2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications

Automatically Exploiting Potential Component Leaks in Android Applications

Li Li, Alexandre Bartel, Jacques Klein, Yves le Traon
University of Luxembourg - SnT, Luxembourg
{li.li, alexandre.bartel, jacques.klein, yves.letraon}@uni.lu

***Abstract*—We present PCLeaks, a tool based on inter-component communication (ICC) vulnerabilities to perform data-flow analysis on Android applications to find potential component leaks that could potentially be exploited by other components. To evaluate our approach, we run PCLeaks on 2000 apps randomly selected from the Google Play store. PCLeaks reports 986 potential component leaks in 185 apps. For each leak reported by PCLeaks, PCLeaksValidator automatically generates an Android app which tries to exploit the leak. By manually running a subset of the generated apps, we find that 75% of the reported leaks are exploitable leaks.**

I. INTRODUCTION

The number of Android apps has increased exponentially in recent years. As of May 2012, Android became the most popular mobile operating system, running on the largest set of activated devices, and being the market leader in most countries [1]. Currently, more than 1.17 millions of apps exist in Google play and more than 80% of the apps are free of charge [2]. Not surprisingly, Android phone users are increasingly relying on the apps to manage their personal data. Because of that, the number of malware is also increasing. Kaspersky [3] has reported in its 2013 security bulletin that there are more than 148,427 mobile malware variants in 777 families and that 98.05% of the found mobile malware target the Android platform. As reported by Securelist [4], nearly half of the found Android malware are Trojan (e.g., SMS-Trojan) that steal personal data stored on the user's smartphone.

Some families of malware focus on private data leaks. The more private data they want to leak, the more permissions they have to declare. An application asking for numerous permissions or for permissions the application should not require may alert the user [5, 6]. For instance, a note book application asking for permission to send SMS looks suspicious. Instead of drawing the attention of the user by asking for permissions, malware may exploit vulnerabilities existing in other apps to leak sensitive data. Thus, it is essential to detect those vulnerable apps and thereby keep them from entering the app stores.

State-of-the-art approaches are focusing on either exploiting ICC vulnerabilities or detecting full private data leaks. For example, Epicc [7] is designed to detect ICC vulnerabilities (e.g., Activity Hijacking). But it does not perform data-flow analysis based on the detected ICC vulnerabilities. In other words, Epicc only knows where component may leak something, but it does not know if any data is flowing through the leak which yields many false positives. For private data leaks detection, AndroidLeaks [8], for example, uses static analysis technique to automatically find sensitive data leaks in Android apps on a massive scale. Another tool named IccTA [9], which performs inter-component (and also inter-app) communication based taint analysis to detect privacy leaks in Android apps. However, those tools are mainly focusing on private data leaks. To sum up, none of these tools tackle potential component leaks.

The only existing tool analyzing potential component leaks is ContentScope [10]. However, it only focuses on Content Providers, one of the four component types of an Android application. In this paper we present PCLeaks which finds potential component leaks on the other three components: Activity, Service and Broadcast Receiver. Note that when we talk about potential component leaks, each leak is always within a single component. It is not necessary to exploit inter-component potential leaks since such leaks are covered by intra-component potential leaks. Thus, we only perform intra-component data-flow analysis in this paper.

PCLeaks uses a static taint analysis technique to detect potential component leaks. A "traditional" leak starts with a *source*, a statement retrieving sensitive data from the system, and ends with a *sink*, a statement sending data outside of the application. In this paper, we focus on two types of "potential" leaks. The first type, Potential Passive Component Leak (PPCL), starts at an Android component *entry-point* and ends at a *sink*. For this leak, the component passively leaks data that it receives from other components. The second type, Potential Active Component Leak (PACL), starts at a *source* and ends at a component *exit-point*. For this leak, the component actively sends sensitive data to other components, which may leak the sensitive data intentionally or carelessly.

An example of the two types of component leaks is shown in Figure 1. The single component contains two *sources*, two *sinks*, two *entry-points* and two *exit-points*. There are four data-flow paths, marked as (A), (B), (C) and (D). Path (A) represents a private data leak. Those kind of leaks are well studied by tools such as Flowdroid [11], ScanDal [12] or DroidChecker [13]. Path (D) transfers data from an *entry-point* to an *exit-point*, it does not contain any real *source* or *sink*. Path (B) is a potential passive component leak (PPCL), starting at an *entry-point* and ending at a *sink*. Path (C) represents a potential active component leak (PACL), starting at a *source* and ending at an *exit-point*. In this paper we focus on detecting paths similar to (B) and (C), that is, potential component leaks. Roughly speaking, the fact that a leak of a component is exploited or not is dependent on another component.

Our approach relies on the Control Flow Graph (CFG) of the analyzed apps. If a *sink* node is reached from a *entry-point* node, a PPCL is detected. If a *exit-point* node is reached

978-1-4799-6513-7/14 $31.00 © 2014 IEEE
DOI 10.1109/TrustCom.2014.50

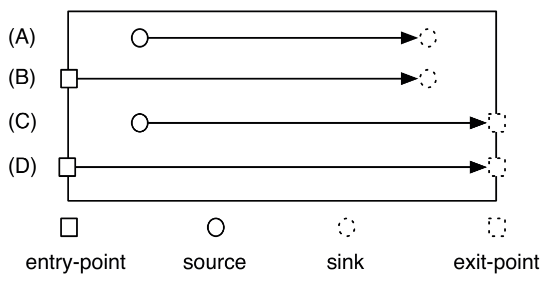


Fig. 1: An example of Potential Component leaks, where (B) is a potential passive component leak (PPCL) and (C) is a potential active component leak (PACL).

from a real *source* node, a PACL is detected. The precision of our approach depends on the precision of the generated CFG. The event-driven nature of the Android system causes discontinuities in the CFG that PCLeaks models by generating a dummy main method for those discontinuities [14]. Then, it performs a data-flow analysis on the precise CFG and finally outputs the detected potential component leaks. We also present a tool called PCLeaksValidator to automatically generate apps to validate leaks reported by PCLeaks. The purpose of the generated apps is to check whether leaks are true positives (e.g., really send sensitive data outside of the app) or not.

The contribution of this paper are as follows:

- PCLeaks, a static taint analysis tool to detect potential component leaks (PACLs and PPCLs).
- PCLeaksValidator, a tool to automatically generate applications to validate leaks reported by PCLeaks.
- An empirical experiment to evaluate PCLeaks and PCLeaksValidator over 2000 real-world Android applications.

