IccTA: Detecting Inter-Component Privacy Leaks in Android Apps

Li Li*, Alexandre Bartel†, Tegawendé F. Bissyandé*, Jacques Klein*, Yves Le Traon*, Steven Arzt†,
Siegfried Rasthofer†, Eric Bodden†, Damien Octeau‡§, Patrick McDaniel‡
*SnT, University of Luxembourg, firstName.lastName@uni.lu
†EC SPRIDE, Technische Universität Darmstadt, firstName.lastName@ec-spride.de
‡Department of Computer Science and Engineering, Pennsylvania State University, {octeau,mcdaniel}@cse.psu.edu
§Department of Computer Sciences, University of Wisconsin

***Abstract*—*Shake Them All* is a popular "Wallpaper" application exceeding millions of downloads on Google Play. At installation, this application is given permission to (1) access the Internet (for updating wallpapers) and (2) use the device microphone (to change background following noise changes). With these permissions, the application could silently record user conversations and upload them remotely. To give more confidence about how *Shake Them All* actually processes what it records, it is necessary to build a precise analysis tool that tracks the flow of any sensitive data from its source point to any sink, especially if those are in different components.**

Since Android applications may leak private data carelessly or maliciously, we propose IccTA, a static taint analyzer to detect privacy leaks among components in Android applications. IccTA goes beyond state-of-the-art approaches by supporting inter-component detection. By propagating context information among components, IccTA improves the precision of the analysis. IccTA outperforms existing tools on two benchmarks for ICC-leak detectors: DroidBench and ICC-Bench. Moreover, our approach detects 534 ICC leaks in 108 apps from MalGenome and 2,395 ICC leaks in 337 apps in a set of 15,000 Google Play apps.

I. Introduction

Modern mobile operating systems have enhanced usage experience to allow users to easily install third party software. With the growing momentum of the Android operating system, 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].

The success of the Android OS in its user base as well as in its developer base can partly be attributed to its communication model, named Inter-Component Communication (ICC), which promotes the development of loosely-coupled applications. By dividing applications into components that can exchange data within a single application or even across several applications, Android encourages software reuse, and thus reduces developer burden.

Unfortunately, the ICC model, which provides a message passing mechanism for data exchange among components, can be misused by malicious apps to threaten user privacy. Indeed, researchers have shown that Android apps frequently send users private data outside the device without their prior consent [49]. Those applications are said to leak private data. Recently, researchers have investigated ICC methods as features for vulnerability detection [34], in lieu of permissions and API calls. However, there is still a lack of a comprehensive study on the characteristics of the usage of ICCs by Android malware. Typically, what is the extent of the presence of privacy leaks in Android malware?

To answer such a question, an Android analysis tool has to be developed for tracking privacy leaks. Although, most of the privacy leaks are simple, i.e., easily identifiable as they operate within a single component, there have recently been reports of cross-components privacy leaks [44]. Thus, analyzing components separately is not enough to detect leaks: 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.

In this paper, we use a static taint analysis technique to find privacy leaks, e.g., paths from sensitive data, called sources, to statements sending the data outside the application or device, called sinks. A path may be within a single component or cross multiple components. 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 [33], for example, uses static analysis to detect component hijacking vulnerabilities by tracking taints between sensitive sources and sinks. FlowDroid [7] performs taint analysis within single components of Android applications but with a better precision. Most recently, Amandroid [44] has been proposed to detect ICC-based privacy leaks in Android apps. However, it does not currently tackle `Content Provider`, one of the four Android components. It is also not sensitive to some complicated ICC methods such as `bindService` and `startActivityForResult`.

Thus, we propose IccTA, an Inter-component communication Taint Analysis tool, for a sound and precise detection of ICC links and leaks. Although our approach is generic and can be used for any data-flow analysis, we focus in this paper on using IccTA to detect ICC-based privacy leaks. To verify our approach, we developed 22 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 22 apps cover the top 8 used ICC methods illustrated in Table I.

Besides, we test IccTA on 15,000 real-world apps randomly selected from Google Play market in which we detect 2,395 ICC leaks in 337 apps. We also launch IccTA on the MalGenome set containing 1260 malware, where IccTA reports 108 apps with 534 ICC leaks. By comparing the detecting rate $r = \frac{\#\ of\ detected\ apps}{\#\ of\ tested\ apps}$ of the two data sets, we found that $r_{MalGenome} = 8.6\%$ is much higher than $r_{GooglePlay} = 2.2\%$. Thus, we can conclude that *ICC are significantly used by malware to leak private data*, making ICC a potential feature for malware detection.

The contributions of this paper are as follows:

- We present the findings of an empirical study on the use of ICC in Android malware and benign apps.
- We propose a novel methodology to resolve the ICC problem by directly connecting the discontinuities of Android apps at the code level.
- We developed IccTA, an open-source tool for inter-component taint analysis.
- We provide an improved version of DroidBench with 22 new apps for the assessment of tools which detect ICC-based privacy leaks.
- Finally, we present an assessment of IccTA using i) the DroidBench and ICC-Bench test suites, ii) 15,000 real-world Android applications, iii) 1,260 malware apps from MalGenome.

We make available online our full implementation as an open source project, along with the extended DroidBench apps and the scripts to reproduce our experimental results on

https://sites.google.com/site/icctawebpage/

To better mitigate mobile ICC leaks, we also release the 445 problematic apps (337 from Google Play and 108 from MalGenome) to the research community at the above website.

II. Motivation

To motivate our work, we present an overview of the Android ICC system highlighting the implications of its design and implementation choices in II-A. We further perform an empirical study of how ICCs are used in Android apps, to expose the difference of usage between malware and benign apps (cf. Section II-B). Finally, we give a concrete example to introduce ICC leaks in Section II-C.

A. Android ICC Overview

An Android application is made up of basic units, called components, described in a special file, the *Manifest*, included in the application package. There are four types of components: *Activities* that represent user interfaces and constitute the visible part of Android applications; *Broadcast Receivers* that wait to receive event messages, such as incoming calls or text messages, from other components or the system; *Content Providers* which act as the standard interface to share structured data between applications; and *Services* which execute (compute-intensive) tasks in the background. Android *Service* components are particular, as their processing is hidden to the device user, opening numerous opportunities for malicious actions.

