

# Android Inter-App Communication Threats and Detection Techniques

Shweta Bhandari<sup>a,\*</sup>, Wafa Ben Jaballah<sup>b</sup>, Vineeta Jain<sup>a</sup>, Vijay Laxmi<sup>a</sup>, Akka Zemhari<sup>c</sup>, Manoj Singh Gaur<sup>a</sup>, Mohamed Mosbah<sup>c</sup>, Mauro Conti<sup>d</sup>

<sup>a</sup>*Malaviya National Institute of Technology Jaipur (MNIT Jaipur)*

<sup>b</sup>*Orange Labs, Paris, France*

<sup>c</sup>*LaBRI - University of Bordeaux, CNRS, 33405 Talence cedex, FRANCE*

<sup>d</sup>*University of Padua, Italy*

---

## Abstract

With the digital breakthrough, smart phones have become very essential component for many routine tasks like shopping, paying bills, transferring money, instant messaging, emails etc. Mobile devices are very attractive attack surface for cyber thieves as they hold personal details (accounts, locations, contacts, photos) and have potential capabilities for eavesdropping (with cameras/microphone, wireless connections). Android, being the most popular, is the target of malicious hackers who are trying to use Android app as a tool to break into and control device. Android malware authors use many anti-analysis techniques to hide from analysis tools. Academic researchers and commercial anti-malware companies are putting great effort to detect such malicious apps. They are making use of the combinations of static, dynamic and behavior based analysis techniques.

Despite of all the security mechanisms provided by Android, apps can carry out malicious actions through inter-app communication. One such inter-app communication threats is collusion. In collusion malicious functionality is divided across multiple apps. Each participating app accomplish its part and communicate information to another app through Inter Component Communication (ICC). ICC does not require any special permissions. Also there is no

---

\*Corresponding author. Mobile: +91-7597385348

Email address: [er.shwetabhandari@gmail.com](mailto:er.shwetabhandari@gmail.com) (Shweta Bhandari)

compulsion to inform user about the communication. Each participating app needs to request a minimal set of privileges, which may make it appear benign to current state-of-the-art techniques that analyze one app at a time.

There are many surveys on app analysis techniques in Android; however they focus on single-app analysis. This survey highlights several inter-app communication threats, in particular collusion among multiple-apps. In this paper, we present Android vulnerabilities that may be exploited for carrying privilege escalation attacks, privacy leakage and collusion attacks. We cover the existing threat analysis, scenarios, and a detailed comparison of tools for intra and inter-app analysis. To the best of our knowledge this is the first survey on inter-app communication threats, app collusion and state-of-the-art detection tools in Android.

*Keywords:* App Collusion, Privacy Leakage, Inter Component Communication, Inter App Communication, Multi App Analysis

---

## 1. Introduction

Nowadays, mobile devices such as smartphones, are widely used for social networking, online shopping, banking, etc. Mobile applications are increasingly playing an essential role in our daily life, making the safety guards in mobile operating systems an important concern for researchers and practitioners. Android is the most popular mobile operating system, with 84% of the worldwide smartphone sales to end users in first quarter of 2016 [1], and over 50 billion app downloads so far. The large popularity of Android and its open nature made it a primary target of hackers who are now developing malicious apps at an industrial scale [2–9].

An Android app consists of components and uses a special interaction mechanism to perform Inter-Component Communication (ICC). ICC enables modular design and reuse of functionality across apps and app components. In Android, ICC communication model is implemented as a message-passing system, where messages are encapsulated as Intent objects. Through Intents, an

app (or app component) can utilize functionality exposed by another app (or app component), e.g. by passing a message to the browser to render content or to a navigation app to display a location and provide directions to it. This light communication model has been used by developers to design rich application scenarios by reusing existing functionality. Unfortunately, because many Android developers have limited expertise in security, the ICC mechanism has brought a number of vulnerabilities [2, 3, 10–13]. Some of the ICC vulnerabilities viz. Activity hijacking vulnerability (where a malicious Activity is launched in place of the intended Activity), Intent spoofing vulnerability (where a malicious app sends Intents to an exported component which originally does not expect Intents from that app) etc.

Different research efforts have investigated weaknesses from various perspectives [14–20], including detection of information leaks, analysis of the least-privilege principle, and enhancements to Android protection mechanisms. Despite the significant progress, such security techniques are substantially intended to detect and mitigate vulnerabilities in a single app [10, 21–24], but fail to identify vulnerabilities that arise due to the interaction of multiple apps. Vulnerabilities due to the interaction of multiple apps, such as collusion attacks and privilege escalation chaining, cannot be detected by techniques that analyze a single app in isolation. Thus, there is a pressing need for security analysis techniques in such rapidly growing domains to take into account such communication vulnerabilities.

The principle of malware collusion has been recently described in a few research papers [11, 17–19, 25–30] as the next step that malware writers may evolve into. Collusion refers to the scenario where two or more applications possibly (not necessary) developed by the same developer, interact with each other to perform malicious tasks. The danger of malware collusion is that each colluding malware only needs to request a minimal set of privileges, which may make it appear benign under single-app analysis mechanisms [4, 17, 18, 30]. The scenario could be think of as two utility apps one for cab booking and another is a browser app. Now cab booking app needs to access client’s location and browser

app needs to connect with the internet. Lets assume that both the apps are developed by same adversary and he intentionally puts a communication channel between these two apps. Whenever user invokes cab booking app, along with serving to the user it also sends location information to the browser app. Since browser have the access to internet, it can easily send the location information of the user to any command and control (C&C) server. Malware writers have strong incentives to write colluding malware.

The wide usage of ICC calls in benign app pairs make accurate classification quite challenging [4, 18, 31]. Academia and industry researchers have proposed solutions and frameworks to analyze, and detect the collusion attacks [3, 14, 15, 17–20, 26, 30, 32, 33]. Some of these are even available as open-source as [3, 15, 18, 33]. The solutions can be characterized using three broad types of analysis: Static analysis, dynamic analysis and policy enforcement based analysis.

In [18], authors propose a tool named DIALDroid, as the most recent state-of-the-art inter-app ICC analysis tool for large scale detection of collusion and privilege escalation. They also provide the first inter-app collusion real-apps benchmark of 30 apps. Till now, this is the most efficient tool available in the literature for inter-app vulnerability detection. MR-Droid [17] aims to detect inter-app communication threats specifically intent hijacking, intent spoofing and collusion. It proposes a MapReduce based framework to scale up compositional app analysis. DidFail [33] is another state-of-the art to detect intra-component and inter-component information flow in a set of apps. In [26], authors propose XMandroid, that is the first approach for detecting collusion attacks in Android platforms. It claims to identify privilege escalation in case of pending intents and transmission channels between dynamically built components such as broadcast receivers. FUSE [14] is a tool that starts by single-app static analysis accompanied with lint tool to mitigate limitations of static analysis followed by multi-app information flow analysis. IccTA [15] is a static taint analyzer to detect privacy leaks between components in Android apps. If combined with APKCombiner [3], it can also detect inter-app leakage paths.

This survey paper aims to present a general review about inter-app communication threats in particular, collusion attacks in Android framework. It provides a better understanding of the key research challenges. We present an abstract definition of collusion and highlight its origin. Along the way, we cover the Android model, the communication and permission model of Android and the main vulnerabilities that lead to a possible collusion attack. We also cover the existing threat analysis and a detailed comparison of techniques for intra and inter-app analysis. This review gives an insight into the strengths and shortcomings of the known tools and provides a clear comparison for the researchers between these tools. Finally, we present an insight into our future research directions.

This survey paper is organized as follows. Section 2 presents Android model. In Section 3, we present the Inter Process Communication (IPC) model as one of the key features of programming model in Android. Then, in Section 4, we present Android security risks. In Section 5, we elaborate collusion by providing a formal definition and cases where collusion attack is possible followed by the main challenges to detect collusion attack. In Section 6, we review the inter application analysis. Section 7 recalls state-of-art approaches, a thorough comparison between them for collusion detection and lessons learned. In Section 8, we conclude the paper and we present an insight into our future research directions.

## **2. Android**

Android is developed under the Android Open Source Project (AOSP), promoted by the Open Handset Alliance (OHA) and maintained by Google [34]. Android is developed on top of Linux kernel due to its robust driver model, efficient memory, process management, and networking support for the core services. Linux Kernel is customized specifically for the embedded environment consisting of limited resources.

Android apps are written in java; however, the native code and shared li-

braries are developed in C/C++ to support high performance [35]. There are two runtime environments available in Android viz. Dalvik Virtual Machine (DVM) and Android Runtime (ART). In DVM, dex file of the Android apps are translated to their respective native representations on demand using just-in-time (JIT) compiler. However, in case of ART, ahead-of-time (AOT) compilation is performed i.e. at the time of installation itself apps are compiled to a ready-to-run state [36]. Therefore, ART massively improves the performance and battery life of Android device.

Once the OS boot completes, a process known as *zygote* (parent of all apps) initializes. As *zygote* starts, it preloads all necessary Java classes and resources, starts System Server and opens a socket `/dev/socket/zygote` to listen for requests for starting applications. Thus *zygote* process expedites the app launching process.

In the following, we present the main Android app composition followed by Android security model viz. application signing, application permission, and sandboxed environment.

### *2.1. Android App Composition*

Android applications are distributed as binaries in a regular format based on zip files with `.apk` as file extension. It usually contains the following files and directories [37].

1. **Manifest file:** Manifest file is an XML configuration file (`AndroidManifest.xml`) one per app. It is used to declare various components of an application, their encapsulation (public or private) and the permissions required by the app. Android APIs offer programmatic access to mobile device-specific features such as the GPS, vibrator, address book, data connection, calling, SMS, camera, etc. These APIs are usually protected by permissions. For example the `Vibrator` class, to use the `android.os.Vibrator.vibrate(long milliseconds)` function, which starts the phone vibrator for a number of milliseconds. The permis-

sion `android.permission.VIBRATE` must be declared in the app manifest file.

