LinkFlow: Efficient Large-Scale Inter-app Privacy Leakage Detection

Yi He[1], Qi Li[1](✉), and Kun Sun[2]

[1] Department of Computer Science, Graduate School of Shenzhen, Tsinghua University, Shenzhen, China
heyi14@mails.tsinghua.edu.cn, qi.li@sz.tsinghua.edu.cn

[2] Department of Information Sciences and Technology, George Mason University, Fairfax, USA
ksun3@gmu.edu

Abstract. Android enables inter-app collaboration and function reusability by providing flexible Inter-Component Communication (ICC) across apps. Meanwhile, ICC introduces serious privacy leakage problems due to component hijacking, component injection, and application collusion attacks. Taint analysis technique has been adopted to successfully detect potential leakage between two mobile apps. However, it is still a challenge to efficiently perform large-scale leakage detection among a large set of apps, which may communicate through various ICC channels. In this paper, we develop a privacy leakage detection mechanism called LinkFlow to detect privacy leakage through ICC on a large set of apps. LinkFlow first leverages taint analysis technique to enumerate ICC links that may lead to privacy leakage in each individual app. Since most ICC links are normal, this step can dramatically reduce the number of risky ICC links for the next step analysis, where those ICC links are matched among leaky apps. We develop an algorithm to identify privacy leakage by analyzing ICC links and the associated permissions. We implement a LinkFlow prototype and evaluate its effectiveness with more than 4500 apps including 3014 benign apps from five apps marketplaces and 1500 malicious apps from two malware repositories. LinkFlow can successfully capture 6065 privacy leak paths among 530 apps. We also observe that more than 400 benign apps have vulnerabilities of privacy leakage in inter-app communications.

Keywords: Android · Privacy leakage · Large-scale detection

1 Introduction

As an open platform, Android allows users to install apps from the Google Play Store and third-party app marketplaces. Inter-Component Communication (ICC) mechanism enables communication between two components belonging to two different apps, and it allows developers to reuse another app's functionality without reinventing the wheel. However, the easy-to-use ICC can be misused in application collusion attacks [42] to bypass Android's permission-based security model, which only independently restricts individual apps from accessing sensitive resources. Therefore, malicious app developers may deliberately develop multiple apps that may collude via ICC to achieve permission escalation [19,24,27,31].

Researchers have developed a number of effective mechanisms to detect ICC-based privacy leakage between two apps [20,21,23,26,29,32,34,34,38,38,41,44,46,49,50]. However, it is difficult to directly apply those approaches in a large scale detection, since it is very time-consuming to check each pair of apps among the huge number of apps in one app marketplace. For example, according to the existing studies [26,29,34,38,52], it takes at least 5 min for the-state-of-the-art mechanisms to detect privacy leakage between two apps. Thus, when detecting if there exists privacy leakage among 500 apps, it will take more than 10 thousand hours (about 400 days) to analyze each pair of the 500 apps. Such a long detection delay is not acceptable.

There are two main challenges to efficiently detect the inter-app privacy leakage vulnerabilities among a large set of apps. First, we should be able to precisely resolve ICC APIs in each app and then identify inter-apps data flows through those APIs. Currently *Intent* can use about 40 ICC APIs to exchange information [9]. Moreover, we need to harvest the ICC parameters from the app bytecode and track inter-app data flows by matching ICC channels among different apps. For simplicity, we call those ICC channels as *ICC links*. Second, since there will be a huge number of ICC links among a large set of apps, it is difficult to enumerate all possible ICC links among those apps. Therefore, we should be able to reduce the scale of ICC link analysis without introducing false negative detection results.

In this paper, we propose a large-scale privacy leakage detection mechanism called *LinkFlow* that can efficiently detect privacy leakage through ICC among a large set of apps. Our mechanism is based on one key observation that the most of ICC links among all apps in one app marketplace are benign links. Therefore, instead of detecting leakages in all ICC links, we focus on identifying the ICC links among the app components that may leak information, so we can dramatically reduce the scale of targeted ICC links and significantly decrease the detection delays.

LinkFlow uses static taint analysis to filter out the leaky components that may lead to privacy leakage for each individual app. Next, we propose an efficient ICC match algorithm to quickly mine out all the ICC links among those leaky components. We then separate the flows in the leaky components into two component sets, namely, *OutFlow set* and *InFlow set*, based on its leaky flow and use an ICC matching algorithm to find out ICC links between the two component sets. A privacy leakage vulnerability can be identified if an ICC link really delivers sensitive information. Our mechanism also supports incremental analysis when a new app is submitted to the app market. The analysis results generated by LinkFlow not only identify privacy leakage among the existing apps, but also provide guidelines for app developers to mitigate leakage during app development.

We implement a prototype of LinkFlow and evaluate it with 3014 real-world benign apps and 1500 malicious apps. It successfully identifies 6065 privacy leakage paths among 530 apps. It detects 4622 abnormal data flows among 87 apps, which introduce privacy leakages among their inter-apps data flows. LinkFlow only takes around 5 min to analyze one app against the remaining apps; most time is consumed by the taint analysis and the ICC extraction, which are one-time operations. It takes about 10 s to detect if there exists privacy leakage between an app and the rest 4513 apps, which is efficient in large scale detection.

In summary, this paper makes three folds of contributions.

- We propose a large scale app detection framework to efficiently detect vulnerable inter-app data flows that may lead to privacy leakages.
- We propose an ICC matching algorithm that searches the ICC links and identifies sensitive inter-app data flows in order to detect privacy leakage.
- We implement a LinkFlow prototype and use it to study the privacy leakage in real app stores. The experimental results show that it can finish the detection quickly and effectively identify the vulnerable ICC links among apps.

2 Background

Android apps usually consist of multiple reusable components that can communicate with each other either internally or externally via Inter-Component Communication (ICC) mechanisms. There are four types of components, namely, *Activities*, *Services*, *Content Providers*, and *Broadcast Receivers*.

Android provides flexible APIs for components to exchange data or share services through ICC. Components use *Intent* messages or *URI* to describe the corresponding components. *Intent* can be either explicit or implicit [7], where explicit *Intent* requires setting package names and component names of the recipient components in the *Intent* messages and implicit *Intent* needs to define its type by specifying the actions, the categories, and other flags. Using implicit *Intent*, one component that has registered *Intent Filters* to handle one Intent can receive and respond to the *Intent* message.