Android provides specific methods, hereinafter referred to as *ICC methods*, for triggering inter-component communications among any combinations of the above components. ICC methods take as parameter a special kind of object, called *Intent*, which specifies the target component(s) for the message.

All ICC methods[1] are called with at least one *Intent* object as an argument. To facilitate the use, by some apps, of existing features provided by other apps, Android allows to target components by specifying in the *Intent* an *action* to handle. Such Intents are known as *implicit Intents*. When an implicit Intent is used, e.g., for Activities, the system searches in the installed applications and presents the user with a list of applications capable of handling the action (e.g., choose a browser to open a url). These Intents may also specify *categories*, a *mimetype* and *data* for the target components. In order to be selected for receiving the implicit Intents, applications containing the target components need to specify an *Intent Filter* in their manifest file, declaring their capabilities to process such Intents.

Android, however, offers the possibility for components to directly interact with each other. One component can thus send an *Intent* to another by naming it explicitly. These are known as *explicit Intents*.

B. ICC Usage in Android Apps

To the best of our knowledge there have been no empirical investigation of the usage of ICC in Android apps. Yet, given the importance of ICC in the Android development model, as well as its potential correlation with malware functioning as introduced above, a thorough study on real-world apps can provide answers to the following important questions:

How often are ICCs used in Android apps? This question will lead to the investigation of the extent to which each of the different ICC methods are used in apps, and what types of components they are targeting.

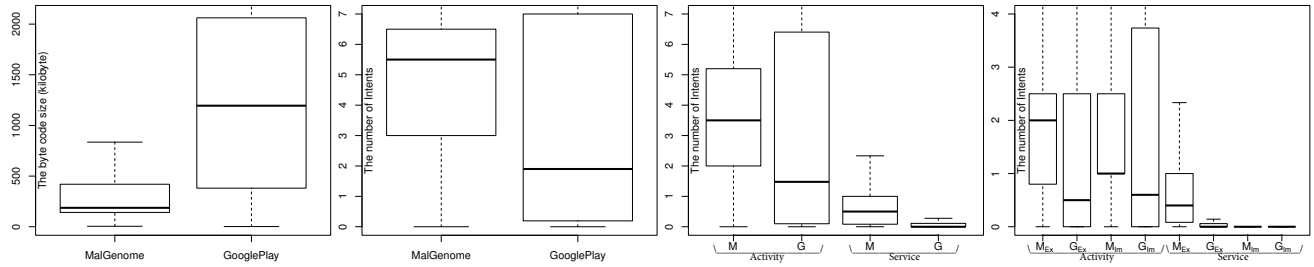
What kind of Intents are used for ICC in apps? This question is important to estimate the differences in the instantiations of *implicit Intents* and *explicit Intents*.

Is the usage of ICC different between malware and benign apps? Investigating this question may open directions for malware detection research, which seeks reliable app features to use in machine-learning processes.

Datasets: We attempt to provide answers to the questions above, using two distinct datasets of over one thousand applications each.

The first dataset, named *MalGenome*, includes $1,023$ Android malware samples collected by Zhou et al. [49]. Although the originally published dataset contains $1,260$ apps, some of these apps share the same package names, therefore we consider them as duplicates.

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



(a) The Bytecode size (b) The Total Intents (c) Activity & Service (d) Explicit & Implicit

Fig. 1: The comparison between MalGenome and GooglePlay apps. "M" means MalGenome, "G" means GooglePlay, "Ex" means explicit and "Im" means implicit. (To highlight the difference between median values, we cut off some upper whiskers).

The second dataset, hereafter referred to as *GooglePlay*, is a set of $1,023$ Android apps[2] randomly selected from the official Google market.

Results: We now present the findings of our investigation.

1) Prevalence of ICCs in app code: First, we compute the usage rate of ICC methods. To that end, we parse the app byte code to count the instances of specific method calls based on a catalog of ICC method names. This analysis was performed on all the $1,023 \times 2$ apps. Table I shows the usage rate of the ICC method names, separating the top 8 most used, from all other ICC methods.

TABLE I: The top 8 used ICC methods. When these methods are overloaded, we consider the one with the most number of calls.

ICC Method	# of Calls	# of Apps
startActivity	54,334 (56.01%)	1,972 (96.4%)
startActivityForResult	11,118 (11.46%)	1,409 (68.7%)
query	8,191 (8.44%)	1,374 (67.2%)
startService	6,660 (6.86%)	1,597 (78.1%)
sendBroadcast	5,119 (5.28%)	1,035 (50.1%)
insert	2,164 (2.23%)	780 (38.1%)
bindService	1,638 (1.69%)	512 (25.0%)
delete	1,633 (1.69%)	472 (23.1%)
Other ICC Methods	6,155 (6.34%)	-
Total	97,012 (100%)	-

The *# of Calls* represents the absolute number of ICC method calls from the entire two sets. The *# of Apps* represents the number of apps using at least once the corresponding ICC method. 96.4% of the apps in our dataset use the `startActivity` ICC method, which accounts for 56.01% of the total ICC methods calls. `startActivity` is used to launch a new Activity component, e.g., to switch from one user interface window to another. The second most used ICC method is `startActivityForResult` which also launches a new Activity component. In this case however, the flow goes back to the calling component. Then follows `query`, an ICC method used to access content providers. `startService`, which appears in 78.1% of the apps, is used to launch a new Service.

2) Number of Intents: Before computing details in the number of Intents[3] in the code of dataset apps, we compare the sizes of apps across the MalGenome and GooglePlay sets. Fig. 1a represents the boxplot[4] of the size of the apps for both sets. The median value for the MalGenome set is 187 KB whereas the median value is 1195 KB for the GooglePlay set. We ensure that this difference of median sizes between the datasets is significantly different by performing a Mann-Whitney-Wilcoxon (MWW). The resulting p-value confirms that the difference is significant at a significance level[5] at 0.001.