2. **dex file:** A Dalvik executable (classes.dex), which contains the bytecode of the program.
3. **res directory:** Resources including string literals, their translations, and references to binary resources.
4. **layout directory:** XML layouts describing user interface elements.
5. **lib directory:** The directory containing the compiled code that is specific to a software layer of a processor.
6. **assets directory:** The directory containing applications assets, which can be retrieved by AssetManager.

An android app is composed of any combination of the following four components:

- **Activities:** The Android libraries include a set of GUI components specifically built for the interfaces of mobile devices, which have small screens and low power consumption. One type of such component is Activities, that represent screens which are visible to the user;
- **Services:** They perform background computation;
- **Content Providers:** They act as database-like data stores;
- **Broadcast Receivers:** They handle notifications sent to multiple targets.

## *2.2. Android Security Model*

Android security depends on restricting apps by combining app signing, sandboxing, and permissions.

### *2.2.1. App Signing*

App signing is a prerequisite for inclusion in the official Android market (Google Play Store). App signature is the point of trust between Google and the third party developers to ensure app integrity and the developer reputation.

Most developers use self-signed certificates that they can generate themselves, which do not imply any validation of the identity of the developer. Instead, they enable seamless updates to applications and enable data reuse among sibling apps created by the same developer [38].

### 2.2.2. App Permission

App permission model regulates how applications access certain sensitive resources, such as users' personal information or sensor data (e.g., camera, GPS, etc.). For instance, an application must have the *READ\_CONTACTS* permission in order to read entries in a user's phone [39]. System permissions are divided into four protection levels. The two most relevant levels to this manuscript are normal and dangerous permissions. Normal permissions require when the app needs to access data or resources outside the app's sandbox, but involves very little risk to the user's privacy or the operation of other apps. For example, permission to set the alarm is a normal permission. Dangerous permissions are required when the app wants data or resources that involve user's private information or could potentially affect user's stored data or the operation of other apps. For example, the ability to read user's contacts is a dangerous permission [40]. Applications can also define their own permissions in order to restrict the use of components in an application that can perform sensitive tasks. The third level of permission is Signature permission that is used by the developers to transfer resources and data between their own applications meanwhile safeguarding them against applications of other developers [41]. Lastly, SignatureOrSystem permission which is high-level permission that includes changing security settings, installing an application, etc. These permissions are maintained by OS developers and manufacturers. They are granted by System to those applications which are either contained in the system image or signed by the same certificate as of system image [42].

App permissions play important role in malware detection. There exist rich literature embodies tools that attempt to identify malicious applications through their permission requests [12, 43, 44]. Researchers have also developed



static and dynamic analysis tools to analyze Android permission specifications [41, 42, 45]. Permission enforcement techniques are also proposed [39].

### *2.2.3. Sandboxed Environment*

Android apps are executed in a sandboxed environment to protect the system, the user data, the developer apps, the device, the network, and the hosted applications, from malware [41]. Each app process is protected with an assigned unique id (UID) within an isolated sandbox. The sandboxing restrains other apps or their system services from interfering the app [46]. Android protects network access by implementing a feature Paranoid Network Security, a feature to control Wi-Fi, Bluetooth and Internet access within the groups. If an app has permission for a network resource (e.g., Bluetooth), the app process is assigned to the corresponding network access id. Thus, apart from UID, a process may be assigned one or more group id (GIDs). An app must contain a PKI certificate signed with the developer key. App signing procedure places an app into an isolated sandbox assigning it an unique UID. If the certificate of an app A matches with an already installed app B on the device, Android assigns the same UID (i.e., sandbox) to apps A and B, permitting them to share their private files and the manifest defined permissions. This unintended sharing can be exploited by the malware writers as naive developers may generate two certificates. It is advisable for the developers to keep their certificates private to avoid their misuse. The Android sandbox relies on, and augments, the Linux kernel isolation facilities. While sandboxing is a central security feature, it comes at the expense of interoperability. In many common situations, apps require the ability to interact. For example, the browser app should be capable of launching the Google Play app if the user points toward the Google Play website [47].

## **3. Inter-Component Communication**

Inter Process Communication (IPC) is known as Inter Component Communication (ICC) in Android [48]. It is the key features of Android programming model. It allows a component of an application to access user's data and can

transfer it to another component of same or other application within the same device, or to an external server. ICC helps to eliminate duplication of functionality in different applications. Developers can leverage data and services provided by other applications. For example, a cab booking application can ask Google Maps for client's or driver's location information. This communication between applications can reduce developer's burden and facilitate functionality reuse.

To support inter-component communication, there exists conventional methods called *overt channels* and non-conventional methods called *covert channels*. Covert channels are intentionally used to hide the messages or communication. Any app using covert channel as a medium of communication can be suspected as malicious. Although overt channels are perfectly benign and widely used for communication in Android apps. The main focus of this paper is to show that how a set of apps appear perfectly benign can carry out a threat. Therefore in this section, we will elaborate overt channels and provide glimpse of covert channels.

### 3.1. Overt channel

*Overt channel* is an unconcealed medium provided by Android framework for communication. In the following, we present the main channels that come under this such as Intents, Content Providers, Shared Preferences, External Storage, and Remote Method Calls.

#### 3.1.1. Intents

Intents enable components of an application to invoke other components of the same or different applications. It is also used to pass data between different components through Bundles. It optionally contains destination component name or action string, category and data. Intents are the preferred message passing mechanism for asynchronous IPC in Android. The Android API defines methods called ICC methods that can accept intents and perform actions accordingly. For example, `startActivity(Intent)`, `startService(Intent)` etc.

ICC is widely facilitated through Intents. In [15], authors highlighted that 2955/33258 applications use ICC through intents. Intents can be sent to three out of four components. Based on destination of ICC calls, they are categorized into two broad categories:

#### *Implicit Intent*

Implicit intents are used when the receiver of the intent is not fixed [15]. Whenever an app wants to send the intent to all the registered components (registration is done using an intent filter in the manifest file) within and across the installed apps. When an app invokes API call with implicit intent then depending on the type of component, framework serves the calls.

- If the calls are intended for activity, then users are asked for the choice.
- If the calls are intended for service, then the framework will randomly choose one of the registered services.
- If the calls are intended for broadcast receivers, then the framework delivers to all the receivers.

In the following, we present a sample code of implicit intent where "com.example.msgSendFirst" is the action string:

---

```
/**
 * Implicit Intent
 */
Intent intent = new Intent("com.exampke.msgSendFirst");
startActivity(intent);
```

---

#### *Explicit Intent*

Explicit intents are used when receiver of the intent is fixed. When an app invokes API call with explicit intent, the framework will deliver the intent to the component that is mentioned in the intent. A Sample code of explicit

intent where "this, LoginActivity.class" is the address of the destination component:

---

```
/**
 * Explicit Intent
 */
Intent intent = new Intent(this,LoginActivity.class);
startActivity(intent);
```

---

### 3.1.2. Content Provider

Content Providers are used to transfer structured data across components of same or different apps. It stores information in tables like relational databases. To access or modify data in Content Provider, apps need `ContentResolver` objects. An app can also attach read and write permissions to the content provider it owns.

---

```
public class SchoolProvider extends ContentProvider {
    /**
     * Declaration
     */
    static final String PROVIDER_NAME = "com.example.provider.School";
    static final String URL = "content://" + PROVIDER_NAME + "/students";
    static final Uri CONTENT_URI = Uri.parse(URL);
    .....
    .....

    /**
     * Database specific constant declarations
     */
    private SQLiteDatabase db;
    static final String DATABASE_NAME = "School";
    static final String STUDENTS_TABLE_NAME = "students";
    .....
```

```

.....

/**
 * Helper class that actually creates and manages
 * the provider's underlying data repository.
 */
private static class DatabaseHelper extends SQLiteOpenHelper {
    DatabaseHelper(Context context){
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db)
    {
        db.execSQL(CREATE_DB_TABLE);
    }

    public boolean onCreate() {
        /**
         * Create a write able database which will trigger its
         * creation if it doesn't already exist.
         */
    }

    public Uri insert(Uri uri, ContentValues values) {
        /**
         * Add a new student record
         */
    }

    public Cursor query(Uri uri, String[] projection, String
        selection,String[] selectionArgs, String sortOrder) {
        /**
         * Code for querying the database
         */
    }
}

```

```

public int delete(Uri uri, String selection, String[]
    selectionArgs) {
    /**
     * Code for deleting records from the database
     */
    public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
        /**
         * Code for updating records from the database
         */
    }
}

```

---

### 3.1.3. Shared Preference

Shared Preference is an operating system feature that allows apps to store key-value pairs of data. Its purpose is to be used to store preferences information. Apps can use key-value pairs to exchange information if proper permissions are defined when accessing and storing data.

---

```

/**
 * Declaration
 */
SharedPreferences sharedPref = getActivity().
    getPreferences(Context.MODE_PRIVATE);
/**
 * Write to Shared Preferences
 */
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.
    saved_high_score), newHighScore);
editor.commit();
/**
 * Read from Shared Preferences
 */

```

```

int defaultValue = getResources().getInteger
(R.string.saved_high_score_default);
long highScore = sharedPref.getInt
(getString(R.string.saved_high_score),
defaultValue);

```

---

#### 3.1.4. External Storage

External Storage is the storage space external to an app. It includes USB connection, SD card and even non-removable storage. Apps accessing the external storage need to declare the `READ_EXTERNAL_STORAGE` permission. Apps declaring the `WRITE_EXTERNAL_STORAGE` can write and read from external storage.