Android is a permission based operating system and restricts resource accesses by declaring different security level permissions. The protection levels can be one of four levels: *normal*, *dangerous*, *signature*, or *signatureOrSystem*. In particular, the later three levels are used to protect resources of apps. Apps declare their permissions in the *Manifest.xml* files with different protection levels, and these permissions are granted forever upon the apps' installation. The usage of ICC also introduces potential privacy leakages that can bypass the permission system [23,34,38,40,41]. For instance, if one component with the permissions to access sensitive data is exported and not protected by *signature* level permission, it may be misused by another component in a privilege escalation attack. Our work focuses on detecting potential ICC privacy leakages among a large number of apps.

3 Threat Model

The component reuse via ICC on Android poses serious security problems against the permission-based security model. Malicious apps may misuse the components in other benign apps to grasp the corresponding permissions, or they can collude to accumulate permissions that will not be granted to a single app.

Component Hijacking. When the exported components of benign apps can be leveraged by malicious apps, sensitive information may be leaked out from the benign apps. For instance, *GoMsg* is a popular app with over a million downloads for sending messages and making phone calls. Its message sending component is exported and can be used by any other apps without permission check. Therefore, malicious apps such as SmsZombie [11] can leverage GoMsg to send premium-SMS. Such component hijacking attacks have been identified in a number of built-in and third-party apps, including Activity Hijacking, Service Hijacking, and Broadcast Theft [23,41].

Component Injection. When benign apps send *Intent* to the corresponding components that have been replaced by components of malicious apps, ICC is manipulated by the malicious apps and ICC information will be leaked to malicious apps [39]. For instance, *One-Password* is a popular password store app, which uses Dropbox SDK to implement the OAuth login function. However, malicious apps can register the same Intent Filter as One-Password to intercept the OAuth access token [14] or return the fake token to One-Password to log in with the attacker's account.

Application Collusion. Since permission model focuses on restricting the access capability of an individual app, it cannot detect application collusion attacks that malicious apps may collude via ICC to achieve permission escalation. For instance, *SoundComber* [47] is a sound-based Trojan that uses sound sensors to record user's keyboard input and other audio data such as phone conversations. Android security protection may deny the installation of apps requesting both sensitive sensors and network permissions. However, SoundComber does not requires the network permission, since it can use ICC to transfer the sensitive data to another colluding app that has the network permission to send the data to a remote server.

In this paper, we focus on developing an efficient approach to detecting inter-app privacy leakage incurred by ICC channels. Privacy leakages incurred by other interfaces (e.g., interfaces defined by Android Interface Definition Language (AIDL)) are not the focus of our paper [20,42].

4 LinkFlow Overview

In this section, we present an overview of LinkFlow architecture that aims to detect privacy leakage across multiple apps on a large scale. It is built upon the following two key observations, which have been correctly verified by our experimental results.

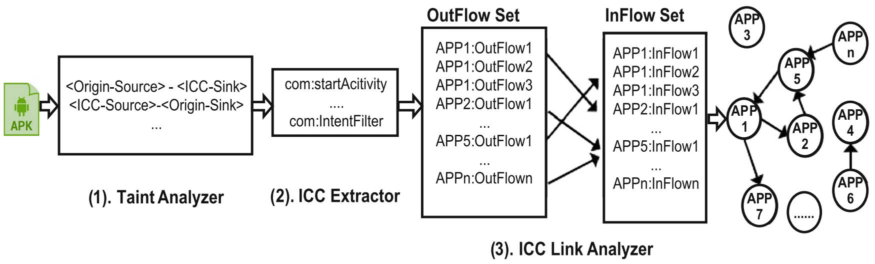


Fig. 1. LinkFlow architecture

Observation 1: *Not all components of an app interact with the other components of the app or other apps, and thus they will not leak privacy.* One component cannot be accessed by any other components (of the app or other apps) if it does not communicate with them via ICC. Therefore, we only need to analyze the code of app components that interact with the other components of the same app or other apps. In other words, we focus on the components that may transfer data out of the app.

Observation 2: *Only a small portion of ICC links are leaky, and the most of ICC links are benign.* Thus, we can identify privacy leakage by analyzing ICC links that deliver sensitive information, which is protected by the permission level at Dangerous, Signature, or SignatureOrSystem.

Figure 1 shows the architecture of LinkFlow, which consists of three major components *Taint Analyzer*, *ICC Extractor*, and *ICC Link Analyzer*. Given a set of apps, these three components run in sequence to identify potential ICC-based leakage among all the apps. First, the taint analyzer performs static taint analysis on each app to identify flow paths that may leak sensitive information. Second, ICC extractor is responsible for extracting and resolving ICC methods and the parameters in ICC links according to flow paths generated by taint analyzer, and generates two sets of components for outgoing flows and incoming flows, respectively. By leveraging the flow paths, ICC extractor significantly reduces the number of ICC links for analysis. Finally, ICC link analyzer matches the ICC links of apps to find out the abnormal data flows that may incur privacy leakages among apps. The first two steps can precisely screen out the leaking components in apps and the ICC APIs used in those components. Based on the reduced ICC links, the third step identifies the leaking ICC links and can generate an ICC link graph to better illustrate the leaking paths.

4.1 Taint Analyzer

The usage of taint analysis is to detect the leaky components that contain ICC-based leaky data flows. A leaky data flow is a path starting from the source API that accesses the sensitive data to the sink API that sends this data out of the application or device. We inspect all leaky flows that send data to ICC APIs (e.g., *Intent.putExtra*) that are called *ICC-Sink* or read data using ICC APIs (e.g., *Intent.getExtra*) that are named *ICC-Source* in this paper.

4.2 ICC Extractor

We use ICC extractor in the second step to extract the parameters of *Intents*, the ICC APIs, the components' *Intent Filters* and other necessary messages. Since it only extracts the ICC APIs that match the flows in ICC-Sink or ICC-Source and analyze the components that may communicate with other apps, it dramatically reduces the number of ICC links under analysis. After this step, we can extract the leaky components and leaky path using ICC APIs in each individual app. We define a tuple $A = \{C, P, F\}$ for each app to record the analysis results, where

- **C** is the set of components in one app. For each component $c \in C$, c contains the ICC extractor's analysis results that are extracted from the Manifest and bytecode. It consists of a set of *Intent Filters*, a set of permissions used in the component, and a set of ICC methods. In addition, it also includes *Intent* messages that are referred as Exit Points.
- **P** is the set of total permissions declared in the app's manifest file.
- **F** is the set of flows resulted from the static taint analysis.