To account for any potential bias that the difference of app sizes between datasets may introduce, we proceed to normalize all results according to a unit of dex code size. A normalized result nR is obtained by applying the formula $nR = \frac{iR}{\lceil bS \div 100K \rceil}$, where iR is the initial result and bS is the byte code size.

a) Absolute number of Intents: Fig. 1b represents the boxplot of the normalised numbers of Intents per apps. The median number of Intents is 5.5 per 100KB per app for the MalGenome dataset, and 1.9 for the GooglePlay dataset. The MWW test again shows that this difference is statistically significant.

Malicious applications manipulate significantly more Intents than benign apps

b) Number of Intents vs Component types: We further investigate the difference of number of Intents per app in the two datasets by comparing the usage of Intents for ICC exchange with specific types of components. Fig. 1c shows the boxplot of number of Intents used to launch an *Activity* (left two) and a *Service* (right two). The median values are respectively 3.50 and 0.50 for the MalGenome dataset, and 1.48 and 0.00 for the GooglePlay dataset.

c) Explicit Intents vs Implicit Intents: Table II provides comparison data on the proportion of *implicit* and *explicit* Intents among the overall numbers of *Intents*. In the MalGenome set, 27,278 Intents in total are found, where 14,034 (51.4%) are *explicit Intents*. In the Google Play set, however, only 40.1% (22,955 out of 56,213) of the Intents are *explicit Intents*.

Fig. 1d presents boxplots detailing the normalised number of implicit and explicit Intents for both Activity and Service. The median values between the MalGenome dataset and the

[2]We choose $1,023$ apps to avoid a class imbalance issue.

[3]In this paper, we do not distinguish the difference between Intents and ICC methods since basically an Intent is corresponding to an ICC method.

[4]We use the R tool to draw the boxplot. Each boxplot contains five main horizontal lines. From top to bottom: MAXIMUM (i.e., the greatest value, excluding outliers), UPPER QUARTILE (25% of data points are above this line), MEDIAN, LOWER QUARTILE and MINIMUM.

[5]Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, there is one chance in a thousand that the difference between the datasets is due to a coincidence.

TABLE II: Comparison of the use of Intents in the data sets.

Dataset	Activity (expl./impl.)	Service (expl./impl.)	Receiver (expl./impl.)	Total (expl./impl.)
MalGenome	8803/9569	5204/422	27/3253	14034/13244
Google Play	20018/30461	2715/828	222/1969	22955/ 33258

GooglePlay dataset are close in the case of implicit Intents. On the other hand, there is a larger difference for explicit Intents. We further confirm that this difference is statistically significant via the MWW test, using a significance level of $\alpha = 0.0001$.

Malicious applications tend to use more explicit Intents than benign apps.

This empirical investigation of ICC in malware and benign apps highlights the importance of ICC in the context of Android app security management. In particular, the focus of this study has demonstrated that some Android properties, e.g., possibility to explicitly target a component, thus bypassing user's choice, are exploited by malicious apps. Such apps can indeed *leak* private data across components. After presenting our approach for detecting leaks that are related to ICC, we will perform a final experiment to investigate whether there is a correlation between the number of Intents and the number of ICC leaks. A positive correlation will thus provide confirmation that ICC can be explored as a feature for malware detection.

C. ICC leaks

We define a privacy leak as a path from sensitive data, called *source*, to statements sending this data outside the application or device, called *sink*. A path may be within a single component or across multiple components. In this paper, the *sources* and *sinks* we use are provided by SUSI [38].

```
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     void onClick(View v) {
8      String id = telMnger.getDeviceId();
9      Intent i = new
        Intent(Activity1.this,Activity2.class);
10     i.putExtra("sensitive", id);
11     Activity1.this.startActivity(i);
12 }});}}
13 class Activity2 extends Activity {
14  void onStart() {
15   Intent i = getIntent();
16   String s = i.getStringExtra("sensitive");
17   sms.sendTextMessage(number,null,s,null,null);
18 }}
```

Listing 1: A Running Example.

Listing 1 illustrates the concept of ICC leak through a concrete example. The code snippets present two Activities: $\texttt{Activity}_1$ and $\texttt{Activity}_2$. $\texttt{Activity}_1$ registers an anonymous button listener for the *to2* button (lines 5-11). An ICC method `startActivity` is used by this anonymous listener. When button *to2* is clicked, the `onClick` method is executed and the user interface will change to $\texttt{Activity}_2$. An `Intent` containing the device ID (lines 15), considered as sensitive data, is then exchanged between the two components by first attaching the data to the Intent with the `putExtra` method (lines 10) and then by invoking the ICC method `startActivity` (lines 11). Note that the Intent is created by explicitly specifying the target class ($\texttt{Activity}_2$).

In this example, `sendTextMessage` is systematically executed when $\texttt{Activity}_2$ is loaded since `onStart` is in the execution lifecycle of an `Activity`. The data retrieved from the `Intent` is thus sent as a SMS message to the specified phone number: *there is an ICC leak triggered by button to2*. When *to2* is clicked, the device ID is transferred from $\texttt{Activity}_1$ to $\texttt{Activity}_2$ and then outside the application.

In this paper, we aim to perform static taint analysis for Android apps to detect such inter-component communication (ICC) based privacy leaks. In static taint analysis, a leak k corresponds to a sequence of statements, which starts from a *source* s and ends with a *sink* d. *Sources* are identified as they return private data from the user's point of view into the application code, while *Sinks* are identified as they send data out of the application. An ICC leak is a special leak which contains, in its statement sequence, at least one ICC method. Normally, $C(s) \neq C(d)$, where $C(s)$ means the component of method s. But in some cases, $C(s)$ can equal to $C(d)$. Take ICC method `startActivityForResult` as an example, component C_1 can use this method to start a component C_2 (in method m_1). Once C_2 finishes running, C_1 runs again (in method m_2) with some result data returned from C_2. An ICC leak may occur as $m_1 \rightarrow C_2 \rightarrow m_2$ but in this situation $C(m_1) == C(m_2)$.

