Detecting privacy leaks in Android Apps

Li Li, Alexandre Bartel, Jacques Klein, and Yves le Traon

University of Luxembourg - SnT, Luxembourg
{li.li,alexandre.bartel,jacques.klein,yves.letraon}@uni.lu

Abstract. The number of Android apps have grown explosively in recent years and the number of apps leaking private data have also grown. It is necessary to make sure all the apps are not leaking private data before putting them to the app markets and thereby a privacy leaks detection tool is needed. We propose a static taint analysis approach which leverages the control-flow graph (CFG) of apps to detect privacy leaks among Android apps. We tackle three problems related to inter-component communication (ICC), lifecycle of components and callback mechanism making the CFG imprecision. To bridge this gap, we explicitly connect the discontinuities of the CFG to provide a precise CFG. Based on the precise CFG, we aim at providing a taint analysis approach to detect intra-component privacy leaks, inter-component privacy leaks and also inter-app privacy leaks.

Keywords: Static analysis; Taint analysis; Privacy Leaks; ICC; CFG

1 Introduction

Android has become the most popular mobile phone operating system over the last three years. There are hundreds of thousands of applications emerging every day. 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]. Meanwhile, the Android operating system also becomes a worthwhile target for security and privacy attacks. A major problem in Android is private data leaks. A lot of data leaks have been reported this years, such as sending short messages, making phone calls and HTTP connections.

We use a static taint analysis technique based on control-flow graph (CFG) of analyzed apps to detect privacy leaks in Android. Static taint analysis technique is a kind of data flow analysis technique which keeps track of values derived from sensitive data. We first label the private data that we call *source* (for instance a method returning GPS coordinate), and then track the data by statically analyzing the code. If the private data goes to a method which sends it outside the application, also called *sink* method, we identify this as a private data leak and we tag the path from the source to the sink as a detected tainted path. In the CFG, a tainted path means it is reachable from the *source* method to the *sink* method. Thus, privacy leak detection identifies paths between pre-defined *source* and *sink* methods in the CFG of analyzed apps.

[1] http://en.wikipedia.org/wiki/Android_(operating_system)

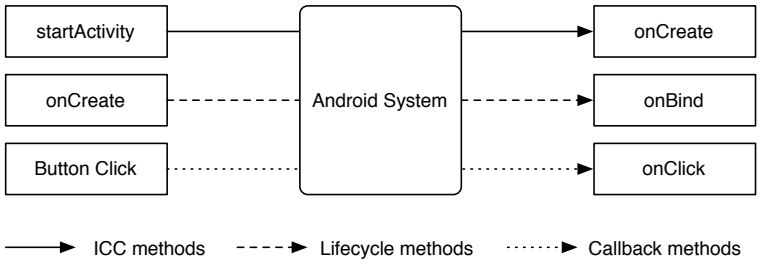

Fig. 1. The three problems causing imprecise CFG of analyzed apps in Android

2 Problems

Since we detect privacy leaks through identifying paths from *source* methods to *sink* methods in the generated CFG of analyzed apps, a precise CFG is essential. However, there are 3 problems that make the generated CFG imprecision. The 3 problems are shown in Fig. 1. The first problem is related to inter-component communication (ICC) methods in Android, we detail it in Section 2.1. The second problem is related to Android's lifecycle methods and the last problem is related to callback methods. We detail them in Section 2.2 and Section 2.3 respectively.

```
1  class Activity1 {
2   void onCreate(Bundle state) {
3    Button btn = (Button) findViewById(to2a);
4    btn.setOnClickListener(new OnClickListener() {
5     public void onClick(View view) {
6      String deviceid = telphonyManager.getDeviceId();
7      Intent intent = new Intent(this, Activity2.class);
8      intent.putExtra("deviceid", deviceid);
9      Activity1.this.startActivity(intent);
10 }}}}
11 class Activity2 {
12  void onResume() {
13   Intent intent = getIntent();
14   String deviceid = intent.getStringExtra("deviceid");
15   HttpClientHelper.send(deviceid);
16 }}
```

Listing 1.1. An example code about crossing component data leaks

2.1 ICC methods

A component is the basic unit to build Android apps. There are four types of components: a) Activity, representing the user interface; b) Service, executing tasks in background; c) Broadcast Receiver, receiving messages from other components or the system; and d) Content Provider, acting as the standard interface to share structured data between applications. Some specific Android system methods are used to trigger component communication. We call them Inter-Component Communication (ICC) methods. The most used ICC method is `startActivity` method which starts a new Activity. Components use Intent to communicate between them. All ICC methods take at least one Intent as their

parameter. Intents can also encapsulate data and thus transfer them between components.

