and `startActivityForResult`. We handle ICC methods for *Service*s and *Broadcast Receiver*s in a similar way.

StartActivity. Figure 5 shows the code transformation done by FlowDroid-IccTA for the ICC link between $\texttt{Activity}_1$ and $\texttt{Activity}_2$ of our motivating example. FlowDroid-IccTA first creates a helper class named `IpcSC` (B in Figure 5) which acts as a bridge connecting the source and destination components. Then, the `startActivity` ICC method is removed and replaced by a statement calling the generated helper method (`redirect0`) (A).

In (C), FlowDroid-IccTA generates a constructor method taking an `Intent` as parameter, a `dummyMain` method to call all related methods of the component (i.e., lifecycle and callback methods) and overrides the `getIntent` method. An Intent is transferred by the Android system from the caller component to the callee component. We model the behavior of the Android system by explicitly transferring the Intent to the destination component using a customized constructor method, $\texttt{Activity}_2\texttt{(Intent i)}$, which takes an `Intent` as its parameter and stores the Intent to a newly generated

field `intent_for_ipc`. The original `getIntent` method asks the Android system for the incoming Intent object. The new `getIntent` method models the Android system behavior by returning the Intent object given as parameter to the new constructor method.

The helper method `redirect0` constructs an object of type $\texttt{Activity}_2$ (the target component) and initializes the new object with the `Intent` given as parameter to the helper method. Then, it calls the `dummyMain` method of $\texttt{Activity}_2$.

To resolve the target component, i.e., to automatically infer what is the type that has to be used in the method `redirect0` (in our example, to infer $\texttt{Activity}_2$), Flowdroid-IccTA uses the ICC links computed by Epicc-IccTA. Epicc-IccTA resolve the target component not only for explicit *intents*, but also for implicit *intents*. Therefore, there is no difference for Flowdroid-IccTA to handle explicit or implicit *intent* based ICCs.

StartActivityForResult. There are some special ICC methods in Android, such as `startActivityForResult`. A component C_1 can use this method to start a component C_2. Once C_2 finishes running, C_1 runs again with some result data returned from C_2. The control-flow mechanism of `startActivityForResult` is shown in Figure 6. There are two discontinuities: one from (1) to (2), similar to the discontinuity of the `startActivity` method, and the other from (3) to (4).

The `startActivityForResult` ICC method has a more complex semantic compared to common ICC methods that only trigger one-way communication between components (e.g., `startActivity`). Figure 7 shows how the code is instrumented to handle the `startActivityForResult` method in our motivating example. To stay consistent with common ICC methods, we do not instrument the `finish` method of $\texttt{Activity}_3$ to call `onActivityResult` method. Instead, we generate a field `intent_for_ar` to store the *Intent* which will be transferred back to $\texttt{Activity}_2$. The *Intent* that will be transfered back is set by the `setResult` method. We override the `setResult` method to store the value of *Intent* to `intent_for_ar`. The helper method `IpcSC.redirect0` does two modifications to link these two components directly. First, it calls the `dummyMain` method of destination component. Then, it calls the `onActivityResult` method of the source component.

