Using An Instrumentation based Approach to Detect Inter-Component Leaks in Android Apps

Li Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon
SnT, University of Luxembourg, Luxembourg
{li.li, tegawende.bissyande, jacques.klein, yves.letraon}@uni.lu

I. Introduction

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 and 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. Those applications are said to leak private data [2]. 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. 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.

Thus, we propose IccTA[1] , 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. we test IccTA on 15,000 real-world apps randomly selected from Google Play market in which we detect 337 apps with 2,395 ICC leaks. We also launch IccTA on the MalGenome [5] 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 *malware are using ICC to leak private data more than benign apps*, making ICC a potential feature for malware detection. This paper is an extended abstract version of our research paper [3], where interested readers can find more details of this work.

[1] https://sites.google.com/site/icctawebpage/

II. ICC Problem

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.

```
 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`

(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. 1: Handling `startActivity` ICC method.

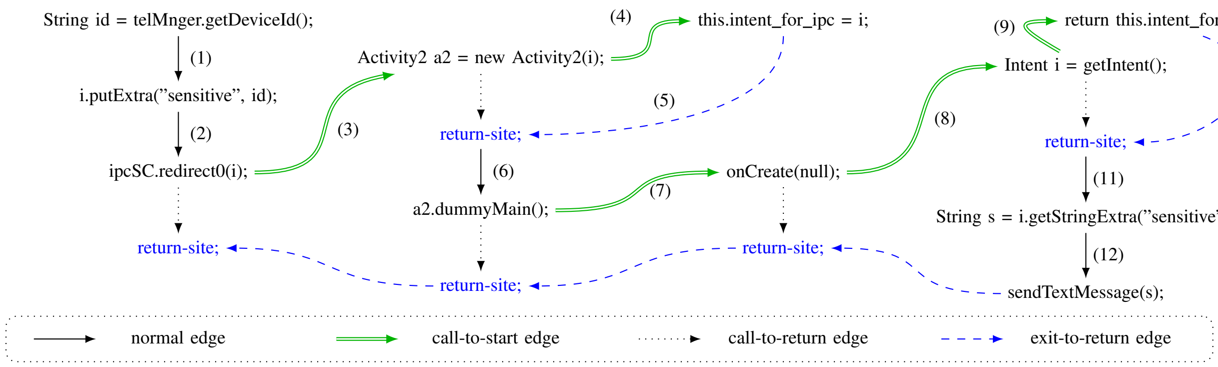

Fig. 2: The control-flow graph of the instrumented running example.

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.

III. Instrumentation based Approach

In this section we briefly introduce our instrumentation based approach, IccTA, which first modifies an Android app's code representation to directly connect components (through ICC links [4]) and then uses a modified version of FlowDroid [1] to build a complete control-flow graph (CFG) of the whole application. This allows propagating the context (e.g., the value of Intents) between Android components and yielding a highly precise data-flow analysis.

Fig. 1 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. 1) 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 extracted by our extended Epicc [4] 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.

Fig. 2 represents the CFG of the instrumented running 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). Fig. 2 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.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *PLDI*. 2014.

[2] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. "Automatically Exploiting Potential Component Leaks in Android Applications". In: *IEEE TrustCom*. 2014.

[3] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps". In: *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. 2015.

[4] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis". In: *USENIX Security*. 2013.

[5] Yajin Zhou and Xuxian Jiang. "Dissecting android malware: Characterization and evolution". In: *IEEE Security and Privacy*. 2012.