---

```

/**
 * Write to SD card
 */
File myFile = new File("/sdcard/mysdfile.txt");
myFile.createNewFile();
FileOutputStream fOut = new FileOutputStream(myFile);
OutputStreamWriter myOutWriter = new OutputStreamWriter(fOut);
myOutWriter.append(txtData.getText());
myOutWriter.close();
fOut.close();

/**
 * Read from SD Card
 */
File myFile = new File("/sdcard/mysdfile.txt");
FileInputStream fIn = new FileInputStream(myFile);
BufferedReader myReader = new BufferedReader(
    new InputStreamReader(fIn));
myReader.close();

```

---

### 3.1.5. Remote Method Calls

Remote methods enable to make method calls that look local but are executed in another process. It is same as remote procedural calls (RPC) in other systems. In Android, due to sandboxing, one process cannot access the memory of another process. In order to communicate, they need to pack their objects into the primitives that operating system can understand and unpack them again at receiver's end. This is facilitated by Binder, Messenger and AIDL.

- *Binder*: Binder supports remote method calls within same application without the support of multi-threading. Binders are the entity which allows activities and services to obtain a reference to another service. It allows not simply send messages to services but directly invoke methods on them. Binder class provides direct access to public methods in the service but can be used only when the service is used by the local application and in the same process. For example, it would work if a music application that needs to bind an activity to its own service that's playing music in the background.
- *Messenger*: Messenger supports remote method calls across the applications without the support of multi-threading. It represents a reference to a Handler that can be sent to a remote process via an Intent. Messenger provides an interface to the service to communicate with remote processes. This allows inter-process communication without the use of AIDL. This can be used in the case where remote IPC is required but multi-threading support by the service is not required.
- *AIDL*: AIDL supports remote method calls across the applications with the support of multi-threading. Android provides an API to handle marshalling and unmarshalling of objects called Android's Interface Definition Language (AIDL). AIDL is necessary only if remote access of the service is required for IPC and want to handle multithreading in that service. AIDL is complex to implement as this interface sends simultaneous requests to the service, which must then handle multi-threading.



### *3.2. Covert channel*

*Covert channel* is a secret medium which exploits shared resources and use them for communication [26]. Timing channels and storage channels come under the covert channel.

#### *3.2.1. Timing channels*

In timing channel, the information between applications is synchronously transmitted using shared resource having no storage capability. Battery use, Phone call frequency are examples of timing channels.

#### *3.2.2. Storage channels*

In storage channel, the information between applications is asynchronously transmitted using shared resource with storage. Phone call logs, Content providers are examples of storage channels.

## **4. Android Security Risks and Consequences**

Android ensures security through its sandbox model, application signing and the permission model for managing IPC effectively and efficiently. In spite of these measures, Android is vulnerable to many security risks. According to the recent OWASP mobile security report [49], out of 91 reported security risks, 85 are recorded to be present in Android. This makes Android security a serious concern. These risks are outcome of either maliciously exploiting the legitimate procedures provided by android such as ICC, or taking advantage of unchecked processes occurring in the system. In the following, we focus on Intent based attacks and their consequences.

### *4.1. Intent based attacks*

We focus our attention on the security challenges of Android communication from the perspectives of Intent sending and receiving. In section 4.1.1, we focus on the Intent receiving, and consider vulnerabilities related to receiving Intents coming from other applications. In Section 4.1.2, we consider how sending Intents to the wrong application can leak user information.

#### 4.1.1. Intent Spoofing

Intent spoofing refers to a typical scenario where a vulnerable app has a component that expects Intent from Android framework or itself. If the component is exposed, then other malicious apps can send forged Intents, and then spoof this app in order to trigger misbehaved actions. In the following, we classify the Intent spoofing to three subclasses [10]: malicious broadcast injection, malicious activity launch, and malicious service launch.

*Malicious Broadcast Injection.* Broadcast receivers are vulnerable to malicious broadcast injection when they receive Intents with system actions [10]. An example scenario of this attack is that some Intents contain action strings that only the operating system may add to broadcast Intents. If a malicious application sends an Intent explicitly addressed to the target Receiver, without containing the system action string. The Receiver will be tricked into performing functionality that only the system should be able to trigger if it does not check the Intent's action.

*Malicious Activity Launch.* A malicious activity launch is executed by other applications with explicit or implicit Intents. The impact of this attack is that the malicious Activity's UI will load instead of the targeted one. In [10], the authors classify three types of possible attacks when launching malicious activity:

- Modification of data in the background caused due to non verification of the origin of the Intent or leading to a change in the application state;
- A user can be misleded between malicious and victim applications. She might make changes to the victim application while believing she is interacting with the malicious one.
- A victim application could leak some sensitive information returning a result to its caller upon completion.

*Malicious Service Launch.* In the same way as an Activity, a Service, if not protected with permissions, any application can bind it. This vulnerability

could lead even to leak information or perform unauthorized tasks, depending on the type of Service [10].

#### 4.1.2. Intent Hijacking

The Intent hijacking threat is illustrated when an Intent could not reach the intended recipient via an implicit ICC, and then it may be hijacked by an unauthorized app [10]. We classify this threat based on the type of the sending component: broadcast receivers hijacking, activity hijacking, and service hijacking.

*Broadcast Receivers Hijacking.* Broadcast Receivers can be vulnerable to active denial of service attacks or eavesdropping. An eavesdropper can read the contents of a broadcast Intent without interrupting the broadcast. This is a risk whenever an application sends a public broadcast. A malicious Broadcast Receiver could eavesdrop on all public broadcasts from all applications by creating an Intent filter that lists all possible actions, data, and categories [10].

*Activity Hijacking.* In an Activity hijacking attack, a malicious Activity is launched instead of the intended Activity [6]. This attack works as follows: The malicious Activity registers to receive another applications implicit Intents, and it is then started instead of the expected Activity. The impact of this attack is dreadful since the malicious activity could read the data in the Intent and then immediately relay it to a legitimate activity [50] into the victim application.

*Service Hijacking.* The service hijacking attack occurs when a malicious service intercepts an Intent designed for a legitimate service [51]. The impact of this attack is that the initiating application establishes a connection with a malicious service instead of the legitimate one. The malicious service can steal data and lie about completing requested actions [52].

#### 4.2. Android Risk Consequences

In the following, we focus on two major consequences of the Android security risks: privilege escalation, and privacy leaks.

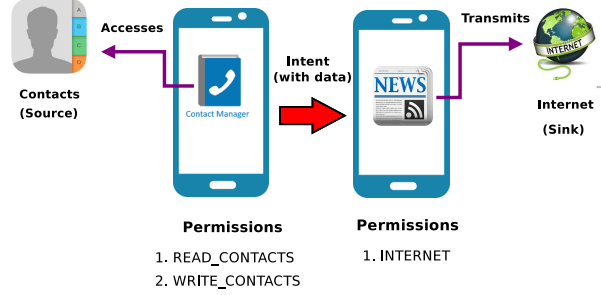


Figure 1: Privileged Escalation attack scenario

#### 4.2.1. Privilege Escalation

Android security framework enforces permission protected model that allows the user to regulate the access of data by an application. It has been shown that applications can bypass this security model by exploiting transitive permission usage known as privilege escalation [53], [28], [26], [54]. This refers to the scenario where two or more applications with a limited set of permissions communicate with each other to gain indirect privilege escalation and can perform unauthorized actions.

Figure 1 shows an example of privilege escalation attack. In this example, an android device contain 2 apps - **Contacts manager** and **News**. Contact manager includes permissions **READ\_CONTACTS** and **WRITE\_CONTACTS**. On the other hand, News app includes permission **INTERNET**. Contact manager cannot access internet and news app cannot access contacts stored on the device. However, they can communicate via Intents. Contact manager sends an Intent carrying contact information as a payload to news app. As news app can access internet, it can transmit it to outside world. Hence, news app can access contacts, even if it does not contain permission to do so. Hence its privileges got escalated.

#### 4.2.2. Privacy Leaks

Privacy leak occurs if there is a secret (without user consent) path from sensitive data as source to statements sending this data outside the application or device, called sink. This path may be within a single component or across

multiple components. Thus, analyzing components separately is not enough to detect leaks. It is necessary to perform an inter-component analysis of applications. Android app analysts could leverage such tools to identify malicious apps that leak private data. For the tool to be useful, it has to be highly precise and minimize the false positive rate when reporting applications leaking private data. For example, IccTA, an inter-component communication Taint Analysis tool [15]. It is for a sound and precise detection of ICC links and leaks.

Recent works have demonstrated that Android apps exhibit different privacy leaks, that are mainly build around the collusion attack [55–58]. The main vulnerability comes from the fact that these leaks are exacerbated by several applications that can interact to leak data using the inter-app communication mechanism [2]. The aforementioned security risk could lead to the collusion attack resulting in privacy abuse. Through the inter-app ICCs, two or more apps can collude to perform malicious actions. We give more details about the collusion attacks in Section 5.

## 5. Collusion

The Android security model is designed to protect data, applications and devices from security threats. It is guarding apps by combining app signing, sandboxing, and permissions. Unfortunately, these restrictions can be bypassed by colluding apps. The combined permission of these apps allow them to carry out attack, that could not be possible by a single app. Let us consider the following example where a collusion consists of one app permitted to access some personal data, and this app passes the data to a second app that is allowed to transmit the data. Moreover, the Android OS does not check if an app that is accessing a permission-protected resource through another app has itself requested that permission. We believe that collusion is worth investigation since it could be exploited by criminals, and become a serious threat in the near

future.

In this section, we start with a brief history of the origin of collusion. Then we define collusion along with its categories on the grounds of application properties. We also highlight some challenges faced in detecting collusion. For the sake of clarity, we define some terminologies such as *Sensitive Resource Access*: When an Android app access the system resource that is protected with some dangerous permission then that access to the resource is called Sensitive Resource Access. *Sensitive Information*: Any piece of data that generates from sensitive resource access becomes Sensitive Information. *Leakage*: It happens when the sensitive information moves out of device boundaries without user consent.