Take Listing 1.1 as an example. `Activity1` and `Activity2` are two components. One ICC method exists in `Activity1` is `startActivity`. `Activity1` contains one *source* method `getDeviceId` which returns the unique device ID (e.g., the IMEI for GSM and the MEID or ESN for CSMA phones) considering that the device id is sensitive data. `Activity2` contains one *sink* method `send` which sends data outside the application. Neither `Activity1` nor `Activity2` contains taint path. But In fact, it does exist one data leak from *source* method `getDeviceId` in `Activity1` to *sink* method `send` in `Activity2`.

Because of the component communication mechanism of Android, we cannot detect crossing component taint paths by tracking tainted data since there is no real code connection between two components but instead only glue code for inter component communication. What we need is to connect components together so we can build a precise CFG and thereby enabling us to track tainted data across multiple components. To achieve this, we want to tackle the following challenges.

Getting Precise ICC links among components Two types of ICC links exist in Android: explicit ICC links and implicit ICC links. Identifying implicit ICC links is more difficult than identifying explicit ICC links because they have complicated matching mechanism for two components. The Android system introduces three conditions (*Action*, *Category* and *Data*) to perform implicit ICC (also IAC). To precisely get all the implicit ICC links, we need to handle all the conditions related to implicit ICC.

Distinguishing Intent Data. Intents are used to transfer data between components. One Intent can contain a lot of data but only part of these data may be tainted. We need to distinguish them to avoid false positive results.

Resolving Special ICC methods. Some ICC methods, which are called special ICC methods, have more complicated semantics comparing with common ICC methods that only trigger one-way communication between components (e.g., `startActivity` method). We need to handle them specifically. With the method `startActivityForResult`, a result may be sent back from an activity when it ends. For example, Component A uses `startActivityForResult` method to start Component B and waits for until B ends. When B ends, Component A retrieves results returned from Component B and runs again.

2.2 Lifecycle methods

In the lifecycle management of the components in Android, there is no main method as in a traditional Java application. Instead, the Android system switches between states of a components lifecycle by calling callback methods such as onStart, onResume or onCreate. The lifecycle of an Activity is shown in Fig. 2. At least six lifecycle methods (e.g., onPause) are involved in an Activity's life

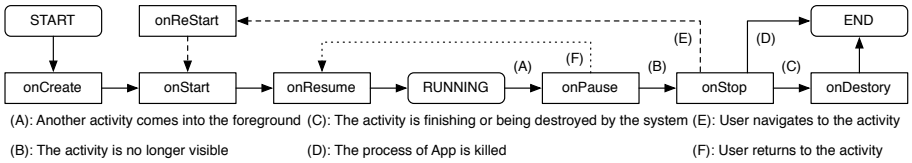

Fig. 2. The lifecycle of an Activity

time. These methods are not directly connected in the app's code, but instead they are executed by the Android system.

2.3 Callback methods

With the user-centric nature of Android apps, a user can interact a lot with the apps (or system) 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. For example, method `onClick` (line 5 in Listing 1.1) is a callback method which will be executed when its related button is clicked. However, there are no code directly connected to the methods in the application.

3 Aims and Goals

Our main goal is to detect private data leaks in Android applications. For this we define the specifications, design and implement a static analysis tool that will detect sensitive taint paths between customized sources and sinks in Android applications. The tool will not only detect intra-component sensitive paths, but also inter-component and inter-application sensitive paths.

The main expected contribution of this research to the field of Engineering Secure Software and Systems is a taint analysis tool that is precise, sound, efficient and that produces less false positive for analysts who work in the field of Android security. The main usage of the tool is to detect privacy leaks. This tool can build call paths between two methods, one being a *source* and the other a *sink*. We define *sources* and *sinks* for our goal to detect private data leaks. However, the tool could be used for other purpose. For example, it can be used to detect that a resource is opened but never closed by defining the open resource method as the *source* and the close resource method as the *sink*.