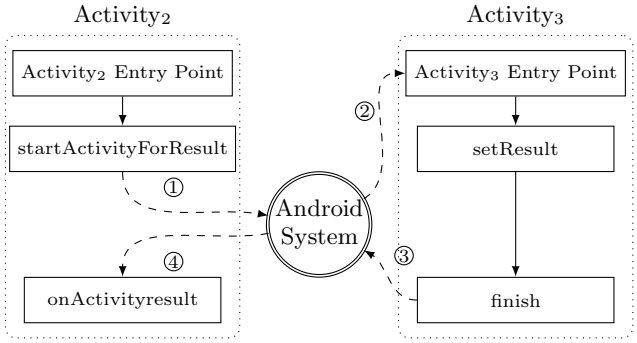

Figure 6: The control-flow mechanism of `startActivityForResult`

5.1.2 Lifecycle and Callback Methods

One challenge when analyzing Android applications is to tackle the callback methods and the lifecycle methods of components. There is no direct call among those methods

```
(A)
act.startActivityForResult(i);
IpcSC.redirect0(act, i);

(C)
void setResult(Intent i) {
 this.intent_for_ar = i;
 a2.dummyMain();
}
public Intent getIntentFAR() {
 return this.intent_for_ar;
}
```

```
(B)
class IpcSC {
 static void redirect0(Activity a2,
                       Intent i) {
  Activity3 a3 = new Activity3(i);
  a3.dummyMain();
  Intent retI = a3.getIntentFAR();
  a2.onActivityResult(retI);
 }
}
```

Figure 7: An Example about running FlowDroid-IccTA to `startActivityForResult` ICC method. (A) represents the modified code of `ButtonOnClickListener` and (C) the modified code of $\texttt{Activity}_3$. (B) is the glue code connecting `ButtonOnClickListener` and $\texttt{Activity}_3$. Some method parameters are not represented to simplify the code.

in the code of applications since the Android system handles lifecycles and callbacks. For callback methods, we need to take care of not only the methods triggered by the User Interface (UI) events (e.g., `onClick`) but also of callbacks triggered by Java or the Android system (e.g., the `onCreate` method). In Android, every component has its own lifecycle methods. To solve this problem, IccTA generates a `dummyMain` method for each component in which we model all the methods mentioned above so that our CFG based approach is aware of them. Note that FlowDroid also generates a `dummyMain` method, but it is generated for the whole app instead of for each component like we do.

5.1.3 The CFG of instrumented motivating example

Figure 8 represents the CFG of the instrumented motivating example presented in Listing 1. In the CFG, `getDeviceId` is a *source* method in the anonymous `OnClickListener` class (line 6) called by $\texttt{Activity}_1$. Method `sendTextMessage` is a *sink* in $\texttt{Activity}_2$. There is an intra-component tainted statement path from the *source* method to *sink* method (represented by edges 1 to 12).

Figure 8 also shows that IccTA builds a precise cross-component control-flow graph. Since we use an technique instrumenting the code to build the CFG, the context of a static analysis is kept between components. This enables IccTA to analyze data-flows between components and thereby enables IccTA to have a better precision than existing approaches.

5.2 ApkCombiner: Reducing an IAC problem to an ICC problem

In Android, Inter-Application Communication (IAC) is similar to Inter-Component Communication (ICC). Indeed, IAC also relies on component communication, except that the source component and the destination component belong to different applications. If we can connect applications, an *IAC Problem* becomes a standard *ICC Problem*.

Analyzing Multiple Applications. As shown in Figure 4, FlowDroid can only analyse one application at a time. Therefore, we develop a tool, *ApkCombiner*, to combine multiple apps into one. ApkCombiner combines all the parts of Android apps including bytecodes, assets, manifest and all the resources. Then, we use IccTA to analyze the combined app to compute IAC based privacy leaks. As FlowDroid-IccTA handles the combined application as a single applications, it only detects ICC based privacy leaks. To distinguish ICC leaks from IAC leaks, IccTA checks if all statements of the tainted path belong to the same application or not.