III. Our Approach

In this section, we introduce in Section III-A the specificity of Android apps that makes statically analyzing them difficult. Then, we present an overview of our tool called IccTA, which is designed to detect ICC leaks in Section III-B. IccTA uses a two-step approach: 1) ICC links extraction; 2) Taint flow analysis for ICC. Sections III-C and III-D detail these two steps respectively.

A. Static Analysis for Android is Difficult

Despite the fact that Android apps are mainly programmed in Java, off-the-shelf static taint analysis tools for Java do not work on Android applications. Static Analyzers for Android 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 *Intent Filter* and *Intent*. The dynamic resolution done by the Android system to match *Intent Filter* and *Intent* 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. Take Listing 1 as an example, analysis tools need to be able to find the link from ICC method `startActivity` to `Activity2` and to be able to propagate the Intent `i` in line 11 to method `getIntent` in line 15.

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

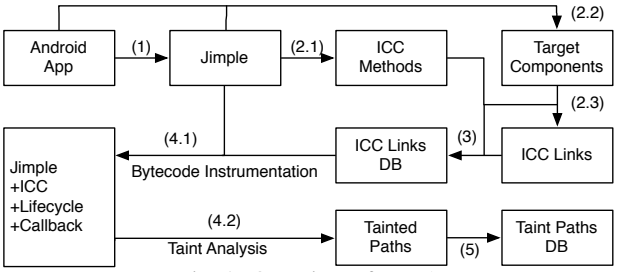


Fig. 2: Overview of IccTA.

mainly done by handling specific callback methods such as the *onClick* (line 7 in Listing 1) method which is called when the user clicks on a button. Static analysis requires a precise model that simulates users' behaviors.

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* or *onResume*. 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.

B. IccTA Overview

Fig. 2 shows the overview of IccTA, our open source tool to detect ICC leaks. Even if Android apps are implemented in Java, an app is compiled into Dalvik bytecode instead of the traditional Java bytecode. In a first step, IccTA uses Dexpler [11] to transform this Dalvik bytecode into $Jimple$, a Soot's internal representation [28]. Soot is a popular framework to analyze Java based apps. In the second step (arrows $2.*$), IccTA extracts the ICC links, and in step 3, stores them as well as all the collected data (e.g., ICC call parameters or Intent Filter values) into a database. Based on the ICC links, in step 4.1, IccTA modifies the Jimple representation to directly connect the components to enable data-flow analysis between components. In step 4.2, by using a modified version of FlowDroid [7], a high precise intra-component taint analysis tool for Android apps, IccTA builds a complete control-flow graph of the whole Android application. This allows propagating the context (e.g., the value of Intents) 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. At last (step 5), IccTA stores the reported tainted paths (leaks) into database.

In both steps (3) and (5), we store all the results including the ICC methods with their attribute values such as URI and Intent, the target components with their Intent Filter values, the built ICC links and the reported ICC leaks into a database. This allows to only analyze an app once, and then reuse the results from the database.

In the next two sections, we detail the main technical contributions of IccTA, which lie in steps 2 and 4.

```
(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
+}
```
Fig. 3: Handling `startActivity` ICC method.

C. ICC links extraction

In this section, we detail our approach to extract the ICC links of the analyzed apps. An ICC link $l : m \rightarrow C$ is used to link two components in which the source component contains an ICC method m that holds information (e.g., the class name for an explicit Intent or the *action, category, mimetype,* ... information for an implicit Intent) to access the target component C.

As shown in Fig. 2, IccTA uses three steps to extract the ICC links from an app. In step (2.1), IccTA leverages Epicc [37] to obtain the ICC methods and their parameters (e.g., *action* of Intents). Epicc is a tool, based on Soot and Heros [13], to identify ICC methods as well as their parameter values (e.g., *action*, *category*). In IccTA, we use IC3 [36], an advanced tool that implements the idea of Epicc, to also parse the `URIs` (e.g., *scheme*, *host*) to support `Content Provider` related ICC methods (e.g., `query`) and to fully support the data field of Intents. In step (2.2), IccTA identifies all the possible target components by parsing the configuration file named `AndroidManifest` of an app to retrieve the values of the *Intent Filters*. In some situations, analyzing the bytecode is also necessary since `Broadcast Receiver` can be registered at runtime. In step (2.3), we match ICC methods with their target components, i.e., the *Intents* with *Intent Filters*, through the rules introduced by the Android documentation [4].

D. Taint Flow Analysis for ICC

In this section, we detail our instrumentation approach to perform taint flow analysis for ICC. As detailed in Section III-A, there are three types of discontinuities in Android: (1) ICC methods, (2) lifecycle methods and (3) callback methods. We first describe how IccTA tackles ICC methods in Section III-D1. Then, we detail how IccTA resolves lifecycle and callback methods in Section III-D2.

1) ICC Methods: In step (4.1) of Fig. 2, the *Jimple* code is instrumented by IccTA to connect components. This code modification is required for all ICC methods (listed in Table I). The main idea of the transformation is to replace an ICC method call with an instantiation of the target component with the appropriate *Intent*. 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. Fig. 3 shows the code transformation done by IccTA for the ICC link between $\texttt{Activity}_1$ and $\texttt{Activity}_2$ of our running example. IccTA first creates a helper class named `IpcSC` (B in Fig. 3)

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), 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 $\texttt{Activity}_2$), IccTA uses the ICC links stored in step (3) in which not only the explicit Intents but also the implicit Intents are resolved. Therefore, there is no difference for IccTA to handle explicit or implicit Intents based ICCs.

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. Fig. 4 shows the control-flow mechanism of `startActivityForResult` ICC method. 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`). Fig. 5 shows how the code is instrumented to handle the `startActivityForResult` method for Fig. 4. To stay consistent with common ICC methods, we do not instrument the `finish` method of $\texttt{C}_2$ to call `onActivityResult` method. Instead, we generate a field `intent_for_ar` to store the *Intent* which will be transferred back to $\texttt{C}_1$. The *Intent* that will be transferred 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 the destination component. Then, it calls the `onActivityResult` method of the source component.