4.3 ICC Link Analyzer

After obtaining the tuple of an app from ICC Extractor, we perform the ICC link analysis to analyze all the necessary data of the leaky components and use a fast ICC matching algorithm to enumerate all ICC links among these apps to identify privacy leakage. In particular, we accurately infer if ICC links really incur privacy leakage by evaluating the corresponding permissions that are mapped from the ICC APIs. After this step, LinkFlow can generate an analysis report to list all the potential privacy leakage among apps, e.g., among all apps in an app marketplace. This report provides guidance to mitigate the vulnerabilities or ban the malicious apps. In particular, it allows app developers to understand what components could be leveraged by other apps and thus help reduce the chances of privacy leakage.

5 LinkFlow Design

In this section, we present the design details of LinkFlow. As shown in Sect. 4, it has three steps to detect privacy leakages.

5.1 Step 1: Taint Analysis for Single App

We leverage static taint analysis to analyze intra-app data flows and trace how the data is created, modified, and consumed. In Android, app actions are triggered by the user events that are handled by specific callback methods. For instance, the *onClick* method is called when the user clicks a button. One app's state is changed by calling the components' lifecycle callback methods such as *onStart* when a component is started or *onResume* when a component is resumed. In order to perform control flow analysis, we need to generate calls for these callbacks that do not have direct calls in the code. LinkFlow generates a dummy main method to be used as the entry-point and creates direct calls for those callback methods. Then LinkFlow uses Spark algorithm [36] to construct a call graph for these methods and perform forward and backward inter-procedural data flow analysis based on the call graph [45].

After discovering all the sensitive intra-app data flows, we can obtain a set of paths recording the sensitive data flows as follows:

Flows(app) = {*path1: source1* $\sim$ *sink1; path2: source2* $\sim$ *sink2; ...*}

Where sources are APIs that return sensitive data of the app or Android system (e.g., reading contacts) and sinks are APIs that transmit sensitive data out of the app (e.g., via an HTTP connection). Since we target at identifying leaky paths that use ICC APIs to obtain and then leak privacy information, we separate ICC API related sources/sinks from the original sources/sinks and name them as ICC-Sources and ICC sinks. For simplicity, we call the sources and sinks excluding ICC sources and ICC sinks as *origin-sources* and *origin-sinks*, respectively.

We focus on two types of leaky paths, origin-source $\sim$ ICC-Sink and ICC-Source $\sim$ origin-sink, since the components with origin-source $\sim$ ICC-Sink paths may suffer component injection attacks and the components with ICC-Source $\sim$ origin-sink paths may suffer component hijacking attacks. When one Intent is sent out of one app via ICC-Sink, this Intent can only be received by the components using the targeted ICC-Source. We summarize the ICC-Sinks and the targeted ICC-Sources in Table 1. Components containing these two types of leaky paths are considered as leaky components. In this way, LinkFlow can find out the leaky components that send sensitive data out the app via ICC APIs or read data in via ICC APIs.

Table 1. ICC-Sinks with targeted ICC-Sources

ICC-Sinks	Targeted ICC-Sources
Context: send*BroadCast(Intent,...)	BroadcastRecevier: onReceive(Intent)
Activity: startActivity*(Intent, ...)	Activity: getIntent()
Context: startService(Intent, ..)	Service: onBind(Intent)
ContentResolver: insert, query, delete, update	(depend on the URI)

5.2 Step 2: ICC Extraction for Single App

Considering a large number of ICC links among apps and most of them are not leaky, we can reduce the ICC link scale by only checking the leaky ICC links among the leaky components and ignoring the normal ICC links, as shown in

Fig. 2. Instead of inspecting all ICC links among all apps, we only need to check the ICC links between two leaky components, since our goal is to find out the leaky ICC links. However, exist ICC leak detection tools such as IccTA [38] and DidFail [34] need to analyze all ICC links to identify the leaky ones, and most of the analysis time is consumed by analyzing the normal ICC links.

After identifying all leaky components, we collect the Intent parameters of the ICC APIs and obtain the Intent Filters of those components. Since inter-component communications via Intent message mechanism are dynamically resolved by Android system, it is difficult for static analysis tools to analyze those links between components. Therefore, to analyze the ICC links among apps, we need to precisely resolve the ICC methods and the Intent messages. There are various ICC APIs and a large number of Intent data handling methods such as *intent.getStringExtra*, *intent.getLongExtra*, etc. We extract the Intent Filters of components from the *Manifest.xml* and the ICC APIs parameters from the bytecode.

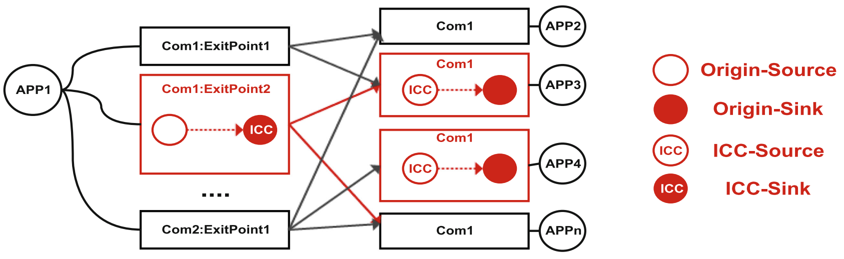


Fig. 2. Only checking the leaky ICC links (red arrow) among the leaky components. (Color figure online)

Now we combine the ICC extraction results with the taint analysis results to obtain the following flow information.