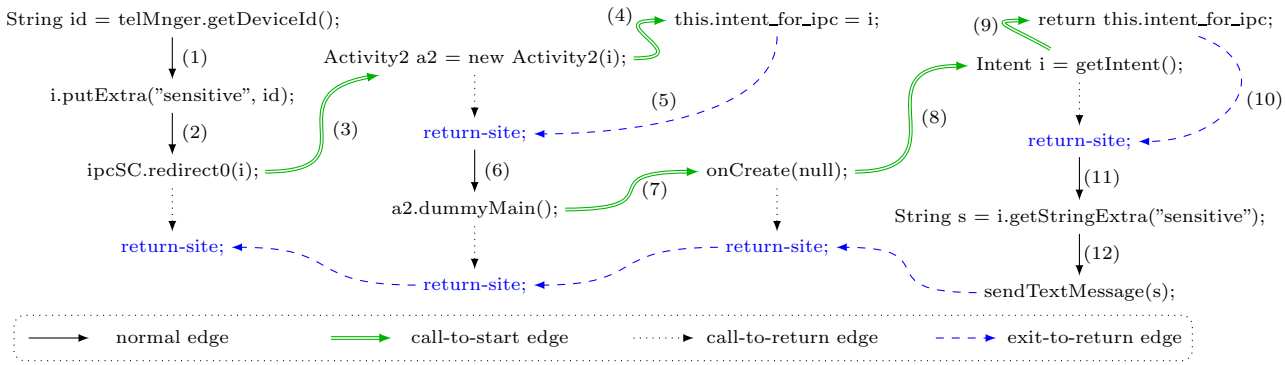

Figure 8: The control-flow graph of the instrumented motivating example

Reducing the Number of Combined Apps to Analyze. In practice, when increasing the number of applications to analyze, and if all those applications are combined with ApkCombiner, the processing time and memory requirement of FlowDroid-IccTA also grows. To solve this problem, we need to decrease the number of Android apps to combine. Our solution is to build an IAC graph, where a node is an application and an edge a link, to represent the dependencies between applications. The idea behind being that if there is no link between two applications there is no need to combine them.

The IAC graph is made up of small independent IAC (sIAC) graphs (connected components). Given a sIAC graph, ApkCombiner combines all the nodes (apps) in it into one app, then IccTA extracts leaks from the resulting app. However, in some case, if a sIAC graph still contains a lot of nodes. This will also limit our approach to be scalable. Our solution is to limit the length (how many apps are involved) of an IAC leak[7]. For example, If a sIAC graph contains 10 nodes (where $\texttt{A}_i$ is connected to $\texttt{A}_{i+1}$, $i \in \{1, 9\}$) and the length limitation is set to five. Then, the sIAC graph is split into five sIACs (e.g., one sIAC is from `A2` to `A6`) that IccTA can analyze. The trade-off limitation length enables our approach to become scalable.

Another good point of building an IAC graph is that new applications can be added to the graph in an iterative and incremental manner. When new apps are involved, we only run them against Epicc-IccTA and add them to the existing IAC graph. We do not need to run the previously computed apps again when adding the new apps to the IAC graph.

In short, by building an IAC graph, the original set of Android applications is split into multiple small sets that IccTA can analyze.

6. EVALUATION

Our evaluation addresses the following research questions:

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

RQ2 Can IccTA find leaks in real-world applications and how fast is it?

RQ3 How do IccTA compare to other academic ICC leak detection approaches?

[7]In practice we have not seen a leak going through more than 2 apps.

6.1 RQ1: IccTA vs FlowDroid and Commercial Tool

We evaluate and compare IccTA with FlowDroid and IBM AppScan Source 9.0 on DroidBench to test for ICC and IAC leaks. Unfortunately, we were unable to compare IccTA to other static analysis tools as their authors did not make them available.

DroidBench. DroidBench [2] is a set of hand crafted Android applications for which all leaks are known in advance. The fact of knowing all leaks in the applications is called the *ground truth* and is used to evaluate how well static and dynamic security tools find data leaks. DroidBench version 1.2 contains 64 different test cases with different privacy leaks. However, all the leaks in DroidBench are intra-component privacy leaks. Thus, we developed 26 apps and 23 test cases to extend DroidBench with ICC and IAC leaks. A test case is applied on one application to test for ICC and on two applications to test for IAC. In total, 18 apps contain inter-component privacy leaks and 6 apps contain inter-app privacy leaks. The new set of test cases covers each of the top 8 ICC methods in Table 1. Moreover, among the 26 new apps, two of them do not contain any privacy leaks. If a tool detects privacy leaks on these two apps, the detected leaks are false alarms. Finally, for each test case application we add an unreachable component containing a sink. These unreachable components are used to flag tools that do not properly construct links between components.