### 5.1. History

The problem of colluding apps can be traced back to confused deputy attack. This attack was first reported in 1988 by Norm Hardy [59]. This attack can happen when an application provides a public interface and access some sensitive resources. Other applications could use that interface to access the sensitive resources. The application providing access to the sensitive resource is called a confused deputy.

In 2011, the work in [43] mapped confused deputy attack with permission re-delegation attack. In particular, a careless developer may expose permission-protected resources through exported component. Other applications can access those resources through ICC to that component.

In 2011, the first documented example of intentional permission re-delegation was presented by [11]. They developed Soundcomber, a Trojan with few and innocuous permissions, that can extract a small amount of targeted private information from the audio sensor of the phone and conveys information remotely without direct network access. This is the example of intentional permission re-delegation and illustrated in Figure 2.

The Soundcomber example shows the difference between app collusion and confused deputy attacks. In app collusion the exposure of the sensitive resource

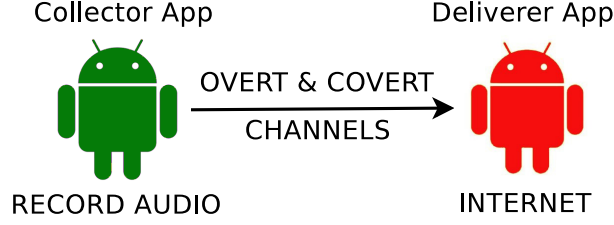


Figure 2: Soundcomber working architecture

is intentional [53].

### 5.2. Definition

Assume,  $A$  be a set of all Android apps and  $P$  be a set of all possible dangerous permissions in Android.

Let,  $a$  and  $b$  are two apps with permission set  $p_a$  and  $p_b$  respectively such that:

$$a, b \in A \text{ and } p_a, p_b \subset P,$$

Suppose,  $a$  performs sensitive resource access that requires permission  $\alpha$  to generate sensitive information  $\iota$  such that,

$$\alpha \in (p_a - p_b),$$

If  $f(\iota)$  (perform any operation(s) on  $\iota$ ) flows to  $b$  through any number of apps, and  $b$  performs sensitive resource access that requires permission  $\beta$  to leak  $f(\iota)$  such that,

$$\beta \in p_b,$$

**Then we say that  $a$  and  $b$  are colluding apps.**

### 5.3. Scenarios

For the sake of completeness, we illustrate the collusion definition by three different scenarios based on app properties: 1) Among colluding apps, all are signed by the same signature; 2) Among colluding apps, all are signed with different signatures; 3) dynamic colluding apps.

#### *5.3.1. Among colluding apps, all are signed by same signature*

Android requires that all apps have to be signed with developer's certificate before they can be installed. Android uses this certificate to identify the developer of an app [38]. There are some signing considerations as:

- App Modularity: Android allows apps signed by the same certificate to run in the same address space, if applications choose this, the system treats them as a single application.
- Code/Data Sharing: Android provides signature-based permissions enforcement, so that an app can expose functionality to another app that is signed with a specified certificate.

In this scenario, apps request a minimal permission set at the time of installation; and they can expand it later with the permissions of other apps with same signature. The shared permission set exposes sensitive information leakage path and thus emanate collusion.

#### *5.3.2. Among colluding apps, all are signed with different signatures*

Android applications signed by different signatures always run in different user address space. Such apps are not allowed to access each others data but can communicate with each other. Through this communication they can pass data. The general mode of such communication are Intents. Application do not need any specific permission to send Intent. In this scenario, collusion emanate if there exist some mode of communication between apps and they are sharing some sensitive information through it that is creating leakage path.

#### *5.3.3. Dynamic colluding apps*

This is a new type of threat called threat of split-personality behavior [60] where the attackers divide malware samples into a benign and malicious part, such that the malicious part is hidden from analysis by packing, encrypting or outsourcing of the code. In such cases the app appears as benign during



analysis phase but while running on real device it become malicious. Similar case exists in case of app collusion, depending on the availability of an analysis system, malware can either behave benignly or load malicious code at runtime. In this scenario, static part of the app collude with dynamic part of the code to leak information. The dynamic part may be another app that get downloaded and installed using social engineering attack or it can be some dynamic code loading. The proof of concept of this type of collusion is demonstrated in [61]. They modified the well known open source malware named AndroRat (Android Remote Administration Tool). This app connects to server and allow remote control of the device. The authors divide the app into two and developed AndroRAT-Split. They put the main service class into one app and added the activity class into another app to create collusion attack. Detection rate of original sample on VirusTotal is  $15/56$  i.e.  $15$  out of  $56$  antivirus engines says that the app is malicious; whereas when AndroRAT-Split is analyzed on Virus-Total, the detection rate is  $0/56$  i.e. it is considered as benign app.

#### 5.4. Challenges in Collusion Detection

The detection of Android ICC based collusion faces many challenges. In the following, we summarize the major ones:

- How to characterize the context associated with communication channels with fine granularity?
- How to provide scalable solutions with minimum complexity to vet a large number of apps for possible collusion?
- How to define security policies for classification that reduce the number of false alerts?
- It's pretty hard to have a policy that is at the same time consistent and still provides realistic results without over-tainting.

The solutions to detect collusion in apps must be capable of doing analysis of multiple apps simultaneously aka inter-app analysis.

## 6. Inter-Application Analysis

This section discusses the main defense techniques for inter-app analysis: static, dynamic and policy based.

### 6.1. Static Inter-App Analysis

Static inter-app analysis consists of examining and auditing the code without executing it [62]. Android apps are analyzed without really running them by inspecting the source code. Static analysis techniques act as a potential weapon for conducting the behavioral analysis of an application i.e. detecting whether an application is benign or malicious. It extensively explores data flows in a program and subsequently detect paths through which information can be leaked. It can be used to detect problems such as cross site scripting (XSS) [63], SQL injection [64], buffer overflows [65], access control problems and many more. Resources and techniques of static inter-app analysis are detailed below.

#### 6.1.1. Resources

The resources of Android apps from which information can be extracted in static analysis includes manifest file, dalvik byte code, libraries, etc [66].

The information obtained from manifest file includes the name of the package, list of components, list of permissions, version, etc. It also reveals information about intents and intent-filters used for communication. Level of API and libraries required by an application for execution is also mentioned in the file [67]. Android apps are written in java and compiled to byte code. This byte code is further translated to dalvik byte code and stored in classes.dex (dalvik executable) file. This file reveals information about the structure of an application and methods used by it. It is analyzed to detect potentially malicious actions such as sending SMS to premium numbers, use of reflection or encryption, access of sensitive resources, etc. [68].

Java libraries can be statically analyzed in order to obtain data flow summaries of an application. It can be useful to determine malicious flow in an application.

### 6.1.2. Techniques

The technique employed to perform static analysis depends on the depth and purpose of analysis. Various static analysis techniques used by researchers include taint analysis [69], dataflow analysis [70], entry point analysis [71] etc. Some of the most prevalently used techniques are explained below.

Taint analysis is also known as user-input dependency checking [69]. The concept behind taint analysis is that any variable altered by the user becomes tainted and is considered vulnerable. The taint may flow from variable to variable during a course of operations and if the tainted variable is utilized to perform some harmful operation, it becomes a breach in security. Taint analysis detects the set of instructions that are affected by user inputs. It helps in identifying sensitive information leakage.

Data flow analysis determines the information flow between various components. It is the essential analysis need to detect leakage of sensitive data. For instance, for a variable, it can detect all the possible sources of a variable's value i.e. where do values assigned to a variable come from, all the possible values a variable can possess, all the sinks where its value passes further, etc [72]. Data-flow analysis can be of various types depending on the context of analysis [66].

- Context sensitive data flow analysis [73] is an inter-procedural analysis technique. It examines target of a function call by focusing on calling context.
- Path sensitive data flow analysis [74] takes into account the branching statements. It analyzes the information obtained by the state obtained at conditional instructions.
- Flow sensitive data flow analysis [75] considers the order of instructions in a program.
- Inter-procedural data flow analysis [76] takes into account the flow of

information between procedures. It is achieved by constructing call graphs

- Intra-procedural data flow analysis [77] involves the flow of information within a procedure.

Entry point analysis [71] helps in determining where a program starts its execution. It is very difficult to identify starting point due to the use of callbacks and multiple entry points.

Accessibility analysis [78] contributes in evaluating the likelihood of following a path between two components. It helps in building reachability graphs showing the path followed through methods for execution of an application.

Side-Effect analysis is performed to compute which variables of a method are affected by its execution [66].

## 6.2. *Dynamic Inter-App Analysis*

Dynamic inter-app analysis refers to the analysis of a program by executing it [62]. Android apps are examined and reviewed by actually executing them on real devices and emulators. Since static analysis does not portray the complete picture of an application, for example network data stored in the memory heap during run time is not available before executing app, obfuscated strings are hard to recognize from decompiled codes etc, therefore dynamic analysis is important to identify malicious applications, information leakage, sensitive data flows and vulnerabilities present in applications [79].

### 6.2.1. *Resources*

The resources of Android apps from which information is extracted in dynamic analysis includes application framework, information about native code libraries, kernel parameters, cpu parameters, memory parameters, information about dynamically loaded libraries, etc. [80].

Application framework is a software library which imparts basic structure for developing applications. It provides information about components and processes currently running and invoked API calls. It keeps record of which disk

location belong to which file, keystrokes done and all the network inputs and outputs of function call.

Native library is a library that includes native code. Native code refers to a code written in C or C++ and compiled to machine code. Since native libraries are linked at run time, dynamic analysis is the suitable approach to identify the data flows in an application [14].

Kernel is the core of Android operating system. It is responsible for the management of hardware interactions. It provides information about the interactions with system protected resources which are not directly accessible [81].