– **Method and Class** with the full method signature in Soot format [35], for instance, <*android.telephony.TelephonyManager: java.lang.String.get DeviceId()*>.
– **Component** where the Flows belong to. We need to find out in which component the source/sink method is called.
– **Category** of the source/sink API. The SuSi [43] project provides a detailed category of the API. As one efficient way to express the behavior of the flows, it is easy for end users to comprehend how the sensitive data is used.
– **Permission** associating with the API. We use the relation map provide by PScout [17] to achieve it.
– **Exit Point.** An Exit Point is an ICC method used to send Intent and communication with other components. When apps start a new context in the *Exit Points* and the Intent messages are passed by the Android OS, the data-flow will discontinue. Therefore, we need to extract all Exit Point methods.

The Component and the Exit Point information are used to identify the ICC-based leaky paths and obtain the Intent messages' parameters and the Intent Filter. The methods, classes, permissions, and categories are used to describe the sources/sinks and identify the privacy leakage due to the misuse of the sensitive API.

5.3 Step 3: ICC Link Analysis for All Apps

The first two steps have extracted detailed flow information for each individual app, and now we can obtain the leaky components, the Intent messages, and the ICC methods of all the apps to perform ICC link analysis.

We define *abnormal data flow* as a leaky ICC link that indicates sensitive data sent out via one ICC-Sink method in App1 and received by one ICC-Source method in App2. We record one abnormal data flow as $A = \{\mathit{App1{:}\ outflow} \sim \mathit{App2{:}\ inflow},\ \mathit{outflow} \in \mathit{OutFlow},\ \mathit{inflow} \in \mathit{InFlow}\}$, where InFlow is a set of exported components with an abnormal flow of ICC-Source to origin-sink that reads data from other components and OutFlow is a set of components (may not be exported) with an abnormal flow of origin-source to ICC-Sink that sends data to other components.

We develop an ICC matching algorithm to find out all potential ICC links among the leaky apps and construct an ICC graph where leaky apps are nodes and ICC links are links. The ICC matching algorithm is shown in Algorithm 1. First, we traverse the OutFlow Set and check the ICC APIs and parameters used in the *outFlow* to determine if the Intent is implicit or explicit. For explicit Intent, the receiver components are defined and the destination can be directly obtained from the parameters of Intent. For implicit Intent, there may exist multiple receiver components depending on the apps installed in user's device. To verify if there is an implicit ICC link between an *outFlow* and an *inFlow*, we need to evaluate the ICC-Sink of the *outFlow* and the ICC-Sources of each *inFlow*: (1) if their methods and target component types are matching (as shown in Table 1) and (2) if the Intent Filters of the ICC-Sources can receive the Intent sending by the ICC-Sinks. For instance, to identify the apps that use ICC to link to Contacts Manager, since Contacts Manager's ICC-Sink is *startService*, we need to check the InFlow set to find out the Service Components with the onBind ICC-Source. Because we have extracted the action, categories, and flags of the Intent sent by the ICC-Sink in first two steps, we can check if the Intent Filter of the ICC-Source component can receive the Intent. Our methods can reveal all apps that contain services to handle the Intent sent by Contacts Manager and construct an ICC link graph to save these leaky links and the corresponding apps.

By traversing the ICC link graph, we can obtain the linked apps of each app and then generate a leaky report for each app. The report for an app can tell which apps use ICC to communicate with it and may cause privacy leakage. The report also provides the detailed leaky path, the potential privacy leakages, the potential permission leakages, and the risk level. We can determine the severity of these leaky API by using the categories of these APIs that are summarized

Algorithm 1. The ICC Matching Algorithm

```
Require: InFlowSet, OutFlowSet;
Ensure: LinkedFlowSet: linked ICC flow;
 1: $graph \leftarrow initLinkGraph()$
 2: for $inflow \in InFlowSet$ do
 3:     $intent \leftarrow inflow.ICCSink.intent$
 4:     if $isExplicitIntent(intent)$ then
 5:         $app \leftarrow getAppByPackage(intent)$
 6:         $addEdge(graph, app, inflow.app)$
 7:     else
 8:         for $outflow \in OutFlowSet$ do
 9:             $intentFilter \leftarrow outflow.intentFitler$
10:             if $intentFilter.canReceive(intent)$ then
11:                 $addEdge(graph, outflow.app,$
12: $inflow.app)$
13:                 $LinkedFlowSet.add(inflow,$
14: $outflow)$
15:             end if
16:         end for
17:     end if
18: end for
```

by SuSi [43]. We leverage PScout [17] to map APIs to their corresponding permissions. Then we can confirm if there are permission leaks by checking if the InFlow app contains the permission required by the origin-sink in the OutFlow app. If not, it means the InFlow app can leverage the permissions of the OutFlow app and the privilege escalation happen. In general, the report can help both app developers and users to mitigate the vulnerabilities or ban the malicious apps.

6 LinkFlow Implementation

LinkFlow extends FlowDroid [16] to implement the Taint Analyzer that provides precise taint analysis to efficiently identify all suspicious ICC flows. In particular, it leverages PScout [17] and Susi [43] to generate the required parameters for the taint analysis. Moreover, LinkFlow utilizes ICC Extractor to precisely infer the ICC parameters, e.g., the type of *Intent* values and the parameters of *Intent Filters*. LinkFlow significantly reduces the number of ICC links based on the flow taint analysis results.

6.1 Taint Analyzer

Our Taint Analyzer constructs a call graph for apps and then performs forward data flow analysis to find paths from the source API to the sink. Next, it performs backward dependence analysis to exclude the paths that do not have any dependence on the source APIs.

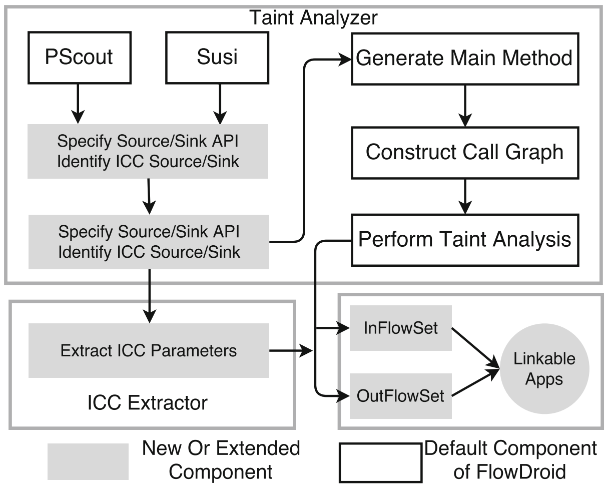


Fig. 3. Implementation of LinkFlow

As shown in Fig. 3, besides leveraging the basic taint analysis functionalities provided by FlowDroid [16], we extend FlowDroid in three First, we extend the source/sink analysis module to identify and analyze sensitive ICC source/sink, which is critical for reducing the complexity of the later ICC analysis. Second, we identify the callbacks of all four types of components and analyze source/sink to find out all non-isolated components that interact with other components. It can reduce the code base to be analyzed. Third, we extend FlowDroid to analyze *Service and BroadCast Receiver*.