4 Solution

We plan to statically detect privacy leaks with the CFG of analyzed apps. But three problems detailed in Section 2 exist in Android system which make the CFG imprecise. We cannot rely on an imprecise CFG to detect privacy leaks. Thus, our approach first builds a precise CFG by connecting all the isolate CFGs

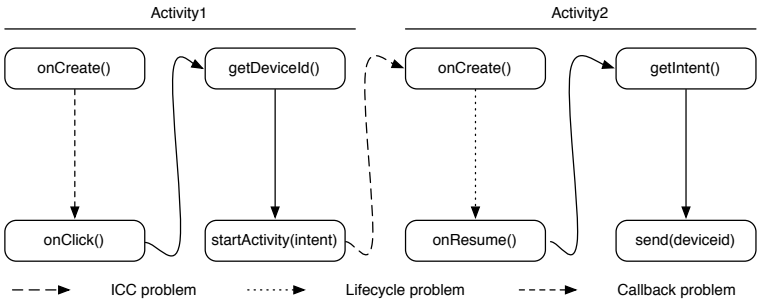

Fig. 3. The precise CFG of the example illustrated in Listing 1.1

of Android apps. Then, based on the precise CFG, we check whether a *source* method can reach a *sink* method or not. If a *sink* method is reached from a *source* method, it means that a privacy leak is detected. Since the precise CFG also models the inter-component communication, we can detect ICC based as well as IAC based privacy leaks .

The precise CFG of the example illustrated in Listing 1.1 is shown in Fig. 3. In the CFG, we connect `startActivity` and `onCreate` methods to resolve ICC problem. We connect `onCreate` and `onResume` methods to resolve Lifecycle Problem and we connect `onCreate` and `onClick` methods to resolve Callback problem. Based on the accurate CFG, we can detect an ICC based privacy leak from `getDeviceId` in `Activity1` to `send` in `Activity2`.

Current Release. To detect privacy leaks in Android apps, we have realized a prototype tool based on our previous work[2]: Epicc [1] which generates links between components of Android applications and Flowdroid [2] which performs intra-component taint analysis. Both Epicc and Flowdroid use the Soot framework [3] which uses the Dexpler plugin [4] to convert Android Dalvik byte code to Soot's internal representation called Jimple. The current version has basically solved the three problems detailed in Section 2. But in some special case (e.g., distinguish the data in an `Intent`), still need to be advanced.

Evaluation Plan. We aim at analyzing private data leaks in either third-party apps or preloaded apps. We intend to test intra-component, inter-component and inter-app based privacy leaks. We plan to test our approach against sample apps written by ourselves since we know the expected output of the apps so that we can compare the precision of our tool with the other existing tools. Then, we look for privacy leaks in the real word applications.

5 Related Work

Android privacy leaks have recently attracted lots of attentions. AppIntent [5] analyzes user-intended sensitive data transmission in Android. Woodpecker [6]

[2] With our colleagues of TU Darmstadt and Penn State University

studies capability leaks which analyze the reachability of a dangerous permission from a public, unguarded interface. Yajin et al. [7] report passive content leaks which cause affected applications to passively disclose in-application data. Our approach is different from them that we provide a generic taint analysis tool which can detects all the above leaks with low false alarms. CHEX [8] uses taint analysis technique to detect component hijacking vulnerabilities in Android Applications. However, it does not analyze calls into the Android framework itself.

We aim at providing a static taint analysis tool with more precise and sound results comparing with the existing tools. To achieve this we rely on FlowDroid [2] a highly precise taint analysis tool for Android and Epicc [1] a highly effective ICC mapping (from ICC method to destination component) tool in Android.

6 Conclusion

We have described the problems of detecting privacy leaks in Android apps. We have also described the solutions of resolving the above problems. Our motivation is to build an precise CFG and thereby to detect ICC based as well as IAC based privacy leaks in Android apps.

References

[1] Damien Octeau et al. “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.

[2] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th Conference on Programming Language Design and Implementation*. 2014.

[3] Patrick Lam et al. “The Soot framework for Java program analysis: a retrospective”. In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. 2011.

[4] Alexandre Bartel et al. “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*. Beijing, China, 2012.

[5] Zhemin Yang et al. “Appintent: Analyzing sensitive data transmission in android for privacy leakage detection”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1043–1054.

[6] Michael Grace et al. “Systematic detection of capability leaks in stock Android smartphones”. In: *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*. 2012.

[7] Yajin Zhou and Xuxian Jiang. “Detecting passive content leaks and pollution in android applications”. In: *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*. 2013.

[8] Long Lu et al. “Chex: statically vetting android apps for component hijacking vulnerabilities”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 229–240.