Other parameters of cpu, memory etc. can be captured dynamically when the program is running.

#### *6.2.2. Techniques*

Various dynamic analysis techniques used by researchers that include System hooking [82], Taint analysis [2], Instrumentation [83], System call tracing [84], Debugging [85], Code emulation [86] etc.

System hooking involves altering or amplifying the functionalities of applications or components of application, by anticipating function calls, events and transmitted messages between the components [87]. It assists in conducting dynamic analysis by intercepting and modifying API calls made by the target app. It is used to capture data flows, construct event ordering, record the parameters of passed messages and store values of run-time variables [81].

Dynamic taint analysis [88] starts by tainting the data that is initiated from untrusted sources, specifically user supplied inputs. Later, these tainted variables are stored and whenever any of these variables are used for carrying sensitive data they are tracked down to detect sensitive paths [86].

The term instrumentation pertains to the capability of monitoring or evaluating the performance of product and interpreting errors [89]. Android apps are instrumented to monitor actions of specific components such as logging number of times a particular service is called, etc. It is achieved by injecting smali codes. Smali is an intermediate representation of dalvik byte code which is inserted to

keep log of actions of specific components.

In order to perform system call tracing, a system call tracer is embedded into the system that logs the invoked interrupts or APIs as the program runs on the system [90].

In code emulation, the malicious code is executed on virtual machines with replicated CPU and memory management system, rather than real processor [90].

### *6.3. Policy Enforcement Based Analysis*

Policy (a.k.a rule) enforcement based techniques make use of certain set of policies(rules) that are considered as normal or benign. These policies can be represented either in the form of regular expressions or any new policy language. The access of apps to any policy protected resource is verified against the pre-defined policy-set [91]. Verification can be done statically on the intermediate program code and can also be enforced at install-time or run-time. If the resource access adhered to the policy-set, it is considered benign. Any violation is referred as malicious behaviour [92]. The challenge posed by this defence mechanism lies in identifying, defining and maintaining the policy-set. It should not be very strict that may generate false-positives but at the same time it should not be too liberal to generate more false-negatives [9].

#### *6.3.1. Resources*

The resources of Android apps from which information can be extracted in policy based analysis techniques depend on the nature of the policy-set and where they are applied [93]. For eg., if the policy set considers permissions and their corresponding API call then, manifest, dex files and libraries are suffice to extract relevant information. However, when policies are enforced at install-time or run-time, hooking or instrumentation need to be done. In later case, relevant information can be extracted from system parameter like registers, CPU etc.

#### *6.3.2. Techniques*

In order to protect users' private/sensitive data in Android, various solutions have been proposed based on controlling the access of sensitive resources. The

access control can be apply at various system abstraction layers viz. kernel-layer, middle-ware layer and application layer [94]. The access controls are of various type viz. Mandatory Access Control (MAC), Discretionary Access Control (DAC), Role Based Access Control (RBAC), Context Based Access Control (CBAC) and Attribute Based Access Control (ABAC) [95, 96].

In **MAC**, whenever app wants to access policy protected resource, Android kernel will verify the access against predefined rule-set. The access is allowed only if it is authorized. These rule-set is not modified by app or user. In **DAC**, user can define an access control list (ACL) on specific resources. These resources can be accessed when the owner provides permission. **RBAC** is based on the roles of an individual user. The user is assigning to different positions, with permissions to use the resources. The user can access sensitive data based on their assigned role. Till Android 5.1.1, once privileges are granted to the applications, they cannot be revoked. However, in many cases whether the application get a privilege or not depends on the user context and therefore **CBAC** comes into existence in Android. It has the capability to give privileges with dynamically granted or revoked to applications. In **ABAC**, granting privileges to the users is based on attributes which combine with the policies. Authorization relies on a set of operations that is determined by evaluating the attributes associated with the subjects, objects, and requested services.

## **7. State of the art approaches**

In the following section, we present different experimental approaches for the detection of intra and inter-application communication vulnerabilities.

### *7.1. Static Analysis*

This section briefly explains about the approaches conducting static analysis.

*7.1.1. MR-Droid: A Scalable and Prioritized Analysis of Inter-App Communication Risks [17]*

**Objective:** The paper aims to detect inter-app communication threats specifically intent hijacking, intent spoofing and collusion. Authors proposed a MapReduce based framework to scale up compositional app analysis. They also prioritized the identified ICC risks, based on the communication context of apps.

**Methodology:** The MapReduce based approach is divided into two broad steps. In the first step, MR-Droid identifies ICC nodes (both sources and sinks), ICC edges (intent based ICC communication channels) and group inter-app ICCs that belong to an app pair using MapReduce. In the second step, risk assessment module of MR-Droid assigns risk levels to the app pairs based on the semantics and contextual information of the identified ICC channels. The risk assessment module detects the presence of risk and assigns ranking to the detected risk. Prioritizing risks helps to reduce false alarms. To validate the approach, authors manually analyzed around 200 apps.

**Dataset Used:** The dataset consists of 11,996 apps from 24 popular app categories belonging to Google Play Store and 8 apps from DroidBench 3.0 [97] inter-app communication category.

**Limitations:** The paper suffers from the following limitations:

- The proposed approach can handle intent based ICC communications only. Therefore, security risks posed by other inter-app channels like content providers, shared preferences etc. cannot be detected.
- Currently, the approach can detect privacy leakage across two apps. However, leakage involving more than two apps are missed.



### 7.1.2. Detecting Inter-App Information Leakage Paths [32]

**Objective:** The paper presented a model-checking based approach for inter-app collusion detection. The authors presented compositional app analysis to identify set of conspiring apps involved in the collusion. They developed 8 colluding apps and contributed to DroidBench 3.0 [97].

**Methodology:** The proposed approach is divided into four broad steps. In the first step, it leverages DARE [98] and IC3 [99] to extract information related to ICC sources, sinks and, intent based communication channels. In the second step, data-flow analysis has been carried out to map sensitive information provided by sensitive API call to the outgoing intent followed by storing all the extracted information in the database. In the third step, PROMELA model is generated for each app. In the fourth step, model checking is done using the generated models and collusion detection property ( $[\Box](\text{STATUS}==\text{SAFE})$ ) mentioned in the linear temporal logic (LTL) form. If the compositional model of apps do not satisfy the property, then that set of apps are declared as colluding apps set.

**Dataset Used:** The dataset consists of self-developed 8 apps and contributed to DroidBench 3.0 [97] under inter-app communication category.

**Limitations:** The paper suffers from the following limitations:

- The proposed approach can handle intent based ICC communications only. Therefore, security risks posed by other inter-app channels like content providers, shared preferences etc. cannot be detected.
- Not addresses the issue of scalability.

### 7.1.3. Towards Automated Android App Collusion Detection [100]

**Objective:** The paper mentioned that collusion can cause information theft, money theft or service misuse. They defined collusion between apps

as some set of actions executed by the apps that can lead to a threat. They proposed two approaches to identify candidates for collusion. One is rule based approach developed in Prolog and other is statistical based approach.

**Methodology:** In rule based approach, some features are used to identify colluding apps. These features include permissions, communication channels, and set of some actions viz. accessing sensitive information, sending information etc. The statistical approach consists defining probabilistic model, training of the model that means estimating the model parameter on the training set and validating the model on test dataset. Additionally the paper also presented that model-checking is the feasible approach to detect collusion in Android apps.

**Dataset Used:** The dataset consists of  $\sim 9000$  malicious and  $\sim 9000$  benign apps developed by Intel Security.

**Limitations:** The paper suffers from the following limitations:

- The rule based approach can be easily evaded using some evasion techniques like reflection, obfuscation etc.
- The statistical approach performance could be due to a bias of validation dataset towards the methodology.
- Not addresses the issue of scalability.

#### *7.1.4. User-Intention Based Program Analysis for Android Security [101]*

**Objective:** The paper proposed a data structure called ICC Map that statically captures cross-app information flow and based on self-made security policies classify communication among apps as collusion or no collusion.

**Methodology:** ICC Map is a hash map data structure. It stores ICC entry and exit points that can be extracted by scanning bytecode of source and target apps respectively. It is used to statically characterize the inter-app ICC

channels among the Android apps. The detailed description of ICC Map is as follows:

- ICC exit points refer to all Intent based ICC APIs like `startActivity(Intent i)`, `startService(Intent i)` etc., user triggers like `onClick()` and APIs that retrieve private data such as `getAccounts()`, `getPassword()` etc.
- After extracting exit points, data dependence graph (DDG) of intra- and inter-procedural dependencies has been constructed.
- Then, from each ICC exit point, backward depth-first traversal on DDG is done to check if it involves private data or user trigger and store this information in *SourceAppICCExitHashMap*.
- Then construct the data dependence graph (DDG) for the target app and perform forward depth-first traversal on its DDG from each ICC entry point to find any critical operations and store this information in *TargetAppICCEnterHashMap*.

In *SourceAppICCExitHashMap*, each entry consists of source component name as key, and a list of ICC exit, sensitive data and user trigger as value. For example, `(compX, {startService(Intent i), getDeviceID(), onClick()})` represents one entry in the *SourceAppICCExitHashMap*, where `compX` is the component name that initiates the inter-app ICC call `startService(Intent i)` with sensitive data device ID included as part of the Intent of this call and `onClick()` as the user event to trigger this call.

Similarly, in *TargetAppEntryHashMap*, each entry consists of target component name as key, and a list of ICC entry, component protection and critical operation as value. For example, `(compY, {onStart(), No, java.io.FileOutputStream.write(...)})` represents one entry in the *TargetAppICCEnterHashMap*, where `compY` is the component name that receives the inter-app ICC call, and `onStart()` is the entry point of `compY` which is not protected (No) and has critical operation `java.io.FileOutputStream.write(...)`.