The paper continues as follows. Section II explains the necessary background on Android security. Section III gives a motivating example and Section IV introduces the details of potential component leaks. In Section V, the paper discusses the implementation details of our approach. Section VI evaluates our approach. Limitations are discussed in Section VII. Section VIII presents the related work and Section IX concludes the paper.

II. Background

A. Android Components

Components are the essential building blocks of an Android apps. As most components can be shared among applications, they act as entry points to the application. Four different types of components exist in Android. The fist one is `Activity`, which represents a screen with a user interface. The second one is `Service`, which is used to run long-time jobs in the background of the app. The third one is `Content Provider`, which provides a standard interface for other components to manage a shared set of data. The last one is `Broadcast Receiver`, which responds to system-wide broadcast announcements. Of these four types of components, only `Activity` provides a user interface.

An abstract object called `Intent` is used to communicate between two components. It describes an action to be performed (e.g., launching an `Activity`) and the data (extras) transferred by the action. There are two kinds of intents in Android: explicit intents and implicit intents. Explicit `Intents`, specify the target component. Implicit `Intents`, do not specify the target component, but instead, they hold enough other information (e.g., action, category and data) for the system to determine an available component to run.

An app must declare all its components in a configuration file named `AndroidManifest.xml` [1]. Implicit `Intents` can only reach components that declare one or more intent filters. A component not declaring any intent filters can still receive explicit `Intents` which normally come from the same app. However, it is still possible to receive explicit `Intents` coming from other apps. The only limitation is that those apps need to be signed by same signature. Intent filters are used to declare the capabilities of components (e.g., what types of broadcasts a receiver can handle). An example about declaring a component with its intent filter is shown in Listing 1. A service called `SendSMSService` is declared by element *service*. An intent filter is declared by element *intent-filter*. Under *intent-filter*, an action, a category and a data are declared by element *action*, *category* and *data* respectively. `SendSMSService` can receive [2] all the implicit `Intents` which hold the same values of action, category and data as the ones declared by the intent-filter (e.g., action equals to ”action.SEND_SMS”, category equals to ”category.SEND_SMS” and mime type equals to ”text/plain”).

```
<service android:name="SendSMSService">
 <intent-filter>
  <action android:name="action.SEND_SMS" />
  <category android:name="category.SEND_SMS" />
  <data android:mimeType="text/plain"/>
 </intent-filter>
</service>
```

Listing 1: An example about declaring a component with its intent filter.

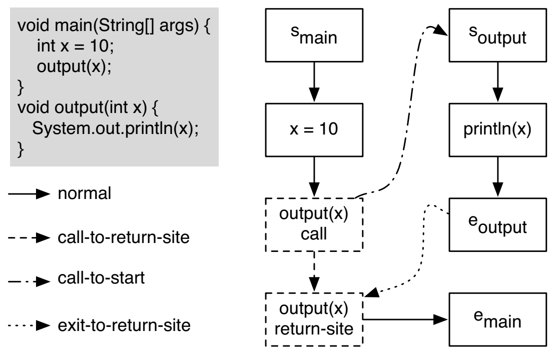
B. Android Event-Driven Nature

Android apps are written in Java and thereby share the event-driven nature of Java. The event-driven nature introduces disconnections between parts of the code. In particular, the callback mechanism introduced in Java is used to implement the event-driven nature. For a concrete example, taking into account the `java.lang.Thread` class. It is used to implement native thread and execute long-time jobs. A developer can extend this class and override the `run` method, and then call the `start` method to send a launching thread event to the system. Then, the system will select an appropriate time to launch the thread by executing the `run` method. There is no code connection between `start` and `run`. When performing a static analysis this has to be modeled.

Similarly, the Android system introduces specific callback methods, called lifecycle methods. Each Android component

[1]Note that for `Broadcast Receivers`, it is also available to programmatically register them.

[2]For more information about intent resolution refer to the official documentation available at http://developer.android.com/guide/components/intents-filters.html#Resolution



```
void main(String[] args) {
   int x = 10;
   output(x);
}
void output(int x) {
  System.out.println(x);
}
```

Fig. 2: An example about CFG and its codes.

has its own internal state. The Android system switches between states of a component by calling specific lifecycle methods of the component. Lifecycle methods are executed by the Android system according to user or system events. For example, when a user navigates back to an existing activity, the `onRestart` method is called. The problem is that there is no direct code connection between lifecycle methods. Thus, it is essential to model the Android's event-driven nature to precisely analyze Android apps.

C. Control-Flow Graph (CFG) and Taint Analysis for Android

To simplify our analysis process and to better describe our approach, we use CFG which are introduced by Reps et al. [15] to intermediately and visually represent the relationships of the app codes. The CFG is made up of a collection of intra-precedure control-flow graph (IPCFG) and the IPCFGs are connected through the call relations in the CFG. In an IPCFG, $\mathtt{s}_{name}$ and $\mathtt{e}_{name}$ are used to specify the start node and the end node respectively. For a procedure call, two nodes (*call* and *return-site*) and three edges (*call-to-return-site*, *call-to-start* and *exit-to-return-site*) are used to represent them.

An example about CFG and its codes is shown in Figure 2. For procedure `output(x)`, two nodes (`call` and `return-site`) are used to represent it. Three edges are also involved for procedure `output(x)`. The first edge is *call-to-return-site* from node `call` to node `return-site` in procedure `output(x)`. The second edge is *call-to-start* form node `call` to node $\mathtt{s}_{output}$ and the last edge is *exit-to-return-site* from node $\mathtt{e}_{output}$ to node `return-site`.

In recent years, many tools to generate call-graphs for Android applications and perform taint-analysis have been developed by researchers such as CHEX [16], TrustDroid [17] or LeakMiner [18]. However, few of them are available online. In this paper we use FlowDroid [11] [3], a highly precise tool for data-flow analysis on Android applications. FlowDroid models the lifecycle of Android components and performs a context, flow, field and object-sensitive taint analysis.