The 23 test cases are listed in the first column of Table 2.

IccTA. We run IccTA on all the 23 test cases. The results are shown in Table 2. IccTA successfully passes 18 test cases, with 17 test cases containing 19 privacy leaks and one test case (`startActivity5`) with no leak.

Among the detected privacy leaks, three of them are IAC based privacy leaks and the remaining ones are ICC based privacy leaks. In the `startActivity5` test case, the source component uses an implicit intent with data type *text/plain* to start another activity. However, no other activity in this test case declares that it can receive an intent with data type *text/plain*. That means there is no connection among the components in `startActivity5` test case. As IccTA takes into consideration the data type of an intent it does not report any privacy leak for this test case.

The `startActivity4` test case does not contain any leaks. However, IccTA does report a false warning. The reason is that the source component uses an implicit intent with an URI to start another activity. Since IccTA relies on Epicc

which does over-approximate URIs links, it reports a false leak.

The current version does not take into account `Content Providers`. This is why IccTA misses leaks for the `insert1`, `delete1`, `update1`, and `query1` test cases. All the four test cases are related to `Content Provider`.

FlowDroid. FlowDroid has been evaluated on the first version of DroidBench in [4]. In table 2, we present the results of FlowDroid on the new 23 test cases. As already explained, FlowDroid has been initially proposed to detect leak in single Android component. However, we can use FlowDroid in a way that it computes paths for all individual components and then combines all those paths together (whatever there is a real link or not). As a result, we expect that FlowDroid detects most of the leaks but yields several false positives. Results of Table 2 confirm this expectation: FlowDroid has a high recall (69.6%) and a low precision (23.9%). FlowDroid misses three more leaks than IccTA in `bindService{2,3,4}`. After investigation, we discover that FlowDroid does not consider some callback methods for service components.

AppScan. AppScan Source 9.0 requires a lot of manual initialization work since it has no default sources/sinks configuration file and is unable to analyze Android applications without specifying the entry points of every components. We define the `getDeviceId` and `log` methods, that we always use in DroidBench for ICC and IAC leaks, as source and sink, respectively. We also add all components' entry point methods (such as `onCreate` for activities) as callback methods so AppScan knows where to start the analysis. AppScan is natively unable to detect inter-component dataflows and only detects intra-component flows. AppScan has the same drawbacks as FlowDroid and should have a high recall and low precision on DroidBench. We use an additional script to combine the flows between components. As expected AppScan's recall is high (56.5%) and its precision low (21.0%). Compared to FlowDroid, AppScan does worse. Indeed, AppScan does not correctly handle `startActivityForResult` and thus misses leaks going through methods receiving results from the called activities in `startForResult{2,3,4}`.

Conclusion. IccTA outperforms both the commercial taint-analysis tool AppScan 9.0 and FlowDroid in terms of precision and recall.

6.2 RQ2: IccTA and Real-World Apps

We run the experiments on a Core i7 CPU running a Java VM with 8 Gb of heap. To evaluate our approach, we use IccTA to analyze 3000 Android apps downloaded from the Google Play market as well as some third-party markets (e.g., wandoujia). IccTA process 3000 apps in about 100 hours. IccTA does not detect any leak for 2575 (85.83%) applications. IccTA reports 425 applications containing privacy leaks. Among the 425 apps, 411 apps only contain intra-component leaks and 14 apps contain at least one ICC leak. From those 14 apps, 13 contain both intra-component leaks and ICC leaks. IccTA detects 6989 IAC links. Among those IccTA detects one IAC leak. This result indicates that components do communicate and share data, but it is rare that an inter-application leak occurs.