Specifying Source/Sink APIs. Since taint paths start from a source API that read or generate private data and end in sink APIs that may leak privacy, we modify the FlowDroid's source/sink manager to analyze two specific types of flows, namely, flows from Origin-source to ICC-Sink and flows from ICC-Source to Origin-Sink. As shown in Fig. 3, we leverage Susi [43] to generate a detailed list of sources/sinks APIs with the API category information, and utilize PScout's [17] to associate APIs with the permissions they require. Thus, LinkFlow can accurately obtain sensitive source/sink APIs and track the data flow of these APIs. Note that, by analyzing two specific flows, we further improve the performance of LinkFlow by reducing the workload of backward flow analysis.

Analyzing ICC Parameters and Excluding Isolate Components. We need to identify components that contain ICC-sink methods may leak data, and thus we need to precisely extract the ICC parameters so as to analyze the Intent receiver components and what data they send. To achieve this goal, we traverse entire app packages by using Soot to analyze the usage of ICC APIs in all components. To reduce the code base under analysis, we need to exclude non-isolated components in LinkFlow. First, during traversing app packages, we also enumerate components that do not have ICC-sink methods. Second we analyze

Intent Filters in the *Manifest.xml* file to find out all public components. These two type components are non-isolated components that need no further analysis. Therefore, we only add these components' lifecycle methods as entry points to the dummyMain methods.

Constructing Call Graph. We extend FlowDroid to construct call paths for all components. In particular, FlowDroid cannot analyze Service and Broadcast Receiver as they use different callback methods. It cannot generate the callback method information in the dummyMain methods so that these methods will not be added to the call graph and cannot be analyzed. Our taint analyzer performs callback resolution analysis to find out the Messenger and Handler used by the Binder interface of Service and the dynamic lifecycle callbacks of Receiver so that they could be added into the dummyMain methods.

6.2 ICC Extractor

The essence of ICC extractor is to precisely extract the ICC parameters. We analyze Intent or URI to obtain detailed ICC parameters since ICC methods use them to set the target components in ICC. The ICC target components can be defined in the two ways, namely, directly set in Intent by using package names or Java.lang.class as the parameters, or set URI by using a permission string.

We can directly obtain ICC parameters by analyzing URI if the ICC methods use URI. For example, an app can use URI.parser("smsto:phone") to call the sending message API, and ContentProvider always uses URI to locate resources. URIs use strings with the special format to describe the resources. In particular, custom URIs are hard-coded in apps' code and system URIs are a limited number of common strings. Therefore, we can extract these URI prefixes via regular expression. In terms of Intent analysis, we need to deal with a serial of Intent related APIs, such as put*Extra, setData*, get*, send*Broadcast*, startActivity* that are capable of reading from, writing to, sending, or receiving Intent.

To extract the parameters, we first perform forward data flow to find the usage of ICC APIs that use Intent to send messages. Then, we do backward intra-procedural data flow analysis to find all callers of Intent. We define a model for Intent to include all the methods of Intent. For each method, we extract the corresponding parameters based on the definitions of Intent methods.

ICC can be explicit and implicit. Explicit ICC methods directly set detailed target components, while implicit ICC methods use IntentFilter to filter the Intent. Explicit ICC can be resolved based on the component setting in the Intent's parameters. For implicit ICC, we set constraints to check if the fields of IntentFilter contain the Intent's parameters.

6.3 ICC Link Analyzer

We implement Algorithm 1 according to the results of ICC Extractor. ICC Link Analyzer constructs an ICC graph and enumerates ICC links that may leak privacy. It generates an ICC link graph and identifies privacy leakage by matching

ICC links in the graph according to the analysis results and the permissions mapped to the ICC APIs. The computed ICC link graph is stored in an *MongoDB* database [8] for privacy leakage query and incremental analysis.

7 Performance Evaluation

We implement a LinkFlow prototype on Ubuntu server 14.04. We perform the experiments on a server with 4 Intel Xeon CPU 2.49 GHZ cores and 14 GB memory. We collect top 1000 popular real world apps from each of five popular apps marketplaces and repositories including Google play [5], APKPure [2], Hiapk [6], Tencent marketplace [12], and F-Droid [4]. We also collect 1500 malware from malware repositories including the MalGenome Project [56] and VirusShare [13]. Then we evaluate if there exists privacy leakage among all those apps. We remove the duplicated apps and the apps that cannot be correctly processed by FlowDroid, and the final number of benign apps tested by LinkFlow is reduced to 3014. Due to the limitation of FlowDroid [16], we set the flow taint analysis time to five minutes for each app. Our experiments show that when the taint analysis process cannot finish within five minutes, the server has run out of memory and failed to process the app.

We first investigate the privilege escalation problems in ICC links to verify that LinkFlow can identify privacy leakage by analyzing ICC links among a set of apps. Then, we use LinkFlow to analyze real world apps to identify the leaky ICC links among these apps. Finally, we measure the performance of LinkFlow and its scalability on incremental detection.

7.1 Impacts on Privilege Escalation

Android apps tend to be over-privileged especially when they use many SDKs [30]. We study the usages of permissions over 4000 apps and find only 17 components in apps are protected by permissions. Unfortunately, almost all leaky components are not protected by permissions. This means the exported leaky components can be easily exploited by malicious apps. Based on the permissions map computed by PScout [17], we investigate sensitive APIs used by these apps and observe that 1106 permission leaks among 530 apps. On average each leaky app has two permissions that could be exploited by other apps via ICC links. Therefore, to detect permissions that are susceptible to privilege escalation, we can analyze ICC links among apps and find out the exploitation paths in the ICC links.