Given the above two hash maps, they connect inter-app ICC calls as follows:

- First search for the source component name in SourceAppICCExitHashMap. The search results return its value(ICCExitName, SensitiveData, UserTrigger).
- Then, search the same with the target component in TargetAppICCEntryHashMap to get its value(ICCEntryName, CompProtection, CriticalOperations).
- After that, connect the ICC exit point in the source component with its corresponding ICC entry point in the target component.

These operations provide the complete path of the ICC calls from the source to the destination across multiple apps. Authors call these paths as ICC links and entire data structure as ICC Map. ICC Map cannot be used for apps collusion detection but it helps to identify pair or group of communicating apps.

After this they define four rules/policies that are as follows:

Suppose component  $C1$  in app  $P1$  calls component  $C2$  in app  $P2$ , i.e.,  $C1 \rightarrow C2$

- If the ICC exit point in  $C1$  does not have a valid user trigger and the target component  $C2$  is not protected by permission checking and has critical operation, then this ICC channel is classified as a high risk inter-app ICC channel.
- If the ICC exit point in  $C1$  has a valid user trigger and the target component  $C2$  is not protected by permission checking and has critical operation, then this ICC channel is classified as a medium risk inter-app ICC channel.
- If the ICC exit point in  $C1$  does not have a valid user trigger and the target component  $C2$  is protected by permission checking and has critical operation, then this ICC channel is classified as a medium risk inter-app ICC channel.
- If the ICC exit point in  $C1$  has a valid user trigger and the target component  $C2$  is protected by permission checking and has critical operation,

then this ICC channel is classified as a low (or no) risk inter-app ICC channel.

Authors have also extended these rules to more fine-grained *16* rules that include inter-app ICC call. The detail description of these *16* rules are available in [102].

Thus, based on ICC Map and a set of security policies they can differentiate between benign communicating apps and colluding ones. In their experiments, the proposed method can correctly detect *97.9%* of the *1,433* malware samples. The false negative rate is *2.1%*, i.e., *31* malware apps are misclassified as benign.

**Dataset Used:** *1,433* malware apps collected from [103] and Virus Share. *2,684* apps from Google Play market.

**Limitations:** ICC Map approach suffers from following limitations:

- ICC Map cannot detect collusion through indirect communication channel such as shared files.
- ICC Map cannot capture the scenario that involves complex string operation like both apps read/write to files, however, the filenames are dynamically generated using string operations. Collusion occurs through such scenario is missed by the proposed approach.
- The approach statically identifies the predicted risk level associated with the inter-app ICC calls, but it does not confirm the existence of the collusion.
- The proposed approach has difficulty in performing the analysis on programs that employ obfuscation techniques, dynamic code loading, or use of reflection.

#### 7.1.5. *IccTA* [15]

**Objective:** IccTA is a static taint analyzer to detect privacy leaks between components in Android apps. It claims to improve its precision of analysis by

propagating context-aware information.

**Methodology:** IccTA tool takes APK file (dalvik bytecode) as input and convert it into Jimple (soot’s intermediate representation [104]). After that, it extracts ICC links and related information like ICC call parameters, Intent filter etc. using Epicc [105] and also parses URIs (eg. scheme, host) to support Content Provider related ICC methods (eg. query) using IC3 [99]. To extract ICC links, IccTA have to identify source and target components. Source components are the components that initiate ICC method and target components are resolved by analyzing the values of Intent filter from `AndroidManifest` file of the app. It also needs to analyze bytecode because Broadcast Receivers may be declared at runtime. Then it stores all the extracted information into a database. Based on the extracted ICC links, IccTA modifies Jimple representation to directly connect the components to enable data-flow analysis between them.

IccTA handles three types of methods, ICC methods are replaced by an instantiation of the target component with the appropriate Intent. For callback methods, the tool takes care of both UI triggered event as well as callbacks triggered by Java or the Android system. To handle lifecycle methods, the tool generates a dummyMain method for each component in which it models, the entire lifecycle model of the component. IccTA leverages FlowDroid [73] to build a complete control flow graph of the Android app under analysis. This graph allows to analyze the context (eg. the value of Intent) between two components. In the end, IccTA also stores the reported tainted path (leaks) into the database which can be reused in later analysis. IccTA achieved 96.6% precision while analyzing privacy leaks from the samples of DroidBench and ICC-Bench. IccTA can perform inter-app analysis when used with APKCombiner [3] which is a static tool that scales down inter-app communication analysis to intra-app communication analysis. APKCombiner disassembled every app to obtain manifest and smali files using android apktool [106], a reverse engineering tool. After that all files corresponding to different apps are

combined together into a single directory and conflicts are resolved.

**Dataset Used:** 22 apps from DroidBench, 15000 apps from Google Play Store, 1260 apps from Genome Malware, 16 apps from ICC-Bench.

**Limitations:** IccTA suffers from the following limitations:

- IccTA resolves reflective calls only if their argument are string constants, which is not always the case.
- It cannot detect leak through multi-threading. It assumes the execution of threads in arbitrary but sequential order.
- It can miss leaks through native calls that their rules model incorrectly.
- It cannot handle rarely used ICC methods like `startActivities` and `sendOrderedBroadcastAsUser`.
- It cannot resolve complicated string operations which are generated using `StringBuilder`.
- The string analysis done by IccTA is within a single methods which may cause false alarms.
- IccTA cannot analyze apps of big size as it requires too much memory consumptions and system often gets hang.

#### *7.1.6. Automatic Detection of Inter-Application Permission Leaks in Android Applications: PermissionFlow [13]*

**Objective:** PermissionFlow is a single-app static analysis approach that handles attacks related to obtaining unauthorized access to permission-protected information. It focuses on three types of attacks viz. permission collusion, confused deputy and Intent spoofing. PermissionFlow uses taint analysis to capture the flow of permissions.

**Methodology:** PermissionFlow consists of three major modules, i.e. Permission Mapper, Rule Generator and Decision Maker. The approach consists of identifying APIs whose execution leads to permission-checking. This is done through permission mapper. Then another module, rule generator will define rules for tainting. It considers the APIs selected by permission mapper, to be the sources of taint and define rules to capture their corresponding sinks. Then based on the information extracted from APK file of an app through apktool and dex2jar, PermissionFlow leverages another open-source tool named Andromeda to identify flows and components. Decision maker will allow or disallow the flow based on permissions.

**Dataset Used:** PermissionFlow tests 313 popular Android Market applications, and then identifies that 56% of them use inter-component information flows that may require permissions.

**Limitations:** Permission flow approach suffers from the following limitations:

- Permission flow does not handle native code permissions.
- Permission flow records a large number of false positives due to the checking of redundant permissions and data dependent checks.
- Permission flow gives false negatives for the apps that transfer protected information between components before returning it. This is due to the use of Implicit intents as it prevents identification of the class names for invoked `Activity`.

#### 7.1.7. FUSE [14]

**Objective:** FUSE proposed a approach that starts by single-app static analysis accompanied with lint tool followed by multi-app information flow analysis. Lint tool is used to mitigate limitations of static analysis. They demonstrated limitations of single-app analysis by detecting more information



flow paths in multi-app analysis.

**Methodology:** FUSE works in two broad steps. Single-app analysis, followed by multi-app analysis based on violation of specified security policies. The first step takes an appkit (collection of apps) as input, analyze each app individually to produce extended manifest data structure. In the second step, all the extended manifests are combined and collusion is checked based on violation of specified security policies. The detailing of all the steps are as follow:

Single-app analysis: In this step, each application in the appkit is individually analyzed to create its corresponding extended manifest data-structure. The extended manifest represents the internal information flow graph from application inputs (sources) to application outputs (sinks). According to FUSE, sources are the inputs to each component or permission protected resources. Sinks are the means by which a component can send (possibly sensitive) information to another component or to the outside world. In FUSE, a version of Andersen’s analysis [107] is used to compute a call graph and determine reachable methods in each application. The analysis also considers Android Framework and Java libraries along with the application. The application is tainted with taint labels at every source and if these labels reach to sinks, data leakage path is flagged.

Multi-app analysis: In this step, FUSE takes the entire appkit along with all the extended manifests as input. The output is the multi-app graph with the flow of information between the applications. In the graph, set of all the permissions and each sources and sinks present in any application becomes the node. There must be an edge from source or permission to the sink, whenever there is the flow of information between them. There also exist the edges from sink to component, if sink can send IPC message to that component.

FUSE defines coarse-grained information flow assertions based on permissions combination. The multi-app graph is checked against these assertions. User is alerted if there is any violation of the assertion occurs in the graph. FUSE also uses security linter tool, to overcome the limitations of static analy-

sis. The tool issues warning if there exists problem like, component hijacking, dynamic registration of broadcast receiver, use of insecure credentials, presence of reflection, unused permissions or writing to public files.

**Dataset Used:** 189 applications drawn from Nexus 4 running Android 4.4.2. 1124 applications of F-Droid updated till May,2014. 1260 applications from Genome project dataset, applications from DroidBench.

**Limitations:** FUSE suffers from following limitations:

- The biggest limitation of the approach is that it is not publicly available to test. It is designed for commercial purposes.
- Defining coarse-grained information flow assertions leads to many false alert as the behavior of the application is not considered.
- It does not support all versions Android APIs,

#### 7.1.8. *Amandroid [22]*

**Objective:** Amandroid is a static analysis tool, that has the capability of calculating all objects' points-to information in a both flow and context-sensitive way. This tool detects whether there is any information leakage from a sensitive source to a critical sink; by providing an abstraction of the app's behavior.