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

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

(B)
```
+class IpcSC {
+ static void redirect0(C1 c1,
+                        Intent i){
+  C2 c2 = new C2(i);
+  c2.dummyMain();
+  Intent retI = c2.getIntentFAR();
+  c1.onActivityResult(retI);
+ }
+}
```

Fig. 5: Handing the `startActivityForResult` ICC method. (A) and (C) represents the modified code of $\texttt{C}_1$ and $\texttt{C}_2$ respectively. (B) is the glue code connecting $\texttt{C}_1$ and $\texttt{C}_2$. Some method parameters are not represented to simplify the code.

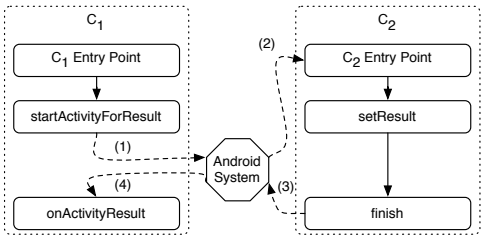


Fig. 4: The control-flow of `startActivityForResult`.

2) Lifecycle and Callback Methods: One challenge when analyzing Android applications is to tackle the callback methods and the lifecycle methods of components. An introduction about lifycycle and callback methods can be found in [30]. There is no direct call among those methods 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 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.

IV. Evaluation

Our evaluation addresses the following research questions:

- RQ1 How does IccTA compare with existing tools?
- RQ2 Can IccTA find ICC leaks in real-world apps?
- RQ3 What is the runtime performance of IccTA?

All the experiments discussed in this section are performed on a Core i7 CPU running a Java VM with 8GB of heap size.

A. RQ1: Comparison With Existing Tools

In this research question, we compare IccTA with four existing tools: FlowDroid [7], IBM AppScan Source 9.0 [3], DidFail [27] and Amandroid [44]. FlowDroid is a state-of-the-art open-source tool for intra-component static taint analysis, AppScan Source is a commercial tool released by IBM, while DidFail and Amandroid are two recent state-of-the-art tools for detecting Android ICC leaks. All the tools are able to directly analyze Android bytecode except AppScan Source, which is only able to analyze the source code of the apps. Unfortunately, we were unable to compare IccTA with other

static taint analysis tools as either they fail to report any leaks (e.g., SCanDroid [21]) or their authors did not make them available (e.g., SEFA [45]).

1) Experimental Setup: We assess the efficacy of all aforementioned tools by running them against about 30 test cases, for ICC leaks, from two benchmarks: DroidBench and ICC-Bench.

DroidBench. DroidBench is a set of hand crafted Android applications for which all leaks are known in advance. These leaks are used as *ground truth* 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 22 test cases to extend DroidBench with ICC leaks. The new set of test cases covers each of the top 8 ICC methods in Table I. Among the 22 new test case applications, we included four (`startActivity{4,5,6,7}`) that do not contain any privacy leaks and thus will help detect false alarm rates of analysis tools. 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.

ICC-Bench. ICC-Bench is another set of apps introduced by Amandroid [44]. It contains 9 test case applications, where one of them uses explicit Intents, 6 of them use implicit Intents and the remaining two use dynamic techniques to register the target component. However, each of the test case applications indeed contain one ICC leak and do not contain any unreachable component as DroidBench does. Because the source code of apps in the ICC-Bench were not available, we could not evaluate AppScan on this benchmark.

2) ICC Data Leak Test: Table III presents the results for comparing how related tools perform in the detection of ICC leaks. All 31 (22 added to DroidBench + 9 from ICC-Bench) test cases, and the corresponding detection outcome for the tools are listed in this table.

FlowDroid. Because FlowDroid has already been evaluated on the first version of DroidBench [7], we present in table III its test results for the newly added 22 test cases which are dedicated to ICC leaks. Although, as mentioned earlier, FlowDroid was initially proposed to detect leaks in single Android components, we can use FlowDroid in a way that it computes paths for all individual components and then combines all these paths together (whether there is a real link or not). Thus, we expect FlowDroid to detect most of the leaks, although with false positives. Results of Table III confirm this, since FlowDroid shows a high recall (70.0%) and a low precision (27.4%). Furthermore, FlowDroid misses three more leaks than IccTA in `bindService{2,3,4}`. After investigation, we discovered that this is due to the fact that FlowDroid does not consider some callback methods for service components.

AppScan Source 9.0. AppScan 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 component. We define the `getDeviceId` and `log` methods, which we always use in DroidBench for ICC 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 data-flows 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 is 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 `startActivityForResult{2,3,4}` test cases.

DidFail. Since DidFail does not handle explicit ICC, it fails to report leaks for test cases that use explicit Intents between components. For implicit ICC, it is able to report all the leaks for test cases `Implicit{1,2,3,4,5,6}` even when those implicit ICCs use advanced features like `mimetype` or `data`. However, DidFail fails on case `startActivity{4,5}` test cases indicate that DidFail is not sensitive on `mimetype` and `data`. Our assumption is that DidFail uses an over-approximation approach to build implicit ICC links. As long as `action` and `category` are matched, an ICC link is constructed. Indeed, `startActivity{4,5}` use `mimetype` or `data`, but do not contain any real ICC link. Because DidFail currently only focuses on `Activity`, it fails to report any leak for the `Service`, `Broadcast Receiver` (dynamically registered or not) and `Content Provider` test cases.

Amandroid. Amandroid is the most recent state-of-the-art tool that is able to detect ICC leaks. Overall, for the 31 test cases, Amandroid reaches a precision of 78.9% (15 true positives, 4 false positives) and a recall of 51.7% (14 missed leaks). Three of the missed leaks and two of the false alarmed leaks are caused by `startActivityForResult`, where Amandroid is not able to combine `setResult` method to `onActivityResult` method. The `startService2` test case uses `IntentService` instead of `Service` which is used by test case `startService1` to implement the service. Amandroid is able to report a leak on `startService1` but fails to report a leak on `startService2`. This indicates that it does not completely model `Service`'s lifecycles. When the callback method changes from `onStartCommand` to `onHandleIntent`, Amandroid is not able to deal with it anymore. Eight other missed leaks indicate that Amandroid currently does not handle the `bindService` method and `Content Provider` components. Amandroid reports two false positives for `startActivity{6,7}`, which indicates that it is not able to distinguish the extra keys of an Intent. Indeed, `startActivity{6,7}` do not contain any

leaks because they use different extra keys for the transferred Intent. Finally, Amandroid misses a leak on test case `DynRegister2` because `DynRegister2` uses string operations (e.g., `StringBuilder` Objects to contact multiple strings) which Amandroid cannot parse.