We also study the privilege escalation problem incurred by app collusion. By analyzing the combined permissions of two linked apps and trace the data flow of their ICC links, we successfully identify 4622 suspicious data flows among those apps. The details of API usage are shown in Table 2. According to the study of Elish et al. [25], the existing dynamic taint analysis mechanisms are unable to detect privacy leakage incurred by app collusion. Based on the ICC link analysis results of all apps, LinkFlow generates a report that lists all potential ICC-based collusion among apps. It can guide app developer to avoid component misuses with leaky ICC links.

7.2 Effectiveness of Privacy Leakage Detection

LinkFlow identifies 417 benign apps with 1723 component leakages. We find 4012 abnormal flows generated by those components. We also find 113 malicious apps with 471 component leakages, and those malicious apps are inclined to use ICC at a higher frequency with 2043 abnormal flows. Those abnormal flows may be triggered by two types of attacks: *component hijacking* and *component injection*. We classify these two types of vulnerabilities based on the types of components in the InFlow and OutFlow sets. As shown in Fig. 4, the leaky paths in OutFlow set mean that these components use the ICC APIs in Table 1 to send sensitive data to other apps and may suffer component injection. The leaky paths in InFlow set means these components may suffer from component hijacking attack via *Intent* spoofing, which leads to privacy leakage.

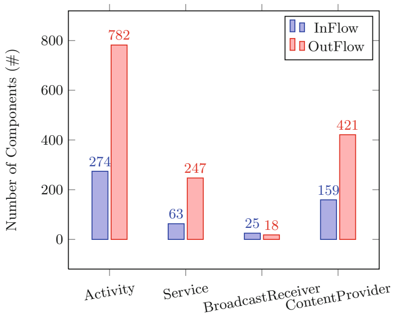


Fig. 4. The number of components included in the InFlow set and OutFlow set

We also study the privacy leakage types. LinkFlow measures the frequency of the sensitivity flows generated by benign apps and malicious apps. The mostly used source category is database information, followed by contact information, network, and location information. Also, we observe three types of sinks in benign apps: *log*, *intent*, and *storage*. In malicious apps, the top sinks are *telephone*, *storage*, *intent*, *log*, and *network*. We examine all these apps to obtain their ICC usages. The top five original sources and sinks methods we collected in the benign apps are shown in Table 2.

We observe that most ICC links are using implicit *Intent* to perform inter-apps communications. Thus, these apps' components may not be safe if they contain abnormal flows. Indeed, the unsafe usage of ICC results in vulnerabilities that can be easily exploited by malware. By performing flow analysis, we find 530 apps with abnormal flows. We verify the exploitability of all those leaky paths by using the ICC matching algorithm and find out all 530 apps have vulnerable

Table 2. The mostly used original sources/sinks

Sources type	Category	Permission	Count
Airpush: onReceive	Message Push	-	1292
ContentResolver: query	SQLite		1097
LocationManager: getLastKnownLocation	Location	ACCESS_LOCATION	861
TelephonyManager: getDeviceId	Indeifier	READ_PHONE_STATE	477
FileInputStream: read	Read File	EXTERNAL_STORAGE	319
Sinks type	Category	Permission	Count
SharedPreferences: putString	Write XML	EXTERNAL_STORAGE	1422
Log:i	Log	-	1035
OutputStream: write	IO	-	531
HttpClient: execute	Network	ACCESS_NETWORK	353
ContentResolver: insert	SQLite	-	254

components. We identify 87 apps that incur privacy leakage through ICC links and 4622 ICC paths among those apps. Typically, the sensitive data is sent to other apps and leaked via log or network.

The number of linkable apps for each app (i.e., the apps that an app can generate data flows with) varies from 20 to 56. On average, each app has three linkable apps. To evaluate the potential impacts of these links, we combine the OutFlow and InFlow sets of each app and map the API to permissions. Then we obtain pairs of linkable flows and linkable permissions that indicate that real flows between the apps and the permissions are enforced on the flows.

We manually confirm the apps incurring privacy leakage. We find that a large number of apps receive Intent messages and leak their data. In particular, most of apps (more than 80% apps) leak their data to logcat, such as Android Guard [1] and Ditty by Zya [3]. These leaky apps leaks device ID, phone numbers, contacts, locations, or SMS Messages. For instance, SMS Popup [10] writes phone numbers and messages to the system log messages that can be directly accessed via ICC.

7.3 Detection Delays of LinkFlow

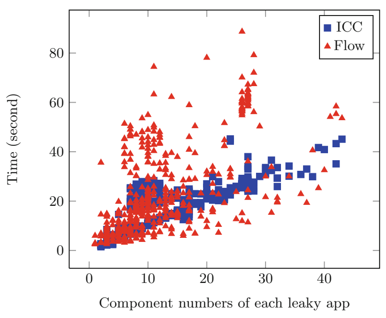
The delays of flow analysis and ICC analysis are shown in Fig. 5(a). The flow analysis delays increase with the increase of the numbers of sources/sinks numbers, and the ICC analysis delays vary according to the numbers of components. For each app, the total abnormal flows extraction delays are about two minutes. We also evaluate the delays of ICC link analyzer. On average, it takes less than 1 min, which is relatively stable. It is scalable even if the InFlow set and OutFlow sets contain millions of flows. Note that, existing tools such as IccTA [38] cannot work well on a large scale, since they rely on ApkCombiner [37] to combine the bytecode of two apps and then perform ICC analysis. Due to the limitation of FlowDroid, the code size cannot be too large. Also, they can only combine two apps at most. For the current app set with 4514 apps, the analysis time is about

848k hours ($\binom{4514}{2} * 5$ min). In contrast, it takes less than 1 min for the ICC link analysis of LinkFlow to analyze all 4514 apps.

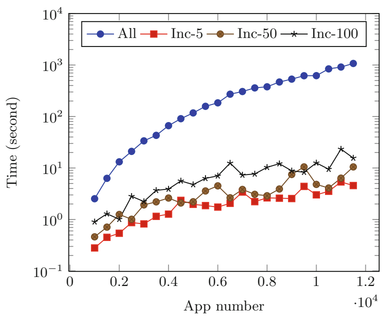
The total analysis delay of LinkFlow on analyzing 4514 apps is about 377 h, including 4514 * 5 min for flow analysis, ICC extractor, and saving to database plus 1 min for ICC link analysis. Our tool supports efficient detection on newly added apps. When a new app is submitted, IccTA needs to run over 377 h to go through the ICC links between the new app and each of the existing apps. This cost is unacceptable as everyday thousands of new apps have been developed and added. In contrast, our taint analysis only needs to run once for each app, so we only need 5 min for taint analysis and ICC extract of the new app plus 20 s for ICC link analysis.