For intra-app leaks, IccTA detects 5986 leaks in the 425 apps. Among the detected leaks, 147 (2.5%) are ICC privacy leaks. We manually check the 147 reported ICC leaks and found out that 17 (11.6%) are false positives. In other words, IccTA achieves a precision of 88.4% on real-word apps. The false positives comes from Epicc that generates false positives for links between components.

Table 2: DroidBench test results
⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported
† C/A: # of Components / # of Applications

Test Case (C/A)†	FlowDroid	AppScan	IccTA
Inter-Component Communication			
startActivity1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
startActivity2 (4/1)	⊛ (4 ⋆)	⊛ (4 ⋆)	⊛
startActivity3 (6/1)	⊛ (32 ⋆)	⊛ (32 ⋆)	⊛
startActivity4 (3/1)	⋆ ⋆	⋆ ⋆	⋆
startActivity5 (3/1)	⋆ ⋆	⋆ ⋆	
startForResult1 (3/1)	⊛	⊛	⊛
startForResult2 (3/1)	⊛	○	⊛
startForResult3 (3/1)	⊛ ⋆	○	⊛
startForResult4 (3/1)	⊛ ⊛ ⋆	⊛ ○	⊛ ⊛
startService1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
startService2 (3/1)	⊛ ⋆	⊛ ⋆	⊛
bindService1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
bindService2 (3/1)	○	○	⊛
bindService3 (3/1)	○	○	⊛
bindService4 (3/1)	⊛ ⋆ ○	⊛ ⋆ ○	⊛ ⊛
sendBroadcast1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
insert1 (3/1)	○	○	○
delete1 (3/1)	○	○	○
update1 (3/1)	○	○	○
query1 (3/1)	○	○	○
Inter-App Communication			
startActivity1 (4/2)	⊛ ⋆	⊛ ⋆	⊛
startService1 (4/2)	⊛ ⋆	⊛ ⋆	⊛
sendBroadcast1 (4/2)	⊛ ⋆	⊛ ⋆	⊛
Sum, Precision, Recall and F_1			
⊛, higher is better	16	13	19
⋆, lower is better	51	49	1
○, lower is better	7	10	4
Precision ⊛/(⊛ + ⋆)	23.9%	21.0%	95.0%
Recall ⊛/(⊛ + ○)	69.6%	56.5%	82.6%
F_1 2⊛/(2⊛ + ⋆ + ○)	0.36	0.31	0.88

We summarize the frequently used *source* methods and *sink* types (Java classes) in Table 3 from the 425 apps having at least one leak. Note that we only count such *source* and *sink* methods that appear in the detected leaks. The most used *source* method is `openConnection` and it is used 601 times in 169 apps. The most used *sink* types is `Log` and it is used 2755 times in 261 apps. The reason why we study *sink* types instead of *sink* methods is that there are a lot of *sink* methods in a same *sink* type. Take the Log *sink* type as an example, there are eight *sink* methods which log the private data to disk.

Let us describe in details three leaks, one for each type of leak.

Intra-component leak: *bz.prana.myphonelocator.* IccTA detects an intra-component privacy leak starting from the `getLongitude` source method in method `onLocationChanged` of class `.SMSReceiver$MyLocationListener`[8]. The location is sent out of the app through SMS by the `sendTextMessage` *sink* method in method `smsReply` of class `.SMSReceiver`. The app is designed to send the location outside

[8]The package name is omitted when the class name starts with the package name

Table 3: The top 5 used source methods and sink types

Method/Type	Counts(#.)	Detail
Source Methods		
[redacted]	601	http connection
getLongitude	514	longitude
getLastKnownLocation	448	Location
getDeviceId	403	IMEI or ESN
getCountry	265	country code
Sink Types		
[redacted]	2755	error or warn
URL	821	execute
SharedPreferences	717	putInt, putString
Message	339	sendTextMessage
File	9	write(string)

the device through SMS. However, to distinguish the intention of detected privacy leaks is out of scope of this paper. We take it as our further work.