III. Motivating Example

We start by giving a motivating example shown in Listing 2 and Listing 3. Two Android apps, namely `Application1` and `Application2`, are introduced. `Application1`, an example of PPCL, contains a Service named `SendSMSService` in which a content of short message is obtained from the received `Intent` when the Service is launched. Then, the content of the message is sent outside of the app by SMS. `Application2`, an example of PACL, contains an Activity named `GetDeviceIdActivity` in which a device id is obtained when the `Button` *btn* is clicked. Then, the device id is stored in an `Intent` and is sent to other components by method `startService()`.

```
1 //Application1
2 class SendSMSService extends Service{
3  int onStartCommand(Intent i,int f,int id){
4   String sms = i.getStringExtra("sms-content");
5   SmsManager sm = SmsManager.getDefault();
6   sm.sendTextMessage(num,null,sms,null,null);
7   return super.onStartCommand(i, f, id);
8 }}
```

Listing 2: A motivating example about potential passive component leak.

```
 1 //Application2
 2 class GetDeviceIdActivity extends Activity{
 3  void onStart(Bundle state){
 4   Button btn = new Button();
 5   btn.setOnClickListener(new OnClickListener(){
 6    void onClick(View v){
 7     TelephonyManager tm = default;
 8     String id = tm.getDeviceId();
 9     Intent i = new Intent();
10     i.setAction("ACTION_SENDTO"); // send email
11     i.setType("text/plain");
12     i.putExtra("mail-body", id);
13     GetDeviceIdActivity.this.startService(i);
14 }}}}
```

Listing 3: A motivating example about potential active component leak.

In Listing 2, `getStringExtra()` (line 4) is a *entry-point* since it retrieves data from an `Intent` sending from other components. `sendTextMessage()` (line 6) is a *sink* since it sends data outside of the app. From `getStringExtra()` to `sendTextMessage()` (line 4-6), `SendSMSService` will passively receive data and send them outside of the app by short message. We call this behavior as potential passive component leak (PPCL). Malicious app may use this leak to send sensitive data outside of the device. Since the malicious app itself does not contain any *sink* (or unnecessary permissions), it will bypass the private data detection tool like FlowDroid and consequently enter the Android apps market unnoticed.

Listing 3, `getDeviceId()` (line 8) is a *source* since it obtains the unique device id (e.g., the IMEI for GSM and the MEID or ESN for CSMA phones) from the system. `startService()` (line 13) is a *exit-point* since it send data stored in an `Intent` to other components. There is a data-flow path from `getDeviceId()` to `startService()` in `Application2` (line 8-13). `GetDeviceIdActivity` actively leaks the device id to other components. We call this behavior as potential active component leak (PACL).

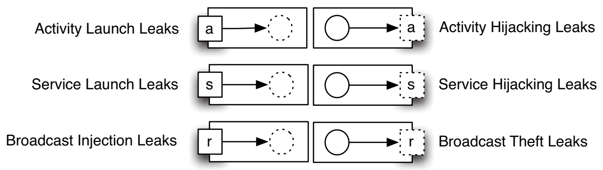
This motivating example illustrates that a few lines of code are enough to create potential leaks that can be exploited by

[3]FlowDroid is open-source and can be downloaded at https://github.com/secure-software-engineering/soot-infoflow-android

malicious applications. The next Section describes the different kinds of potential leaks that we detect in this paper.

IV. Potential Component Leaks

In this section, we detail the classification of potential component leaks. As already said, Potential Passive Component Leak (PPCL) starts at an Android component *entry-point* and ends at a *sink*. Potential Active Component Leak (PACL), starts at a *source* and ends at a component *exit-point*. Moreover, this paper focuses on three types of Android components: Activity, Service, and Broadcast Receiver. As a result, since PPCL and PACL can occur in each of these component types, we define six (3×2) different kinds of potential leaks. Figure 3 illustrates these six kinds of potential leaks.



(a) PPCLs (b) PACLs

Fig. 3: Classification of potential component leaks.

A. PPCL

Figure 3a shows the three kinds of PPCL, that can be triggered by Intent spoofing: Activity Launch Leak, Service Launch Leak and Broadcast Injection Leak.

Activity Launch Leak. Exported Activities can be launched by other components (or applications) with either explicit or implicit Intents. In some case, an Activity may be launched with an Intent and then leaks the Intent's data outside the activity or application. This can be used by malicious apps to passively leak sensitive data. We call this specific leak Activity launch leak.

Service Launch Leak. As for an exported activity, exported services can also be launched by other components or applications. If the service leaks the received Intent's data outside the service or application, the leak is called a Service launch leak. For example, `Application1` in Listing 2 contains a Service launch leak.

Broadcast Injection Leak. A Broadcast Receiver may leak the data it receives from other components or applications. A malicious app can use this to make the Broadcast Receiver passively leak sensitive data. We call this specific leak a Broadcast injection leak.

B. PACL

Figure 3b shows the three kinds of PACL, that may be triggered by Intent hijacking: Activity Hijacking Leak, Service Hijacking Leak and Broadcast Theft Leak.

Activity Hijacking Leak. A malicious Activity can be launched through an Intent hijacking. If the original component reads sensitive data and stores them into an Intent (e.g., extras). The malicious Activity may hijacking the Intent and thereby manipulates the sensitive data. Therefore, when sensitive data is obtained and is sent to other Activities through inter-component communication (e.g., `startActivity`), we call it Activity hijacking leak.

Service Hijacking Leak. A malicious Service may hijacking an Intent, which contains sensitive data in its Extras. In this situation, we call the original component contains Service Hijacking leak. For example, a Service hijacking leak exists in `Application2` of Listing 3.

Broadcast Theft Leak. We call a component that contains Broadcast Theft Leak as it reads sensitive data and sends them through an Intent to a Broadcast Receiver. Because the Intent can be stolen by a malicious Broadcast Receiver.

To summaries, We define six kind of potential leaks (two categories: PPCL and PACL) in this paper. The reason why we distinguish the different component types in each category is that different semantics are performed by Android system when multiple receivable components exist. In detail, for multiple launchable Activities, the system will pop up a selection box to let user decide which Activity is going to be launched. The Android system will randomly select a Service to launch for multiple launchable Services. The Android system launches all available `Broadcast Receivers` for multiple launchable `Broadcast Receivers` [4].