**Methodology:** Amandroid proceeds by converting an app's Dalvik bytecode to an intermediate representation (IR) for subsequent analysis. Then, Amandroid generates an environment model that emulates the interactions of the Android System with the app to limit the scope of the analysis for scalability. Amandroid builds an inter-component data flow graph (IDFG) of the whole app. IDFG includes the control flow graph; that tracks the set of object creation sites that reach each program point. The core component is to build a precise IDFG of the app; the flow-sensitive and context-sensitive data flow analysis to calculate objects points-to information is done at the same time with building inter-procedural control flow graph. Amandroid builds the data

dependence graph on top of the IDFG, then it induces explicit information flow. This framework provides an abstraction of the app’s behavior, and can be used for a number of useful security analysis as data leak detection, data injection detection, and detection misuse of an API.

**Dataset used:** Amandroid is tested on 753 Google Play apps by the Epicc group, and 100 potentially malicious apps from Arbor Networks.

**Limitations:** Amandroid has the following limitations:

- Amandroid has limited capability to handle exceptions. Amandroid may not detect an exception, when an app has a security issue where the core of an exception handler plays a role.
- Amandroid does not handle concurrency and reflections. An app may have multiple components and then may run concurrently; and when multiple components interleave this may induce some security issues.

#### 7.1.9. Android Taint Flow Analysis for App Sets [108]

**Objective:** DidFail conducts static taint analysis of Android apps by augmenting FlowDroid and Epicc tools to detect intra-component and inter-component information flow in a set of apps. It performs analysis in two phases where the first phase determines information flow within the app and second phase determines flow across the apps.

**Methodology:** DidFail accepts a set of apps as the input. The analysis is performed in two phases:

- The first phase constitutes of 4 steps - TransformAPK, FlowDroid (modified), Dare and Epicc.
  1. *TransformAPK*: In this step, each APK is modified by using Soot. Initially, APK is converted to an intermediate representation known as `jimple`. Later, all the send intent method calls are located and just before the method call, a new method call is inserted that provides a unique ID to the sent intent. The jimple code is then repackaged into an APK and passed as an input to the next step.

2. *Dare*: This tool accepts transformed APK as an input and produces retargeted java class files as an output.
  3. *Epicc*: This tool accepts retargeted java class files and transformed APK as an input and provides parameters of sent and received intents such as **action**, **category** as an output to the second phase.
  4. *Modified FlowDroid*: DidFail has modified FlowDroid by adding few intent method calls as sources (`onActivityResult()`) and sinks (`setResult()`). It has also added code to analyze `putExtra` call for the intents that are uniquely identified by ID in TransformAPK step. FlowDroid accepts transformed APK as the input and conducts taint analysis. It provides flows within the components of an app as the output.
- The first phase identifies intents as tuples. For example,  $I < C_1, C_2, ID >$  where  $C_1$  = component that sends the intent,  $C_2$  = component that receives the intent and ID = unique identifier of the intent. This phase identifies flows within an app and pass these flows as an input to the second phase.
  - In the second phase, inter-app communication among the set of apps is resolved i.e. an intent sent by an app is matched with the intent-filters of other apps to identify the receiver. Once the receiver is identified a flow from source→sink is detected.

**Dataset Used:** DidFail is tested on two app sets where the first app set contains three apps developed by the authors and second app set contains three apps taken from Droidbenchmark.

**Limitations:** DidFail suffers from the following limitations:

- DidFail does not handle native calls and reflection.
- DidFail focuses only on **Activity** component of Android app. It does not handle service, broadcast receiver and content provider.
- DidFail cannot detect flow of information when static fields are used as a

source or sink for intents i.e. it misses the flow if an intent reads information from static field.

- If the tainted information propagates through a chain of apps, then DidFail fails to detect the flow.

#### *7.1.10. Analyzing Inter-Application Communication in Android: ComDroid [10]*

**Objective:** ComDroid is a tool that detects application communication vulnerabilities and could be used by developers and reviewers to analyze their own applications before release. The main purpose of this tool comes from the fact that Android’s message passing system can become an attack if used incorrectly (personal data loss, information leakage, phishing, etc.) These vulnerabilities stem mainly from the fact that Intents can be used for both intra and inter application communication.

**Methodology:** ComDroid considers two types of analysis: Intent analysis and Component analysis.

In Intent analysis, ComDroid statically analyzes method invocation to a depth of one method call. In this way it performs flow sensitive intra-procedural static analysis, with a limited inter-procedural analysis. This tool parses dalvik files and tracks the state of intents, registers, sinks, intent-filters, and components. For each method that uses intents, this tool can track the value of each string, class, intent and intent-filter. For each Intent object, ComDroid tracks the following:

- whether the intent has been made explicit;
- whether the intent has an action;
- whether the intent has any flags set; and
- whether the intent has any extra data.

When it detects that an implicit intent being sent with weak or no permission requirements, ComDroid issues a warning as this situation is eavesdropping prone. There are two types of warnings viz. with data, and without data, in order to distinguish action based attacks from eavesdropping.

In Component analysis, ComDroid examines application's manifest file to get components and translates dalvik instructions to get information about each component. ComDroid treats activities and their aliases as separate components because an alias field can increase the exposure surface of the component. It generates a warning about a potential intent spoofing attack, when it detects that a public component is protected with no permission or a weak permission. ComDroid also issues warnings for receivers that are registered to receive system broadcast actions (that are actions sent by the system).

In order to resolve these warnings, a solution proposed by authors was to add a call to `android.content.Intent.getAction()` to verify that the protected action is in the Intent (authentication of the sender of the Intent). This differs from other Intent spoofing attacks where the solution is to make the component private.

**Limitations:** ComDroid suffers from the following limitations:

- False Negatives: ComDroid tracks Intent control flow across functions, and did not distinguish between paths through if and switch statements. For instance, an application might make an Intent implicit in one branch and explicit in another, ComDroid would always identify it as explicit.
- Privilege Delegation: ComDroid does not detect privilege delegation through pending Intents and Intents that carry URI read/write permissions.
- Verification of the existence of attacks: ComDroid issues warnings and not verify the existence of attacks. For instance, some components are intentionally made public for the purpose of inter-application collaboration. It is not possible to infer the developer's intention when making a

component public. It is the role of the developer to verify the veracity of the warnings.

## 7.2. *Dynamic Analysis*

This section briefly explains the proposed tools and approaches that conducts analysis dynamically.

### 7.2.1. *IntelliDroid [109]*

**Objective:** IntelliDroid is a generic tool that generates input specific for a dynamic analysis tool to perform analysis more precisely by reducing false positives. Instead of static or dynamic analysis, this work proposes targeted analysis. It is achieved by preliminarily doing background study about the dynamic analysis tool and static analysis of the application given as input to the dynamic analysis tool. It helps in triggering target APIs and consecutively leads to more efficient and effective dynamic analysis.

**Methodology:** Android apps contain multiple event handlers, which when triggered in a particular sequence with specific inputs, reveal malicious behavior. This environment and input is provided by IntelliDroid to dynamic analysis tools. IntelliDroid acts in 6 steps viz. Specifying target APIs, Identifying paths to target APIs, Extracting call path constraints, Extracting event chains, Determining run-time constraints and Input-injection to trigger call paths.

In the first step, APIs to be targeted are identified by analyzing either API methods [2], system calls [110] or low-level events [111].

In the second step, paths to the targeted APIs are discovered by conducting static analysis. IntelliDroid obtains information about components of an application and its lifecycle methods by reading its manifest file. It identifies entry-points of an application and create a partial call-graph to look for initialization of callback listeners. It adds circumvented listener methods to the entry-points list and creates a new call graph. This process is repeated re-

cursively till no more entry points are found. By traversing path from event handler's entry point to target API invocation, target paths are extracted for every target API.

In the third step, call path constraints are determined by conducting control and data-flow analysis on the control flow graph (CFG) in forward direction. If more than one path exists from one-method invocation to other in CFG, IntelliDroid combines the constraints of each path by using logical OR operator. For the cases where extracted constraints are return values of other method invocation, they are added with main path constraints by using logical AND operator.

The reason for executing fourth step (extracting event chains) is that the path constraints can be heap variables whose value cannot be determined statically by observing entry-points. To determine their values, the lines in the code containing the heap variable definition are looked for. The event handler containing the heap variable definition are tracked and stored. The route from event handler to heap variable store statement becomes supporting path and its constraints becomes supporting constraints. Their value is determined at the time of resolving constraints and subsequently used for storing path constraints.

In the fifth phase i.e. determining run-time constraints, the value of variables that are still unresolved are obtained at run-time, just before the event injection.

In the sixth step, finally the input fulfilling constraints are injected to trigger the call paths. The component of IntelliDroid responsible for injecting input consists of a computer attached to a device. Communication between them occurs through IntelliDroidService. The static part of IntelliDroid is responsible for specifying inputs for the targeted APIs. It supplies these inputs to dynamic part which is accountable for inserting the inputs at device-framework interface of Android.

For the static analysis, source code is not used. The APK files are unpacked using Dare [98] and APKParser [112]. The Java bytecode is then passed to static part which uses WALA static analysis libraries [113]. For dynamic analysis, Z3 constraint solver [114] is used. The IntelliDroidService client program is



executed using Python.

IntelliDroid is tested with TaintDroid, a dynamic analysis tool. On an average 72 inputs have been injected. Out of 75 malware instances, IntelliDroid was successful in identifying 70 instances. It is observed that 138.4 seconds on an averages is required per application.

**Dataset Used:** It has been tested on 1260 malware samples from Malware Genome Project [115] and 1066 benign apps from Android Observatory [116].

**Limitations:** IntelliDroid suffers from the following limitations:

- IntelliDroid does not handle implicit Intents, Content Providers and native code.
- The extracted constraints are sometimes very complex such as trigonometric functions. It cannot be resolved by constraint solver. Currently, human intervention is required to solve such constraints.
- IntelliDroid partially handles reflection as it cannot identify the path constraints after the reflected call.
- IntelliDroid is not capable of creating inputs for encrypted and hashed functions.