ICC leak: *com.dikkar.ifind.* An ICC based privacy leak is detected by IccTA on this application. In method `onLocationChanged` of class `.iFindPlaces`, the `getLongitude` *source* method is called and returns the location of the Android phone. Then, the location is transferred to another component named `.PlaceDetail`, where method `b` of class `j` is called. In method `b`, a *sink* method `Log.d` logs the location into disk with `ServiceHandler` tag name. To verify the detected leaks, we developed an Android application named LogParser. By giving the permission `android.permission.READ_LOGS` [9], LogParser reports all the locations logged by `Find Places`.

IAC leak: *com.bi.mutabaah.id to jp.benishouga.clipstore.* An IAC leak is reported by IccTA between app `com.bi.mutabaah.id` and app `jp.benishouga.clipstore`. The *source* method `findViewById` is called in component `com.bi.mutabaah.id.activity.Statistic`, where the data of a `TextView` is obtained. Then the data is stored into an intent with two extras named `android.intent.extra.SUBJECT` and `android.intent.extra.TEXT`. After that, `startActivity` is used to send the data to app `jp.benishouga.clipstore`, which extracts the data from the intent with the same extra names and writes all the data into a file named `clip.txt` under path `/data/data/jp.benishouga.clipstore/files`.

Conclusion. IccTA finds leaks in real-world apps in a reasonable amount of time. Nevertheless, IccTA only detects a single IAC leak. This is an indication that inter-application leaks are rare.

6.3 RQ3: Compare with Other academic Tools

We identify two academic tools able to deal with ICC leaks: SCanDroid [13] and SEFA [26]. However, ScanDroid fails to report any leaks and SEFA is not available. As a result, we were not able to evaluate them on DroidBench.

To answer the research question, we focus and discuss some key aspects of the various approaches. SCanDroid and SEFA both use a *path matching* approach, which computes paths for all individual components and then combines some paths together, the decision of combining two paths or not is given by a matching algorithm. A *path matching* approach presents at least two main drawbacks.

First, even if the taint analysis is done for each component, the context of the analysis is lost when SCanDroid and SEFA combine the taint paths, since the analysis is performed before the combination of the paths. IccTA does not present this problem because it connects the components at the code level and then performs the analysis. Thus, it keeps the data-flow between two components. Losing the context decreases the precision of the tool. Indeed, an `Intent` can carry data, i.e., it may contain a lot of extras key/value pairs but only part of them are sensitive. A precise tool needs to distinguish them to avoid false positive. For a path matching approach, it is not easy to distinguish them because they do not keep the state of `Intent` when matching two available paths.

[9]Starting from Android 4.1 it is no more granted to regular apps, but it can still be granted to either vendor apps or apps running on rooted phones.

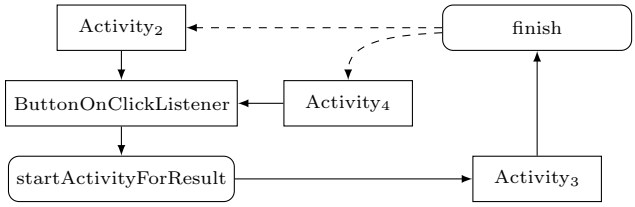

Figure 9: The problem of using path matching approach for `startActivityForResult`

Second, some specific ICC methods such as `startActivityForResult` are difficult to handle with a matching algorithm. It will become even worse when the special ICC methods exist in a class which is invoked by multiple components. Suppose a component `Activity`$_4$ also uses the class `ButtonOnClickListener` shown in Listing 1 to communicate with other components. We present this scenario in Figure 9. A path matching approach first finds a path from `startActivityForResult` to `Activity`$_3$. After the `finish` method of `Activity`$_3$ is called, the `onActivityResult` method of the source component is invoked by the Android system. The problem is that it is difficult to know which component (`Activity`$_2$ or `Activity`$_4$) is the source because they both use the same class `ButtonOnClickListener` where the Intent is created. In fact, It is very difficult to statically resolve this problem since it is caused by the mechanism of dynamic binding of Android (or Java). In our approach, IccTA resolves this problem by explicitly calling the appropriate `onActivityResult` method (see Figures 6 and 7) of the source component (`Activity`$_2$ or `Activity`$_4$) thanks to the helper class `IpcSC`.