IccTA. Our tool, IccTA, also misses a leak on case `DynRegister2` like Amandroid, because, currently, it cannot parse complicated string operations as well. The same reason causes IccTA to yield a false positive on case `startActivity7`, where one extra key is built through complicated string operations. The current version of IccTA performs a simple string analysis to distinguish the extra keys of an Intent between one another.

IccTA outperforms both the commercial and academic tools by achieving a precision of 96.6% and a recall of 96.6% on DroidBench and ICC-Bench.

B. RQ2: Experimental Results on Real-World Apps

To evaluate our approach, we launch IccTA on two Android app sets: 1) *MalGenome* which contains 1260 Malware apps and 2) from *GooglePlay*, with 15,000 randomly selected apps.

For *MalGenome*, IccTA reports 108 apps ($r_{MalGenome} = 8.7\%$) containing at least one ICC leak, with a total of 534 leaks. And for *GooglePlay*, IccTA detects 337 apps ($r_{GooglePlay} = 2.2\%$) with 2,395 ICC leaks. Since $r_{MalGenome}$ is significantly higher than $r_{GooglePlay}$, we can conclude that malware indeed use ICC to leak private data. We further studied the correlations between the number of Intents and the number of detected ICC leaks for the two data sets. In this study, only apps that contain ICC leaks are considered. Interestingly, our results show that there is no correlation for *GooglePlay* apps. However, there exists a positive correlation for *MalGenome*. The Spearman's rho for *MalGenome* yielded the value 0.42 (p-value < 0.001), suggesting that the malware do use ICC to leak private data.

In total, IccTA detects 445 (108 + 337) apps from the *MalGenome* and *GooglePlay* sets. We summarize the most frequently used *source* methods and *sink* categories (Java classes) from those apps in Table IV. The most used *source* method is `getLongitude`: it is used 427 times. The most used *sink* category is `SharedPreferences`: it is used 1188 times. The reason why we study *sink* category instead of *sink* methods is that there are a lot of *sink* methods belonging to a same *sink* category (e.g., Log *sink* category includes eight *sink* methods which save private data to disk).

We further studied the $< sourceMethod, sinkCategory >$ pairs of the detected leaks. We found that the most frequently used pair is $< getLongitude, SharedPreferences >$, which happened 208 times. For example, in *MalGenome*, app *com.evilsunflower.farmer* obtains its longitude in class `SetPreferences` and transfers it into component `PushService`, in which the longitude is leaked. It also frequently happened in *GooglePlay* such as in app *infire.beautyleg.sexy.girls* and *ro.an.moneymanagerfree*.

Now, we give one case study to describe the detail of a leak. *com.wanpu.shuijinddp (version 11)* is an app in which an ICC leak has been reported by IccTA. It takes the device id (we consider the device id as sensitive data) as an unique user id to communicate with a remote server[6] via HTTP. It first reads the device id and stores the id to a private field in class `com.waps.AppConnect`. Then, method `showOffers` of class `AppConnect` transfers the device id to component `OffersWebView` in which the device id has been sent to a remote server through a HTTP parameter. In this case, the device id has been leaked to a specified remote server through an ICC. Besides, the device id may be captured by hackers since it only uses HTTP instead of HTTPS to communicate with the remote server.

TABLE IV: The top 5 used source methods and sink categories

Method/Type	Counts(#)	Detail
Source Methods		
getLongitude	427	get longitude
getLatitude	302	get latitude
getDeviceId	289	get IMEI or ESN
getLastKnownLocation	141	get location
getLine1Number	71	get phone no. of line 1
Sink Categories		
SharedPreferences	1188	putInt, putString
HTTP	665	execute
Log	301	error or warn
File	38	write(string)
Message	15	sendTextMessage

Finally, we investigated the total reported leaks and we found that 1812 out of 2929 (61.9%) leaks are leaked through `Service` components. These findings are interesting since using ICC makes leak detection difficult for analysis tools, while using `Services` hides those leaks to the user. Indeed Service components are running in the background with no interaction with the user (contrary to Activity components).

We were able to find ICC leaks in a large set of real-world apps. Correlation studies have further revealed that malware are indeed using ICC to leak private data.

C. RQ3: Runtime Performance

We present the runtime performance analysis of FlowDroid, Amandroid and IccTA in Fig. 6. We randomly selected 50 apps from our $Googleplay$ set for our study. Among those, only 18 apps have been successfully analyzed by all three tools altogether.

First, we compare the performance between FlowDroid and IccTA_1 to check whether our bytecode instrumentation step influences the final performance or not. As shown in Fig. 6, the performance of IccTA_1 is almost as good as FlowDroid. Indeed, an ICC link introduces 50 lines of $Jimple$ code on average, which is negligible comparing to the total code lines (e.g., 412,090 lines for 1 megabyte bytecode on average).

Second and finally, we compare the performance between IccTA_2 and Amandroid. In this case, we take into account the ICC links extraction time for a fair comparison since Amandroid also builds the ICC links. Fig. 6 shows that the median values of IccTA and Amandroid are similar. However, the runtime performance of IccTA_2 presents significantly less variation than Amandroid's, suggesting that Amandroid is highly sensitive to different properties (e.g., size) of the app.

[6]http://app.dwap.com:8000/action/

TABLE III: Test results on DroidBench and ICC-Bench, where multiple circles in one row means multiple leaks expected and an all empty row means no leaks expected as well as no leaks reported. † indicates the tool crashed on that test case. Because FlowDroid and AppScan are not able to directly report ICC leaks, we try our best to manually match their results to report ICC leaks. For the rest tools, we only consider their reported ICC leaks.