V. Implementation

In this section we discuss the implementation of our approach to find and validate potential component leaks. Our approach strongly relies on the FlowDroid tool and features four steps as illustrated Figure 4. The first three steps describe PCLeaks. In Step1 (Section V-A), PCLeaks extracts the list of reachable Android components. In Step2 (Section V-B), PCLeaks builds a precise CFG with the information provided by Step1. In Step3 (Section V-C), PCLeaks uses the *source* and *sink* methods collection computed by SuSi [19] to perform taint analysis on the precise CFG provided by Step2 and then reports a list of potential leaks it found. In the last step (Section V-D), we use PCLeaksValidator to automatically generate applications to validate leaks reported by PCLeaks. Finally, for each leak reported by PCLeaks we manually run the app, which is generated by PCLeaksValidator to check whether PCLeaks reported a real leak or not.

A. Step 1: Preprocessing

In the first step (Step 1: apktool in Figure 4), PCLeaks extracts the Android XML file using *apktool*[5]. Based on the generated XML file, PCLeaks extracts the following artifacts:
(1) The list of declared components;
(2) The permission attribute of components.
(3) The exported attribute of components.

Computing (1) gives the list of components of an application. If a component is *exported* it means it can be reachable from other applications. If it is not exported, it cannot receive Intents from other apps. In other words, PCLeaks does not

[4]This is not always the case for ordered broadcasts, where the available `Broadcast Receivers` are executed one by one. As each receiver executes in turn, it can abort the broadcast so that it won't be passed to other receivers.

[5]https://code.google.com/p/android-apktool/

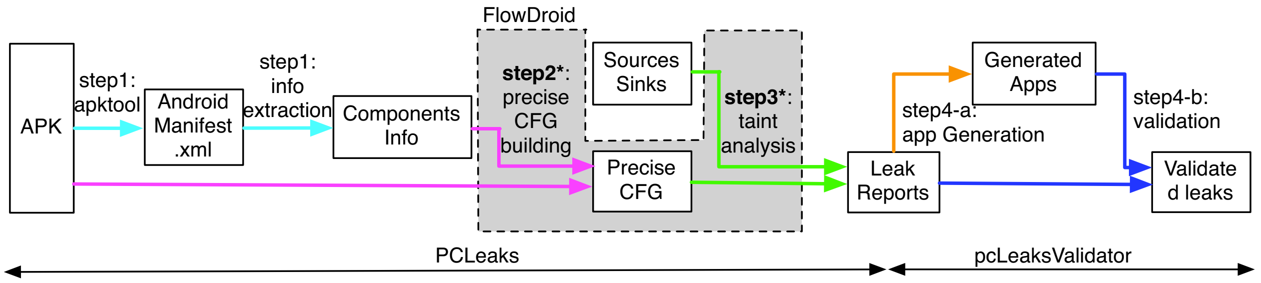


Fig. 4: The processes of our approach (Step1-3 for PCLeaks, Step4 for PCLeaksValidator). * means that we do some improvements for FlowDroid in that step. For example, we feature FlowDroid to generate a precise CFG through the components list in step2 and we leverage FlowDroid to better identify *sink* methods in step3.

analyze non-exported component. We use the algorithm listed in Alogrithm 1 to check whether a component is exported or not. First, we check whether the attribute *exported* is explicitly set in the manifest or not. If it is set explicitly, we directly return the value of the attribute (*true* or *false*). If it is not set explicitly, we need to analyze the default value of the component which is related to the component's type. If the component's type is `ContentProvider` and its app's version is less than or equal to 16, then the component is exported by default. If the component's type is not `ContentProvider` and it contains an *intent-filter* element, then the component is exported. Otherwise, the component is not exported.

But are all those exported components always accessible from another application? Only components not protected by a permission are reachable. Computing (2)[6] gives the list of permission protected components. If a component is protected by a permission, an app trying to communicate with it must declare the same permission. Note that this only works for permissions with protection level *normal* or *dangerous*. Since four protection levels (*normal*, *dangerous*, *signature* and *signatureOrSystem*) exist for permissions in Android, if a component is protected by permissions at *signature* or *signatureOrSystem* level, it is impossible for a malicious app to access the components because the malicious apps would need to be signed with the same signature of they accessed app [7].

We performed a short study of permission protected components on 2000 Android apps which have 11,584 components in total. Among these components, only 13 are protected by a permission and 3 out of the 13 permissions are protected by *signature* level. The rest 10 permissions are protected by *normal* level. Since the number of protected components is negligible, we do not take permissions into consideration for the results in this paper and only include exported and non-permission protected components in the list of reachable components.

B. Step 2: Precise CFG Building with FlowDroid

As mentioned in Section II-B, due to the Android event-driven nature, CFG of apps are imprecise. The imprecision is

[6]Referring to the second item presented above.

[7]This would be possible if the private key of the developer of the benign app is leaked.

Algorithm 1 Checking the exported status of a component

```
1:  procedure ISEXPORTED(comp, version)
2:      if $true == isExportedAttrDeclared(comp)$ then
3:          return $getAttrValue(comp, “exported”)$
4:      end if
5:      if $type(comp) == ContentProvider$ then
6:          if $version <= 16$ then
7:              return $true$
8:          else
9:              return $false$
10:         end if
11:     end if
12:     if $true == isIntentFilterDeclared(comp)$ then
13:         return $true$
14:     end if
15:     return $false$
16: end procedure
```

mainly caused by two kinds of methods: lifecycle methods and callback methods. We use FlowDroid [11] to model lifecyle and callback methods of Android components. We modified FlowDroid to only take into account reachable components extracted at step1 to build the precise CFG. Since we experienced that FlowDroid cannot properly analyze some apps because of the memory limitation, the precise CFG does improved the efficiency of FlowDroid.

For lifecycle methods, because the call sequence is well-defined in the Android's documents, what is needed is to simulate the sequence to call all the lifecycle methods. Since a component has different states in its life time, when leaving a state, it may have different choices and thereby executing different lifecycle methods. Let us take an `Activity` as a concrete example. When another `Activity` comes into the foreground (`onPause()` will be executed) two lifecycle methods may be selected by the Android system depending on user events. If the user returns to the activity, the `onResume()` is executed. If the `Activity` is no longer visible, `onStop()` is executed. FlowDroid generates a graph where both methods are reachable.