Conclusion. Even if we were not able to evaluate state-of-the-art tools detecting ICC leaks (SCanDroid and SEFA), IccTA seems to be more precise mainly because it keeps the context between components unlike *path matching* approaches.

7. LIMITATIONS

In this section, we discuss the limitations of IccTA.

FlowDroid. IccTA is based on FlowDroid to perform static taint analysis and thereby shares the same limitations of FlowDroid. IccTA resolves reflective calls only if their arguments are string constants. It is also oblivious to multi-threading. We experienced that FlowDroid cannot properly analyze some apps (too much memory consumption or hangs). We start by analyzing a set of 5000 and keep only 3000 apps that work with FlowDroid. Running IccTA on a big server could significantly decrease the number of falling

analysis. Moreover, we are very confident that the next release of FlowDroid will resolve this problem.

Epicc. IccTA relies on Epicc to compute links between components. Since Epicc does not handle URIs, it fails to find ICC links for `ContentProvider` and yields false positives for the other three types of components when they communicate using URIs. In practice the number of links is huge due to the false positives. We check the links (intents and intent filters) and only keep the ones not using URIs.

IccTA. At the moment IccTA does not handle some rarely used ICC methods such as `sendActivities` or `sendOrderedBroadcastAsUser`. Data send between component with an intent, is represented as key/value pairs. When a tainted data is put in the intent, IccTA taints all key/value pairs. This could result in false positives if a tainted data is put in an intent and, in the receiving component, a non-tainted data is retrieved from the intent and flows to a sink.

Native Code. Some Android application are packaged with native code. IccTA only analyzes the dex file containing the Dalvik bytecode.

8. RELATED WORK

As far as we know, IccTA is the first approach to seamlessly connect Android components through code instrumentation in order to perform ICC based static taint analysis. By using a code instrumentation technique, the state of the context and data (e.g. an *Intent*) is transferred between components. To the best of our knowledge, there is no other existing static approach to detect Android privacy leaks tackling the ICC problem and keeping state between components.

Static Analyses. There are several approaches using static analysis to detect privacy leaks. PiOS [11] uses program slicing and reachability analysis to detect the possible privacy leaks. TAJ [25] uses the same taint analysis technique to identify privacy leaks in web applications. However, these approaches introduce a lot of false positives. CHEX [18] is a tool to detect component hijacking vulnerabilities in Android applications by tracking taints between sensitive sources and externally accessible interfaces. However, it is limited to at most 1-object-sensitivity which leads to imprecision in practice. LeakMiner and AndroidLeaks state the ability to handle the Android Lifecycle including callback methods, but the two tools are not context-sensitive which precludes the precise analysis of many practical scenarios. FlowDroid [4] introduces a highly precise taint analysis approach with low false positive rate, but it does not identify ICC based privacy leaks. IccTA performs an ICC based static taint analysis by instrumenting the code of the original app while keeping the precision high.

ComDroid [10] and Epicc [19] are two tools that tackle ICC problem, but they mainly focus on ICC vulnerabilities and do not taint data.

SCanDroid [13] is a tool for analyzing ICC based privacy leaks. It prunes all call edges to Android OS methods and conservatively assumes the base object, the parameters and the return value to inherit taints from arguments. This approach is much less precise than our tool since we model all the Android OS methods (except native methods) with our dummy main method in the control-flow graph. Another tool SEFA [26] also resolves the ICC problem. It performs a system-wide data-flow analysis to detect possible vulnerabilities (e.g., passive content leaks). Both SCanDroid and SEFA use a matching approach to analyze inter-component leaks. SCanDroid defines all the methods importing data to an app as *inflow* methods and all the methods exporting data from an app as *outflow* methods. Then, it matches the *inflow* and the *outflow* methods to connect two components. SEFA defines ICC methods as *bridge-sinks* to distinguish with the *sensitive-sinks*. It uses the *bridge-sinks* to match with other components and thereby connecting two components. As we mentioned before, the matching approach has some drawbacks compared to our instrumenting approach. Therefore, even if we were not able to evaluate SCanDroid and SEFA on DroidBench, it comes that IccTA is more precise by design.