⊛ = true positive (correct warning), ⋆ = false positive (false warning), ○ = false negative (missed leak)

Test Case	# of C.	Unreachable C.	Explicit ICC	FlowDroid	AppScan	DidFail	Amandroid	IccTA
DroidBench								
startActivity1	3	T	T	⊛ ⋆	⊛ ⋆	○	⊛	⊛
startActivity2	4	T	T	⊛ (4 ⋆)	⊛ (4 ⋆)	○	⊛	⊛
startActivity3	6	T	T	⊛ (32 ⋆)	⊛ (32 ⋆)	○	⊛	⊛
startActivity4	3	T	F	⋆ ⋆	⋆ ⋆	⋆		
startActivity5	3	T	F	⋆ ⋆	⋆ ⋆	⋆		
startActivity6	3	T	T	⋆ ⋆	⋆ ⋆		⋆	
startActivity7	3	T	T	⋆ ⋆	⋆ ⋆		⋆	⋆
startActivityForResult1	3	T	T	⊛	⊛	○	⊛	⊛
startActivityForResult2	3	T	T	⊛	○	○	○	⊛
startActivityForResult3	3	T	T	⊛ ⋆	○	○	○ ⋆	⊛
startActivityForResult4	3	T	T	⊛ ⊛ ⋆	⊛ ○	○ ○	⊛ ○ ⋆	⊛ ⊛
startService1	3	T	T	⊛ ⋆	⊛ ⋆	○	⊛	⊛
startService2	3	T	T	⊛ ⋆	⊛ ⋆	○	○	⊛
bindService1	3	T	T	⊛ ⋆	⊛ ⋆	○	○	⊛
bindService2	3	T	T	○	○	○ †	○	⊛
bindService3	3	T	T	○	○	○ †	○	⊛
bindService4	3	T	T	⊛ ⋆ ○	⊛ ⋆ ○	○ ○	○ ○	⊛ ⊛
sendBroadcast1	3	T	F	⊛ ⋆	⊛ ⋆	○	⊛	⊛
insert1	3	T	F	○	○	○	○	⊛
delete1	3	T	F	○	○	○	○	⊛
update1	3	T	F	○	○	○	○	⊛
query1	3	T	F	○	○	○	○	⊛
ICC-Bench								
Explicit1	2	F	T	⊛	-	○	⊛	⊛
Implicit1	2	F	F	⊛	-	⊛	⊛	⊛
Implicit2	2	F	F	⊛	-	⊛	⊛	⊛
Implicit3	2	F	F	⊛	-	⊛	⊛	⊛
Implicit4	2	F	F	⊛	-	⊛	⊛	⊛
Implicit5	3	F	F	⊛ ⋆	-	⊛	⊛	⊛
Implicit6	2	F	F	⊛	-	⊛	⊛	⊛
DynRegister1	2	F	F	○	-	○ †	⊛	⊛
DynRegister2	2	F	F	○	-	○ †	○	○
Sum, Precision, Recall and F_1								
⊛ , higher is better	-	-	-	20	10	6	15	28
⋆ , lower is better	-	-	-	53	46	2	4	1
○ , lower is better	-	-	-	9	10	23	14	1
Precision p = ⊛/(⊛+ ⋆)	-	-	-	27.4%	17.9%	75%	78.9%	96.6%
Recall r = ⊛/(⊛+ ○)	-	-	-	70.0%	50.0%	20.7%	51.7%	96.6%
F_1-measure $2pr/(p+r)$	-	-	-	0.39	0.26	0.32	0.63	0.97

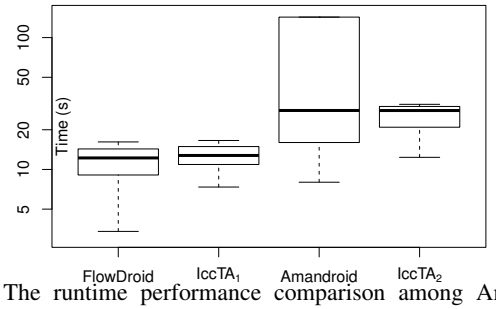


Fig. 6: The runtime performance comparison among Amandroid, FlowDroid and IccTA. $IccTA_1$ does not count the ICC links extraction time while $IccTA_2$ does. All the experiments are performed with the default options.

V. Limitations

At the moment, IccTA resolves reflective calls only if their arguments are string constants. It is also oblivious to multi-threading. For native calls, IccTA carries the limitation of FlowDroid. Currently, IccTA does not handle some rarely used ICC methods such as `startActivities` and `sendOrderedBroadcastAsUser`. IccTA cannot resolve complicated string operations (e.g., by using `StringBuilder`) and the string analysis is within a single method which may cause false alarms. In Android, inter-app communication (IAC) shares the same mechanism as ICC. Thus, our approach is also able to detect IAC leaks (cf. [31]), but in this paper we do not perform experiments on that. We experienced that IccTA cannot properly analyze some apps (too much memory consumption or hangs). Running IccTA on a big server could significantly decrease the failing rate.

VI. 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 [6], the state of the context and data (e.g., an *Intent*) is transferred between components.

Amandroid [44] performs an ICC analysis to detect ICC leaks, and has been developed concurrently with IccTA. Amandroid needs to build an Inter-component Data Flow Graph (ICDFG) and an Data Dependence Graph (DDG) to perform ICC analysis. Since IccTA uses an instrumentation approach,

it does not need to additionally build such assistant graphs. Amandroid provides a general framework to enable analysts to build a customized analysis on Android apps. IccTA provides a *source/sink* configuration to achieve the same function. Amandroid is not able to analyze `Content Provider` as well as some ICC methods such as `bindService` and `startActivityForResult`. Finally, our instrumentation approach is more flexible, and enables generating an app with all components linked at the code level. This app can then be analyzed by any static analysis tool (e.g., Soot or Wala [5]).

DidFail [27] also leverages FlowDroid and Epicc to detect ICC leaks. Currently, it focuses on ICC leaks between `Activities` through implicit Intents. Thus, it will miss leaks involving explicit Intents and components other than `Activities`. Also, it does not handle some parameters for implicit Intents (such as `mimetype` and `data`) and thus generates false links between components. The consequence of that is a higher false positive rate.