For callback methods, FlowDroid has a collection of callback methods extracted by analyzing the Android documentation. Then, for each component, FlowDroid checks whether it contains callback methods or not. If a callback methods exists

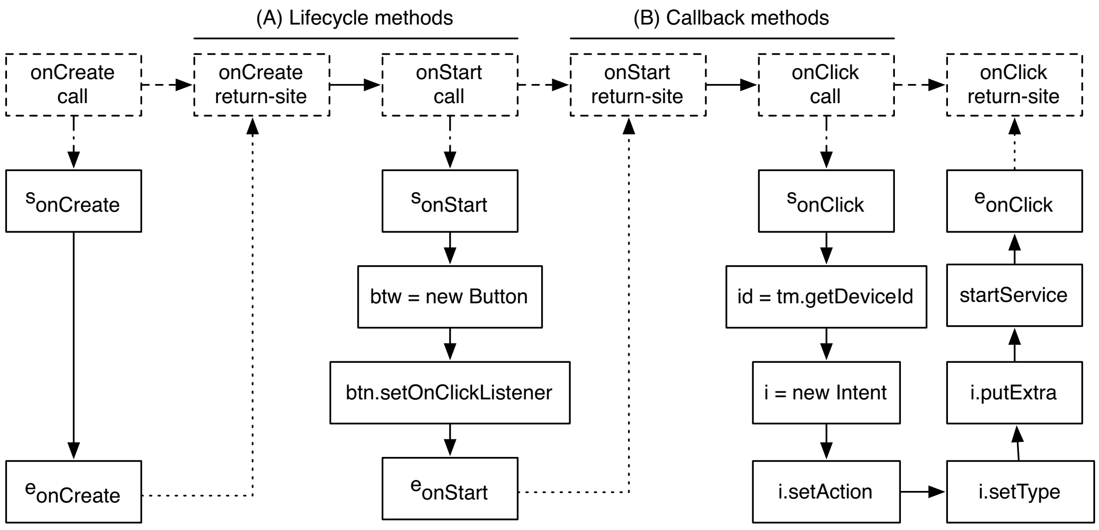


Fig. 5: The CFG of generated dummy main method for `GetDeviceIdActivity` illustrated in Listing 3.

in a component, FlowDroid makes the component reachable in the application graph.

To model the application graph, FlowDroid generates a dummy main method to call all the involved components and their lifecycle and callback methods. The CFG of generated dummy main method for `GetDeviceIdActivity` illustrated in Listing 3 is shown in Figure 5. The lifecycle methods are connected from `onCreate()` to `onStart()` (shown in A) and the callback methods are also connected from `onStart()` to `onClick()` (shown in B). Method `onClick()` is only called after method `onResume()` which simulates an `Activity` going to the running state. Because `onResume()` is not explicitly overridden in `GetDeviceIdActivity`, we do not model it in the generated dummy main method.

C. Step 3: Taint Analysis

Taint analysis is a kind of data-flow analysis. In this work, we leverage FlowDroid which is based on Heros [20], a IFDS/IDE problem solver, to perform a inter-procedural data-flow analysis for Android apps and Dexpler [21], a feature of Soot, to convert Android Dalvik bytecode to Soot's Jimple code for static Android application analysis.

The collection of *source* and *sink* methods used in this work is generated by SuSi [19]. SuSi is an open source tool that automatically identifies the *source* and *sink* methods in Android applications.

Because in this paper we focus on detecting potential component leaks, we also add all the *entry-points* and *exit-points* of components as *source* methods and *sink* methods respectively. The original FlowDroid is only sensitive to the *sinks* exactly defined in its configuration document. The problem is that FlowDroid will ignore all the override methods of the defined *sinks*. For example, if an *sink* `startActivity()` of class `Activity` is defined and `startActivity()` (or `this.startActivity()`) is called in the body of class `CustomActivity` which extends from class `Activity`, FlowDroid will not take into account `startActivity()` in `CustomActivity` as a *sink* even it should be. We corrected FlowDroid's strategy of identifying *sink* methods so that FlowDroid is sensitive to all the override *sink* methods as well.

From the leaks detected by FlowDroid, we filter all the non-potential component leaking paths if it does not starts with a *component entry-point* method or if it does not ends with a *component exit-point* method. If a potential component path starts with a *component entry-point* method, it means a PPCL is detected. If a potential component path ends with a *component exit-point* method, it means a PACL is detected.

D. Step 4: Validation

In order to validate the reported potential leaks, we developed a prototype tool called PCLeaksValidator, which automatically generates Android apps to check the validity of reported potential leaks. For a PPCL, the generated app contains only one component (Activity), which appropriately prepares an Intent and uses it to launch the target component. The extra data of the generated app always uses the default value we defined no matter what key of the extra is. For a PACL, the generated app contains also only one component, where the type depends on the *exit-point* of the PACL.

```
class Service1 extends Service{
 int onStartCommand(Intent i,int f,int id){
  String mail = i.getStringExtra("mail-body");
  Log.i("PCLeaksValidator", mail);
  return;
}}
```

Listing 4: The code generated by PCLeaksValidator for Listing 3.

Take Listing 3 as a concrete example. The code generated by PCLeaksValidator is shown in Listing 4, in which the Intent-Filter of `Service1` is also appropriately configured so that it can be launched by Application 2. Note that for PACLs, `PCLeaksValidator` uses method `Log.i` as sink method for all the generated apps.

We manually run the two apps (one is the analyzed app and the other is generated by PCLeaksValidator) to validate the exploited component leak. Currently, we are not able to automatically run the two apps to validate the results. Because it needs us to automatically trigger the communication event between the tested two apps. As introduced by David et al. [22], simulating user interaction is currently a main challenge for dynamic application analysis. In addition, some communication events rely on the user inputs at run time and thereby make them becomes harder to be automatically triggered. As future work, we would like to enhance PCLeaksValidator to automatically validate the two apps.

VI. EVALUATION

In Section VI-A, we present our experimental results of running PCLeaks on a set of 2000 real-world applications. Then, we detail two case studies (one for PPCL and the other for PACL) in Section VI-B and Section VI-C respectively.

A. Experimental Results

We run PCLeaks on 2000 apps randomly selected from the Google play store. The computer used for the experiment has a Core i7 CPU. We give 8 Gb for the Java VM heap. The apps are different from the collection we use to perform a short study about the permission protection of components. In average, PCLeaks processes an app in about 40 seconds.