AsDroid [15] and AppIntent [28] are another two tools using static analysis to detect privacy leaks in Android apps. Both of them try to analyze the intention of privacy leaks. Analyzing the leaking intention is out of scope of this paper. However, we think it is necessary to distinguish whether a privacy leak is intended or not. We take this as our further work.

Dynamic Analyses. Dynamic taint analyses techniques, on the other hand, track sensitive data at runtime. TaintDroid [12] is one of the most sophisticated dynamic taint tracking systems. It tracks flows of private data of third-party apps. CopperDroid [20] is another dynamic testing tool which observes interactions between the Android components and the Linux system to reconstruct high-level behavior and uses some special stimulation techniques to exercise the app to find malicious activities. Several other systems, including AppFence [14], Aurasium [27], AppGuard [5] and BetterPermission [6] try to mitigate the privacy leak problem by dynamically monitoring the tested apps.

However, those dynamic approaches can be fooled by special designed methods to circumvent security tracking [24]. Thus, dynamic tracking approaches may miss some data leaks and yield an under-approximation. On the other hand, static analysis approaches may yield an over-approximation because all the application's code is analyzed even code that will never be executed at runtime. Both approaches are complementary when analyzing Android applications for data leaks.

9. CONCLUSION

This paper addresses the major challenge of performing data-flow analysis across multiple components or multiple applications. We have presented IccTA[10], an ICC based taint analysis tool able to perform such analysis. In particular, we demonstrate that IccTA can detect ICC based privacy leaks by providing a highly precise control-flow graph through instrumentation of the code of applications. Unlike previous approaches, IccTA enables a data-flow analysis between two components and adequately models the lifecycle and callback methods to detect ICC based privacy leaks. When running IccTA on DroidBench, it reaches a precision of 95.0%. When running IccTA on three thousands applications randomly selected from the Google Play store as well other third-party markets, it detects 130 inter-component based privacy leaks in 12 applications. Other existing privacy detecting tools (e.g., AndroidLeaks) could benefit by implementing our approach to perform ICC and IAC based privacy leaks detection.

[10]Our experimental results and IccTA itself are available at https://sites.google.com/site/icctawebpage.

10. REFERENCES

[1] Android (operating system), Feb. 2014. http://en.wikipedia.org/wiki/Android_(operating_system).

[2] Droidbench–benchmarks, Feb. 2014. http://sseblog.ec-spride.de/tools/droidbench/.

[3] S. Arzt, S. Rasthofer, and E. Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks, 2013.

[4] S. Arzt, S. Rasthofer, 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 *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[5] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard: enforcing user requirements on android apps. In *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 543–548, Berlin, Heidelberg, 2013. Springer-Verlag.

[6] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. Le Traon. Improving privacy on android smartphones through in-vivo bytecode instrumentation. Technical report, May 2012.

[7] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[8] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 3–8, 2012.

[9] P. P. F. Chan, L. C. K. Hui, and S. M. Yiu. DroidChecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, Apr. 2012. ACM.

[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *The Network and Distributed System Security Symposium (NDSS 2011)*, 2011.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[13] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/~ avik/projects/scandroidascaa*, 2009.

[14] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[15] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, May 2014.

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

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

[18] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[19] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

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

[21] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

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

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

[24] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.

[26] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[27] R. Xu, H. Saïdi, and R. Anderson. Aurasium:

practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[28] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[29] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.