We conduct two experiments to evaluate the scalability of LinkFlow. The first experiment is to evaluate LinkFlow with different number of apps ranging from 500 to 12000. For an app marketplace with millions of apps, there are over ten thousand leaky apps. We randomly select apps from all these 530 leaky apps and repeat 500–12000 times, and obtain different sizes of app sets that contain 500–12000 apps. As shown in Fig. 5(a), on average, LinkFlow takes less than 2 min to detect privacy leakage. Note that, since different apps may have different numbers of non-isolated components, the delays may vary even with the same number of components in the leaky app.

The second experiment is to evaluate the incremental analysis delays. We generate a large set of apps based on the real apps data. The numbers of the newly submitted apps are set to 5, 50, and 100. As shown in Fig. 5(b), when the apps number is over 10k, if we do not use the incremental analysis, the detection delays are about 15 min. However, if we use incremental analysis, it takes less than 1 min. The reason is that we only need to match flows in three flow sets, i.e., matching flow pairs that from new OutFlow set to old InFlow set, from old OutFlow set to new InFlow set, and from new OutFlow set to new InFlow set.



(a) Delays of taint analysis and ICC analysis



(b) Delays of incremental detection

Fig. 5. The detection delays of LinkFlow.

8 Discussions

ICC Analysis Across Multiple Apps. Though LinkFlow can effectively detect privacy leakage by analyzing leaky ICC links between two apps in a large set of apps, current version cannot detect leakages with leaky ICC links constructed by more than two apps. Fortunately, it can be extended to analyze ICC links among more than two apps, e.g., more than two apps collude to deliver sensitive data. Since we construct the ICC link graph for all apps, we can obtain the privacy leakage chain of multiple apps. Then, we can perform further analysis and check if these apps are colluding. For instance, app A has an ICC link delivering data to app B, while app B has an ICC link to app C, where A does not have direct links with C. LinkFlow can still check whether C can access A's data or permissions by performing taint analysis on app B and detecting if the data from A is delivered to C. To address this issue, we can leverage IccTA [38] together with ApkCombiner [37] in LinkFlow so that LinkFlow can combine bytecode of the three apps (A, B, C) and then analyze the data flows between A and C.

Apps Collusion Detection. LinkFlow can be applied to detect leakage incurred by collusion among multiple apps. However, it can only detect the collusion attacks constructed via ICC. We notice that many *e-book* apps use the same ad lib (com.waps.OffersWebView) to write sensitive data to SD Card and then transfer data to the Internet. These *e-book* apps have been granted with a large set of different permissions, such as installing app, reading SMS, reading location, and reading contacts. It is clear that these permissions are not directly used in their code but in the ad lib they used. LinkFlow cannot detect such app collusion because the leakages in these attack scenarios are not incurred by the ICC channels. Instead, they deliver sensitive data on the server and then steal them there. We consider it as a future work.

Limitation of Taint Analysis. The taint analyzer of LinkFlow is built upon FlowDroid. Due to the limitations of FlowDroid, LinkFlow may fail to analyze some apps. For instance, taint analysis may run out of memory if apps are implemented with huge bytecode or privacy leakage is constructed by native code. Moreover, current LinkFlow design does not address the class name with obfuscating strings, which is an interesting topic for our future work.

9 Related Work

Android Static Analysis. Static analysis has been extensively studied on Android for privacy leakage detection [28,33,39,41,51,54]. ComDroid [23], CHEX [41], and AppSealer [55] applied static analysis approaches to automatically evaluate component hijacking vulnerabilities of apps. FlowDroid [16] and Amandroid [52] provide context sensitive taint analysis to detect privacy leakage on Android. FlowDroid is the-state-of-art analyzer for taint analysis on Android. It is built on Soot [35] and Dexpler [18] to decompile and analyze the bytecode to detect leakage. DidFail [34] and IccTA [38] were built upon FlowDroid to detect privacy leakages of ICC. However, they cannot efficiently analyze a large set of ICC links among a huge number of apps. DroidSafe [29] identifies malicious flows by combining Android runtime analysis and static analysis. Its detection delays are ten times more than FlowDroid. LinkFlow can address this issue by reducing the number of suspicious ICC links before analyzing the leaky ICC flows.

Android Dynamic Analysis. By monitoring the states of the running processes, dynamic analysis can detect privacy leakages missed by static analysis. TaintDroid [26] implemented dynamic taint tracking for Android by modifying the Dalvik virtual machine to track sensitive data. XManDroid [20,21] monitored different communication links between apps in runtime to detect privilege escalation. API call monitoring mechanisms [44,46,50] dynamically monitored the Android system API calls to reconstruct the behavior of apps. FLEXDROID [48] provides an isolation mechanism to enforce in-app privilege separation. Blueseal [32,49] extended the existing permission mechanism by providing runtime flowing permission checking. Afonso et al. [15] performed a large-scale study on the usage of native code and generated native code sandboxing policies to limit malicious behaviors. Dynamic analysis typically is typically time-consuming, so it has the limitation to be applied to large scale leakage detection. In contrast, static taint analysis mechanisms can quickly analyze the codes of a large number of apps, so we leverage static taint analysis in LinkFlow.

Android Permission Analysis. Since Android's permission-based security mechanism cannot prevent privacy leakage from privilege escalation attack [24,27,31], researchers proposed new mechanisms to solve this problem [22,53]. FlexDroid [22] extended the Android security architecture to enforce privacy protection policies. IntentFuzzer [53] leveraged fuzzy test to generate different *Intent* messages to connect components of Android System apps. It requires modifications of the Android framework to log the actually used permissions of the components. Therefore, it can capture which permissions in these apps may be exploited by other apps. LinkFlow leveraged PScout [17] to statically map ICC APIs to the corresponding permissions. It can accurately verify potential permission leakages by checking if the permissions of the APIs are actually used by the suspicious flows.