We experienced that FlowDroid cannot properly analyze some apps (too much memory consumption or hangs). As PCLeaks is strongly dependent on FlowDroid, it shares the same problem. So we start by analyzing 2453 apps and keep only 2000 apps that work with FlowDroid. Among the 2453 apps, 453 of them could not be processed (e.g., due to errors of *insufficient memory*, *Type mask not found for type* or *Manifest contains more than one manifest node*). Thus, in this paper, we show our experimental results for 2000 apps.

Potential component leaks detected by PCLeaks are shown in Table I. PCLeaks reports 143 PACLs among 43 apps, which is shown in row 1, where only implicit Intents are taken into consideration. There is a significant difference between explicit and implicit Intents for PACL results. PCLeaks reports 15,260 leaks among 1149 apps, where 14,286 leaks are Activity Hijacking Leaks. These results match our expectation that Android apps are using ICC mechanism to transfer data between components, and the most used ICC method is `startActivity` [9]. The good news is that nearly 99% of detected PACLs are using explicit Intents, which is very difficult to be used by malicious apps to leak the sensitive data. That is why in this paper we do not take into account explicit Intents as PACLs when validating leaks.

For PPCLs, PCLeaks reports 843 leaks among 147 apps, which is shown in row 3, where non-exported and permission protected components are excluded. Taking into account the permission attribute of components has little impact on the results: only 5 out of 848 leaks are protected by permission. That means most of the developers are not accustomed to use permission to protect their exported components. This also confirms our permission related short study described in Section V-A where only 13 out of 11,584 components are protected by permission. If we count the non-exported components for PPCLs, the number of detected results are almost six times the number of results excluding the non-exported components. This is good news, as it shows that most of the components are non-exported, which avoids the components to be attacked via Intent spoofing.

TABLE I: The experimental results of detected potential component leaks.

Leaking Type	#. of Leaks	#. of Apps
PACLs (without explicit Intent)*	143	43
PACLs (with explicit Intent)	15260	1149
PPCLs (without permission, without non-exported)*	843	147
PPCLs (with permission, without non-exported)	848	150
PPCLs (with permission, with non-exported)	5540	514

* We take these two rows as potential component leaks result, the other rows are used to demonstrate the influence of explicit/implicit Intent, permission and export attributes to the results.

We randomly select 20 leaks (10 for PACLs and 10 for PPCLs) and run PCLeaksValidator on them. Then, we manually run the generated apps with their related source apps to check the detected leaks. We confirm that 7 PACLs and 8 PPCLs are true positives. The false positives are introduced by the condition-insensitivity of PCLeaks, insufficient string analysis of PCLeaksValidator or bugs in specific case. For example, a false positive is caused by a leak containing an unfeasible condition in its path. Since PCLeaks is currently condition-insensitive, it over-approximates all the possible paths whenever condition statements exist. Another confirmed false positive comes from insufficient string analysis. Since PCLeaksValidator performs a simple string analysis that only traverses the single intra-procedural control-flow graph to determine the value of a string variable. If the value of strings are not determined, we simply ignore them currently. This makes PCLeaksValidator yield false alarms (e.g., the key of an *extra data* is missing). In the future work, we would like to perform precise string analysis [23] to obtain better results.

Figure 6 classifies the detected PACLs and PPCLs according to the types of leak. The highest number of detected potential component leaks are Activity Launch Leaks. PCLeaks reports 534 leaks on Activity Launch Leaks. Indeed, it is easy to launch an Activity since all the Activity's information are defined in `AndroidManifest.xml`. In PACLs, the highest number of detected leaks are Activity Hijacking Leaks, where 110 leaks are reported. Since Activity is well-used in Android, the number of Activity Hijacking Leaks and Activity Launch Leaks confirm our expectation that the leaks related Activity should be the highest detected leaks.

The number of Broadcast related leaks (including system broadcasts) is higher than Service related leaks. This matches the design philosophy of Android components. On the one side, Broadcast is designed to communicate with other apps. It should have more potential leaks. On the other side, some apps (e.g., `In-app payments`[8]) are designed to provide functional services. But instead of exporting the Service component, those apps export a Broadcast for other apps and in

[8]Package name is com.beenverified.android.tests.in_app

```
E/BillingReceiver( 4740): Action: com.android.vending.billing.IN_APP_NOTIFY
E/BillingReceiver( 4740): Extra: inapp_signature => '<user input, can be sensitive data, e.g., deviceid>'
E/BillingReceiver( 4740): Extra: response_code => '<user input>'
```

Listing 5: The log data of `com.beenverified.android.tests.in_app` when it receives a broadcast.

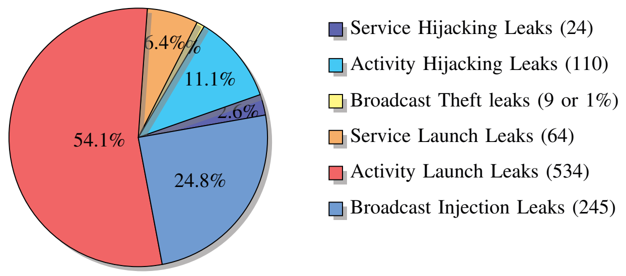


Fig. 6: Breakdown of detected PACLs and PPCLs by type.

the Broadcast they access the non-exported Service through explicit Intent. Therefore, it is normal to have more Broadcast related leaks.

To summaries, PCLeaks reports 986 leaks among 185 apps, where 5 of them contains both PACLs and PPCLs. we manually check 20 results, where 15 of them are true positives. PCLeaks reaches a precision of 75% on the randomly selected results.

B. *Case Study on PPCL*

In-app payments[9] is an Android application that offers in-app purchases. It contains a Broadcast Receiver named `com.example.dungeons.BillingReceiver`, which logs everything of the received Intent in method `logIntent()`. `logIntent()` is called by the entry point method named `onReceive()`. Both `logIntent()` and `onReceive()` are defined in class `com.example.dungeons.BillingReceiver`.

PCLeaks reports a PPCL for **In-app payments**. To verify the report, PCLeaksValidator automatically generates an app, which sends a broadcast message to the detected app with action named `com.android.vending.billing.IN_APP_NOTIFY` and two extras within the Intent. The two extras are named inapp_signature and response_code. Then, we manually run the two apps on our test device. No matter what data stored in the two extras, **In-app payments** logs all of them. A piece of the log data is shown in Listing 5.