SCanDroid [21] and SEFA [45] are another two tools that perform ICC analysis. However, neither of them keeps the context between components and thus are less precise than IccTA by design. ComDroid [14] and Epicc [37] are two tools that tackle the ICC problem, but mainly focus on ICC vulnerabilities and do not taint data. CHEX [33] 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. PCLeaks [29] performs data-flow analysis to detect potential component leaks, which not only includes component hijacking vulnerabilities, but also component launch (or injection) vulnerabilities. ContentScope [50] is another tool that tackles potential component leaks, but it only analyzes `Content Provider` components.

Multiple prior works use static analysis to detect intra-component privacy leaks in Android apps [7], [22], [26], [35], [47]. AndroidLeaks [22] and LeakMiner [47] 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. However, those tools are not able to detect ICC leaks. AsDroid [25] and AppIntent [48] are two other tools using static analysis to detect privacy leaks in Android apps. Both of them try to analyze if a data leak is a feature of the application or not. This kind of analysis is out of the scope of this paper.

Multiple prior works investigated privacy leaks on systems other than Android. PiOS [16] uses program slicing and reachability analysis to detect the possible privacy leaks in iOS apps. TAJ [43] and Andromeda [42] uses the same taint analysis technique to identify privacy leaks in web applications.

Except privacy leaks detection, there has been a rich body of work on other Android security issues [9], [12], [18], [20], [23], [51] such as energy bugs [15], [32] and SSL vulnerabilities [19], [41]. Our work can complement their research by providing a highly precise control-flow graph to enable them to perform inter-component data-flow analysis and consequently to get better results.

Other approaches dynamically track the sensitive data to report security issues. TaintDroid [17] is one of the most sophisticated dynamic taint tracking system. TaintDroid uses a modified Dalvik virtual machine to track flows of private data. CopperDroid [39] is another dynamic testing tool which observes interactions between 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 [24], Aurasium [46], AppGuard [8] and Better-Permission [10] try to mitigate the privacy leak problem by dynamically monitoring the tested apps.

However, those dynamic approaches can be fooled by specifically designed methods to circumvent security tracking [40]. 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. These two approaches are complementary when analyzing Android applications for data leaks.

VII. Conclusion

This paper addresses the major challenge of performing data-flow analysis across multiple components for Android apps. We have presented IccTA, an open source tool, to perform ICC based taint 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 and ICC-Bench, it reaches a precision of 96.6% and a recall of 96.6%. When running IccTA on a set of 1,260 apps of the MalGenome project, it reports 534 ICC leaks in 108 apps (8.6%). When running IccTA on a set of 15,000 real-world apps randomly selected from Google Play market, it detects 2,395 ICC leaks in 337 apps (2.2%). Other existing privacy detecting tools (e.g., AndroidLeaks) could benefit by implementing our approach to perform ICC based privacy leaks detection.

Acknowledgments. This work was supported by a Google Faculty Research Award, by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, by the DFG's Priority Program 1496 Reliably Secure Software Systems and the project RUNSECURE, by the National Science Foundation Grants No. CNS-1064900 and CNS-1228700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Android (operating system). http://en.wikipedia.org/wiki/Android_(operating_system). Accessed: Feb. 2015.

[2] Droidbenchbenchmarks. http://sseblog.ec-spride.de/tools/droidbench/. Accessed: Feb. 2015.

[3] Ibm security appscan source. http://www-03.ibm.com/software/products/en/appscan-source. Accessed: Feb. 2015.

[4] Intents and intent filters. http://developer.android.com/guide/components/intents-filters.html. Accessed: Feb. 2015.

[5] T. j. watson libraries for analysis. http://wala.sourceforge.net. Accessed: Feb. 2015.

[6] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting android and java applications as easy as abc. In *Runtime Verification*, pages 364–381. Springer, 2013.

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

[8] 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.

[9] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: an application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, New York, NY, USA, 2012. ACM.

[10] 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.

[11] 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.

[12] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering (TSE)*, 2014.

[13] 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.

[14] 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.

[15] L. Corral, A. B. Georgiev, A. Sillitti, G. Succi, and T. Vachkov. Analysis of offloading as an approach for energy-aware applications on android os: A case study on image processing. In *Mobile Web Information Systems*, pages 29–40. Springer, 2014.

[16] 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.

[17] 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.

[18] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[19] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[20] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[21] 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.

[22] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] M. Haris, H. Haddadi, and P. Hui. Privacy leakage in mobile computing: Tools, methods, and characteristics. *arXiv preprint arXiv:1410.4978*, 2014.

[24] 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.

[25] 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.

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

[27] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[28] 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.

[29] L. Li, A. Bartel, J. Klein, and Y. Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[30] L. Li, A. Bartel, J. Klein, and Y. Le Traon. Detecting privacy leaks in android apps. *International Symposium on Engineering Secure Software and Systems - Doctoral Symposium (ESSoS-DS2014)*, 2014.

[31] L. Li, A. Bartel, J. Klein, Y. Le Traon, S. Arzt, R. Siegfried, E. Bodden, D. Octeau, and P. Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. Technical Report 978-2-87971-129-4_TR-SNT-2014-9, Apr. 2014.

[32] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[33] 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*, pages 229–240. ACM, 2012.

[34] S. Malek, H. Bagheri, and A. Sadeghi. Automated detection and mitigation of inter-application security vulnerabilities in android (invited talk). In *International Workshop on Software Development Lifecycle for Mobile (DeMobile)*, November 2014.

[35] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[36] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[37] 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.

[38] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *21st Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[39] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the sixth European Workshop on Systems Security (EuroSec)*, 2013.

[40] 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.

[41] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 2014 Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[42] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, S. Guarnieri, et al. Andromeda: Accurate and scalable security analysis of web applications. In *FASE-International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software-2013*, volume 7793, pages 210–225, 2013.

[43] 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.

[44] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM conference on Computer and communications security (CCS 2014)*, 2014.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium, (NDSS)*, 2013.

[51] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium, (NDSS)*, 2012.