10 Conclusion

This paper proposes LinkFlow to provide large-scale privacy leakage detection among Android apps that communicate via Inter-Component Communication (ICC). It addresses the challenge of identifying ICC-based leaky data flow among a large set of apps by only analyzing ICC links among the leaky components. LinkFlow first enumerates all leaky components of apps that may incur privacy leakage and then performs a fast ICC matching algorithm to identify all privacy leakages. We implement a LinkFlow prototype and evaluate our tool over 5000 apps and find out 530 leaky apps. Among these leaky apps, we discover 4622 ICC links among 87 apps that may lead to severe data leakages.

Acknowledgments. The research is partially supported by the National Natural Science Foundation of China under Grant 61572278, the National Key Research and Development Program of China under Grant 2016YFB0800102, and U.S. Office of Naval Research under Grant N00014-16-1-3214 and N00014-16-1-3216.

References

1. Android Guard: http://android.app.qq.com/myapp/detail.htm?apkName=org.androidbeans.guard
2. APKPure. https://apkpure.com/
3. Ditty by Zya. https://play.google.com/store/apps/details?id=com.zya.ditty
4. F-Droid. https://f-droid.org/
5. Google Play. https://play.google.com
6. Hiapk. www.hiapk.com/
7. Intents and intent filters. http://developer.android.com/guide/components/intents-filters.html
8. MongoDB. https://www.mongodb.org/
9. A part of ICC APIs, the defination of Intent. https://developer.android.com/reference/android/content/Intent.html
10. SMS Popup. https://play.google.com/store/apps/details?id=net.everythingandroid.smspopup
11. SMSZombie. http://blog.trustgo.com/SMSZombie/
12. Tencent Markletplace. http://sj.qq.com/myapp/
13. VirusShare. https://virusshare.com/
14. Vulnerability of Dropbox SDK. http://www.slideshare.net/ibmsecurity/remote-exploitation-of-the-dropbox-sdk-for-android
15. Afonso, V., Bianchi, A., Fratantonio, Y., Doupé, A., Polino, M., de Geus, P., Kruegel, C., Vigna, G.: Going native: using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: NDSS (2016)
16. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI, vol. 49, no. 6, pp. 259–269 (2014)
17. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the android permission specification. In: CCS, pp. 217–228 (2012)
18. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In: SOAP, pp. 27–38 (2012)
19. Bartel, A., Klein, J., Monperrus, M., Le Traon, Y.: Static analysis for extracting permission checks of a large scale framework: the challenges and solutions for analyzing Android. TSE **40**(6), 617–632 (2014)
20. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R.: Xmandroid: a new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
21. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: NDSS (2012)
22. Bugiel, S., Heuser, S., Sadeghi, A.-R.: Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In: USENIX Security, pp. 131–146 (2013)
23. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys, pp. 239–252 (2011)

24. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18178-8_30
25. Elish, K.O., Yao, D., Ryder, B.G.: On the need of precise inter-app ICC classification for detecting android malware collusions. In: MoST (2015)
26. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI (2011)
27. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: USENIX Security, vol. 30 (2011)
28. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: automated security certification of android. Technical report, University of Maryland (2009)
29. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in DroidSafe. In: NDSS (2015)
30. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.-R.: Unsafe exposure analysis of mobile in-app advertisements. In: WISEC, pp. 101–112 (2012)
31. Grace, M.C., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: NDSS (2012)
32. Holavanalli, S., Manuel, D., Nanjundaswamy, V., Rosenberg, B., Shen, F., Ko, S.Y., Ziarek, L.: Flow permissions for android. In: ASE, pp. 652–657 (2013)
33. Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: static analyzer for detecting privacy leaks in android applications. In: MoST 12 (2012)
34. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: SOAP, pp. 1–6 (2014)
35. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for Java program analysis: a retrospective. In: CETUS 2011 (2011)
36. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36579-6_12
37. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L.: ApkCombiner: combining multiple android apps to support inter-app analysis. In: Federrath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 513–527. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_34
38. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: IccTA: detecting inter-component privacy leaks in android apps. In: ICSE, pp. 280–291 (2015)
39. Li, L., Bartel, A., Klein, J., Le Traon, Y.: Detecting privacy leaks in android apps. In: ESSoS-DS (2014)
40. Li, L., Bartel, A., Klein, J., Le Traon, Y.: Automatically exploiting potential component leaks in android applications. In: TrustCom, pp. 388–397 (2014)
41. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: statically vetting android apps for component hijacking vulnerabilities. In: CCS, pp. 229–240 (2012)
42. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: ACSAC, pp. 51–60 (2012)
43. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS (2014)
44. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: EuroSec, April 2013

45. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61. ACM (1995)
46. Sakamoto, S., Okuda, K., Nakatsuka, R., Yamauchi, T.: DroidTrack: tracking and visualizing information diffusion for preventing information leakage on android. JISIS **4**(2), 55–69 (2014)
47. Schlegel, R., Zhang, K., Zhou, X.-Y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: NDSS, vol. 11, pp. 17–33 (2011)
48. Seo, J., Kim, D., Cho, D., Kim, T., Shin, I.: FLEXDROID: enforcing in-app privilege separation in android. In: NDSS (2016)
49. Shen, F., Vishnubhotla, N., Todarka, C., Arora, M., Dhandapani, B., Lehner, E.J., Ko, S.Y., Ziarek, L.: Information flows as a permission mechanism. In: ASE, pp. 515–526 (2014)
50. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of android malware behaviors. In: NDSS (2015)
51. Tripp, O., Rubin, J.: A bayesian approach to privacy enforcement in smartphones. In: USENIX Security, pp. 175–190 (2014)
52. Wei, F., Roy, S., Ou, X., et al.: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: CCS, pp. 1329–1341. ACM (2014)
53. Yang, K., Zhuge, J., Wang, Y., Zhou, L., Duan, H.: IntentFuzzer: detecting capability leaks of android applications. In: ASIACCS, pp. 531–536 (2014)
54. Yang, Z., Yang, M.: Leakminer: detect information leakage on android with static taint analysis. In: WCSE, pp. 101–104 (2012)
55. Zhang, M., Yin, H.: AppSealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In: NDSS (2014)
56. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy, pp. 95–109 (2012)