Note that malicious apps can use this PPCL (without user intervention) to communicate with each other to avoid transferring data directly between malicious apps and thereby bypassing the detection of some specific malware detection tools. For example, **Malicious app 1** first send the data to **In-app payments** through a broadcast. Then, **In-app payments** logs all the data it received to disk. At last, **Malicious app 2** parses the log data of **In-app payments** to obtain the data transferring from **Malicious app 1**.

[9]https://play.google.com/store/apps/details?id=com.beenverified.android.tests.in_app

C. *Case Study on PACL*

In this case study, we show an application which contains potential active component leak. More specifically, it contains an Activity hijacking leak. **GetPhoneInfo**[10] is an app to obtain the information of the running Android operating system as well as the running phone itself. The obtained information include SubscriberId, name of the phone (e.g., HTC One), SimOperator Name (e.g., Orange) and many others.

All the sensitive data (phone's information) are obtained in method `getInfo()` of class `GetPhoneInfo`. Then, an implicit Intent is initialized with an action named `android.intent.action.SEND` and a type named `message/rfc822`. The sensitive data is stored into the Intent with an extra named `android.intent.extra.TEXT`. After that, `startActivity()` is called to communicate with other applications (or components), which also send the sensitive data to other applications.

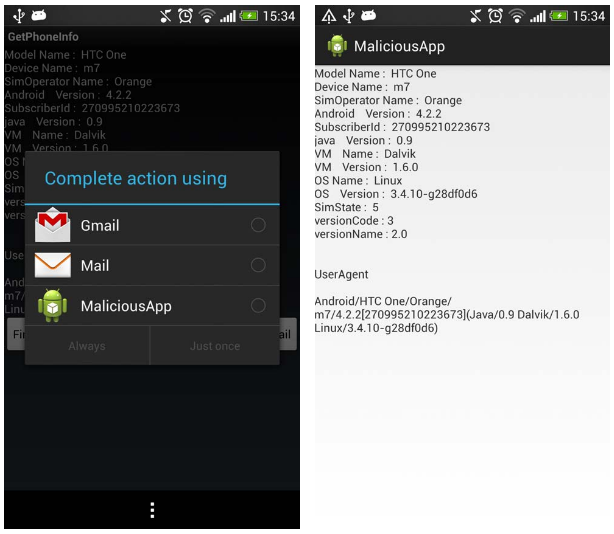
MaliciousApp is an Android application automatically generated by PCLeaksValidator to test whether the app **GetPhoneInfo** will actively leak sensitive data or not. In MaliciousApp, we developed a component which can receive action `android.intent.action.SEND` and the data type is set to `*/*`.

We run the two apps on our test device. The results (screenshots) are shown in Figure 7. Note that to better illustrate the results, we refactored the generated malicious apps to explicitly show the received data in a text view. Figure 7a shows the app selection screen when `startActivity()` is executed. Figure 7b shows the received sensitive data when user launch the Malicious application. The sensitive data is transferred from Figure 7a to Figure 7b. That means a PACL can become a real leak, which exposes the user's private data to other applications.

VII. Limitations

At the moment, we do not handle URIs, which are well used by Content Provider. Therefore, PCLeaks is not able to exploit potential leaks on Content Provider. PCLeaks is not aware of multiple threads, reflections and condition statements. PCLeaksValidator does not handle URIs to generate incomplete malicious apps, which is not able to exploit the corresponding potential component leaks. Currently, PCLeaksValidator only performs string analysis within a single method which may cause false alarms. PCLeaksValidator is not able to run the apps to automatically validate the reported potential component leaks. Some rarely used ICC methods such as `startActivities` are not tackled in this work. The native code used by some apps is not analyzed as well.

[10]https://play.google.com/store/apps/details?id=hello.GetPhoneInfo



```
GetPhoneInfo
Model Name :  HTC One
Device Name :  m7
SimOperator Name :  Orange
Android   Version :  4.2.2
SubscriberId :  270995210223673
java   Version :  0.9
VM   Name :  Dalvik
VM   Version :  1.6.0
```

```
Model Name :  HTC One
Device Name :  m7
SimOperator Name :  Orange
Android   Version :  4.2.2
SubscriberId :  270995210223673
java   Version :  0.9
VM   Name :  Dalvik
VM   Version :  1.6.0
OS Name :  Linux
OS   Version :  3.4.10-g28df0d6
SimState :  5
versionCode : 3
versionName : 2.0


UserAgent

Android/HTC One/Orange/
m7/4.2.2[270995210223673](Java/0.9 Dalvik/1.6.0
Linux/3.4.10-g28df0d6)
```

(a) Application Selection when `startActivity` is executed. (b) Malicous app hijacks the sensitive data.

Fig. 7: A case study about Activity Hijacking Leak. Note that the name “MaliciousApp” is used to better demonstrate the Leak. The real malicious app may use lifelike name and icon to confuse the user.

VIII. Related Work

PCLeaks is developed to detect potential component leaks in Android apps. Information leaks and component vulnerabilities detection in Android apps are two main research topics mostly related to our work.

Information leak detection has been studied for decades and new leaks are still discoved on contemporaty software [24, 25, 26]. Among them, both static analysis and dynamic analysis are performed. One of the most sophisticated static analysis approach is FlowDroid [11], a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android apps. It tracks tainted data between pre-defined *source* and *sink* methods. If a tainted data is transferred from a *source* method to a *sink* method, then a information leak is reported. Several other approaches including SCanDroid [27], LeakMiner [18] and AndroidLeaks [8] also use static analysis to detect privacy leaks. TaintDroid [28] is one of the most sophisticated dynamic approach on detecting privacy leaks in Android apps. It extends the Android mobile-phone platform and tracks the flow of privacy sensitive data through third-party apps at run time. CopperDroid [29] is another dynamic testing tool which uses stimulating technique to exercise the app to find malicious activities. More recently, DroidTrack [30] tracks and visualize the sensitive information diffusion on Android to prevent sensitive data leaks. However, these approaches mainly focus on detecting real private data leaks, our approach is different from them and focuses on detecting potential component leaks, that is leaks that could be exploited.

Component Vulnerabilities detection is another hot topic related to our work. CHEX [16] is a tool to detect component hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources. The entry-point model of CHEX requires an enumeration of all possible “split orderings” which is not necessary in PCLeaks. ComDroid [31] and Epicc [7] are another two tools that focus on detecting inter-component vulnerabilities. However, they do not perform sensitive data-flow analysis. In other words, ComDroid and Epicc are able to detect component vulnerabilities (e.g., Activity Hijacking, Broadcast Injection). But they do not detect whether a component vulnerability leaks sensitive data or not. PCLeaks is different from these tools that it is based on component vulnerabilities to detect potential sensitive leaks.

The related works introduced in this section are either focusing on privacy leaks detection or focusing on component vulnerabilities. Our approach is using both sides to perform potential component leaks detection. ContentScope [10] is a tool similar to our approach which detects sensitive data leaks on components in Android applications. However, it only focuses on `Content Provider`. ContentScope also detects content pollution in Android applications, which is not handled currently by PCLeaks. But with little modification (e.g., defining pollution methods as *sink* methods for PCLeaks), our approach is able to detect component pollution. However, detecting component pollution is out of scope of this paper, we take it as our further work.

Other state-of-the-art works are trying to enhance the user privacy by permission removal [32, 33] or trying to reduce the attack surface of Android applications [34, 35]. DroidForce [36] attempts to enforce complex, data-centric and system-wide policies for Android apps to constraint the malicious behavior. Those ideas could be used to complement our approach to prevent components from leaking sensitive data but are out of scope of this paper.

IX. Conclusion

In this work, we present PCLeaks, a tool to exploit potential component leaks and PCLeaksValidator, a tool which automatically generates a correspond malicious apps to validate the results of PCLeaks. Concretely, PCLeaks first builds a precise control-flow graph for the analyzed apps. Then, it performs static taint analysis with a well-defined set of *source* and *sink* methods to identify potential active component leaks and also potential passive component leaks. We test PCLeaks on 2000 apps randomly selected from Google Play. Among the 2000 apps, PCLeaks reports PACLs in 43 apps with 143 leaks and also reports PPCLs in 147 apps with 843 leaks. By manually checking 20 results through running the generated malicious app with its source app, we confirm that 15 (or 75%) of them are true positives.

In the future work, we would like to enhance PCLeaksValidator to support automatically validating the exploited leaks. Also, we are working towards automatically repairing Android application containing potential component leaks.

Acknowledgment

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289 and the AFR grant 1081630. We thank our anonymous reviewers for their helpful comments.

References

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

[2] *AppBrain Stats*. http://www.appbrain.com/stats/number-of-android-apps. 2014.

[3] *Kaspersky security bulletin 2013. malware evolution*. http://kasperskycontenthub.wpengine.netdna-cdn.com/report/files/ksb13_EN_lite-1.pdf. 2014.

[4] *IT Threat Evolution: Q2*. https://www.securelist.com/ en/analysis/204792299/IT_Threat_Evolution_Q2_2013. 2014.

[5] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. "WHYPER: Towards Automating Risk Assessment of Mobile Applications." In: *Proceedings of the 22st USENIX conference on Security symposium*. 2013.

[6] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. "Checking App Behavior Against App Descriptions". In: *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*. Hyderabad (India), 31 May - 7 June, 2014.

[7] 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: *Proceedings of the 22nd USENIX Security Symposium*. 2013.

[8] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao 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. Vienna, Austria: Springer-Verlag, 2012, pp. 291–307.

[9] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Rasthofer Siegfried, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. *I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis*. English. Tech. rep. 978-2-87971-129-4_TR-SNT-2014-9. Apr. 2014.

[10] 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.

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

[12] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications". In: *MoST 2012: Mobile Security Technologies 2012*. Ed. by Hao Chen, Larry Koved, and Dan S. Wallach. San Francisco, CA, USA: IEEE, May 2012.

[13] Patrick P. F. Chan, Lucas C. K. Hui, and S. M. Yiu. "DroidChecker: analyzing android applications for capability leak". In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. WISEC '12. Tucson, AZ, USA: ACM, Apr. 2012, pp. 125–136.

[14] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. "Detecting privacy leaks in Android Apps". In: *International Symposium on Engineering Secure Software and Systems - Doctoral Symposium (ESSoS-DS2014)* (2014).

[15] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *POPL '95*. 1995, pp. 49–61.

[16] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "CHEX: statically vetting Android apps for component hijacking vulnerabilities". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 229–240.

[17] Zhibo Zhao and Fernando C. Colón Osorio. ""TrustDroid;": Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking". In: *MALWARE*. 2012, pp. 135–143.

[18] Zhemin Yang and Min Yang. "LeakMiner: Detect Information Leakage on Android with Static Taint Analysis". In: *Third World Congress on Software Engineering (WCSE 2012)*. 2012, pp. 101–104.

[19] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks". In: *The 2014 Network and Distributed System Security Symposium (NDSS)*. 2014.

[20] Eric 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. 2012, pp. 3–8.

[21] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves 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*. Beijing, China, 2012.

[22] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. "SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps". In: *The 2014 Network and Distributed System Security Symposium (NDSS)*. 2014.

[23] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. *Precise analysis of string expressions*. Springer, 2003.

[24] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 199–212.

[25] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. "TAJ: effective taint analysis of web applications". In: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, pp. 87–97.

[26] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications." In: *The Network and Distributed System Security Symposium (NDSS 2011)*. 2011.

[27] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. "SCanDroid: Automated security certification of Android applications". In: *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa* (2009).

[28] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *OSDI*. Vol. 10. 2010, pp. 255–270.

[29] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors". In: *EuroSec, April* (2013).

[30] Shunya Sakamoto, Kenji Okuda, Ryo Nakatsuka, and Toshihiro Yamauchi. "DroidTrack: Tracking and Visualizing Information Diffusion for Preventing Information Leakage on Android". In: *Journal of Internet Services and Information Security (JISIS)* 4.2 (2014), pp. 55–69.

[31] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing inter-application communication in Android". In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. MobiSys '11. Bethesda, Maryland, USA: ACM, 2011, pp. 239–252.

[32] Quang Do, B. Martini, and K.-K.R. Choo. "Enhancing User Privacy on Android Mobile Devices via Permissions Removal". In: *System Sciences (HICSS), 2014 47th Hawaii International Conference on*. 2014, pp. 5070–5079.

[33] Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, and Yves Le Traon. *Improving privacy on android smartphones through in-vivo bytecode instrumentation*. Technical Report. 2012.

[34] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android". In: *IEEE Transactions on Software Engineering (TSE)* (2014).

[35] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. "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*. Essen, Germany, 2012.

[36] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. "DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android". In: *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. 2014.