#### 7.2.2. *IntentDroid* [117]

**Objective:** IntentDroid is a framework that dynamically examines Android apps for IAC (Inter Application Communication) related integrity vulnerabilities such as custom uri's, payloads in IAC messages etc. It created attack scenario for 8 vulnerabilities viz. Cross-Site Scripting, SQL Injection, Unsafe Reflections, UI (User-Interface) Spoofing, Fragment Injection, Java Crashing, Native Memory Corruption and File Manipulation. It analyzes Activity component of apps by implementing attack scenarios in a way to obtain effective path coverage with minimum overhead.

**Methodology:** IntentDroid tests the applications in three phases viz. Instrumentation, Testing and Reporting.

In the instrumentation phase, the app under analysis is instrumented to store library calls and access to user-supplied data. The app is reverse engineered through apktool to extract its manifest file. Manifest file is parsed to extract public components. IntentDroid specifies three cases to call any activity as public:

- if the activity is exported via Intent filter(s);
- if the activity access does not require any permissions (system or signature); and
- if any unvalidated data which is originated from any public component, passes through it;

These activities communicate via Intents and therefore all Intents form a set of IAC input points for the Testing phase.

In the testing phase, to detect whether a vulnerability exists in the app or not, IntentDroid has created attack scenarios. Testing occurs in three steps Monitoring, Testing and Exploration. During monitoring, IntentDroid sends a message to the app under test. It uses system-level hooks to records all the security concerned APIs called by the app and custom fields (such as `getStringExtra`, `getFieldExtra`, etc.) accessed by it. IntentDroid analyzes the app with direct and indirect access to custom fields. For direct access, IntentDroid iteratively detects the extra fields of the Intent and looks for the behavior of the app. If any extra field is observed, that IAC input point migrates to testing phase. For indirect access, bundle object created for the message sent by IntentDroid is observed by installing a monitor in `Intent.getBundle()` method. It checks the extra and data fields for payloads. If any inserted payload is found, bundle is considered relevant and exploited further by implementing attack scenarios. After monitoring the app undergoes testing. The app is tested

for implementation of an attack scenario on an IAC input point by sending probe request. If the result is positive, that attack scenario is implemented. In the end during exploration, boolean variables are analyzed to detect the path followed by an Intent. It introduces two terminology for boolean variables viz. Independence and Dominance. A boolean variable is Independent, if its execution is not dependent on any other boolean variable. On the contrary, a boolean variable is Dominating, if its execution decides the application of other boolean variable (nested variables). Boolean variable analysis reflects the handling of incoming data by the Intent.

In the Reporting phase, IntentDroid reports the number of vulnerabilities present in an app after implementing all the possible attack scenarios on the app.

For evaluating IntentDroid, apps in the test dataset are manually tested by professional ethical hacker through brute-force fuzzing tool. It detected 163 IAC vulnerabilities across 80 apps. IntentDroid is able to detect 150 IAC vulnerabilities giving a recall rate of 92%.

**Dataset Used:** IntentDroid is tested on the dataset of 80 apps, out of which 4 are enterprise apps, 4 are shipping apps and remaining 73 are the most popular Google Play apps. These apps are tested on Samsung Nexus 5 device with Android 4.4 installed on it.

**Limitation:** IntentDroid suffers from the following limitations:

- IntentDroid does not test Services, Broadcast Receivers and Content Providers for IAC vulnerabilities.
- IntentDroid does not consider multi-app attack.

### 7.2.3. TaintDroid [2]

**Objective:** TaintDroid is a security framework that dynamically detects sensitive information leakage in ICC between Android apps. TaintDroid

extends the functionality of Android operating system to record the flow of confidential and vulnerable data through installed applications.

**Methodology:** It combines four fragments of taint dispersion viz. Variable Level, Message Level, Method Level and File Level. Variable level capturing is done for single app analysis. Variable taint tags are stored adjacently to variables in memory. Message level capturing is done for tracking communication between applications. One taint tag per message is stored which is the combination of variable taint tags. Method level capturing is done for native libraries granted by system. File level capturing is done to guarantee data preserves their taint markings. One taint tag per file is stored.

TaintDroid divides the Android architecture in three modules viz. Interpreted Code, Userspace and Kernel. It considers a scenario in which a message is transmitted from source app to destination app, where source app is assumed to be trusted and destination app is assumed to be untrusted.

Interpreted code module of the source app taints (labels) the data originated from confidential and vulnerable sources (such as GPS coordinates) as taint sources in a trusted application. The assigned taint labels are stored in *Virtual Taint Map* present in Userspace module.

Userspace module includes Dalvik VM interpreter (which is invoked by native methods) and Binder IPC library. TaintDroid modifies Binder IPC library so that in case of an ICC, the parcel transmitted between two apps carries a taint tag which is a combination of taint markings of all the data carried inside the parcel. The customized Android platform tracks the flow of tainted data through dynamic taint tracking. In dynamic taint tracking, the labels are assigned transitively to components (such as IPC messages, variables etc.) when sensitive information propagates through them. When the tainted data of trusted app is to be sent as an ICC message, the data is first transferred to modified Binder IPC library. It creates a parcel and ensures that parcel holds a tainted tag, representing the combination of all the tainted data tags inside the parcel. The parcel is sent to untrusted application via Kernel.

On the receiver side, modified Binder IPC library extracts the data from the parcel and sends it to Dalvik VM Interpreter. The flow of tainted data is monitored. Whenever tainted data leaves the system either by transmission over the network or by any tainted sink, the scenario is logged and reported to the user immediate. The logged information includes labels of data, receiver of the data and the application culpable for sending the data.

TaintDroid reported that on an average two-third of the considered apps are leaking sensitive data. TaintDroid incurs *14%* of performance overhead on CPU-bound micro-benchmark.

**Dataset Used:** *30* most popular android apps are selected from *12* categories of Android Market. By applying TaintDroid, *65* scenarios of information misuse across *20* apps has been identified. Out of *1130* logged TCP connections, *105* has been found responsible for carrying tainted data out of the system. *15* out of *30* apps have been found leaking user’s location to advertising servers. *7* applications have been detected leaking user’s device ID.

**Limitations:** TaintDroid suffers from the following limitations:

- TaintDroid only handles data flows. It does not considers control flows.
- TaintDroid is not able to tag native code which leaves many sensitive sources untouched.
- In case of File level tracking, storing one taint tag per file gives a lot of false positives.

### *7.3. Policy Enforcement Based Analysis*

#### *7.3.1. Collusive Data Leak and More: Large Scale Threat Analysis of Inter-app Communications[118]*

**Objective:** The paper presents a tool named DIALDroid (Database powered ICC Analysis for Android). To the best of our knowledge, this is the first state-of-art that proposed large scale detection of collusion and privilege

escalation. They also provide the first inter-app collusion real-apps benchmark of 30 apps. Till now, this is the most efficient tool available in the literature for inter-app vulnerability detection [17].

**Methodology:** In this paper, authors proposed DIALDroid that works in four broad steps. In the first step, permissions and intent-filter attributes are extracted from the manifest file and ICC entry/exit points are identified. In the second step, static taint analysis is performed to determine paths from sensitive sources to the intents being sent and intents received to sensitive sinks. It leverages Flowdroid[73] to conduct dataflow analysis at high precision. In the third step, all the extracted data is organized in mysql database comprising of 42 tables. The relational database provides scalable and efficient storage. To reduce the computational complexities, DIALDroid filters out ICC communications that are not sensitive. Finally, security policies are implemented using sql queries to detect the presence of collusion or privilege escalation.

They improved the preciseness of intent discovery by implementing incremental callback analysis. In dataflow analyzer, if any app takes more than 5 minutes, DIALDroid resets the analysis by decreasing precision to maintain the trade-off between performance and precision. To avoid deadlocks, the app is analyzed for maximum 20 minutes. The crash rate of DIALDroid is very less than IccTA+APKCombiner [15] and it is more accurate than [15, 30].

**Dataset Used:** The dataset consists of 110,150 apps which includes 100,206 most popular Google play apps and 9,944 apps from Virushare. DIALDroid is also analyzed on Droidbench apps and ICC bench apps.

**Limitations:** DIALDroid suffers from the following limitations:

- DIALDroid resolves reflective calls only if their argument are string constants, which is not always the case.

- DIALDroid analyzes an app only for 20 minutes. If an app takes more than that time, DIALDroid stops analysis.
- DIALDroid can handle intent based ICC communications only. Therefore, security risks posed by other inter-app channels like content providers, shared preferences etc. cannot be detected.

### 7.3.2. Intersection Automata based Model for Android Application Collusion [20]

**Objective:** This is a static inter app analysis tool that can take multiple apps simultaneously for analysis and detect potentially colluding apps.

**Methodology:** In this paper, authors proposed a novel automaton framework that allows detection of intent based collusion among apps. The presence of collusion is detected by intersecting application and policy automata. Application automaton depicts intent-based communication among apps. Policy automaton have policies like if access of an API that requires `READ_SMS` permission in one app is followed by an API call that requires `SEND_SMS` permission in another app. The policy is searched in the application automaton and if the match is found, the tool will check the presence of `READ_SMS` permission in the second app. If it is not found then the tool declares the presence of collusion otherwise no collusion.

The detection framework operates at the component-level. They tested their approach on 21 apps by taking all possible combinations (two at a time) and successfully detected presence/absence of collusion among them. Time and space complexity of the proposed tool is  $O(n)$  where  $n$  is the sum of all the components in applications under analysis.

**Dataset Used:** 3 applications from DroidBench inter-app communication category, self-developed 14 applications and 4 applications from Google Play Store.

**Limitations:** The tool suffers from the following limitations:

- The false alarm rate is very high as the tool is not performing any data-flow analysis. If there is intent communication between two apps without any data transferred, the tool raise warning of collusion.
- Only intents are considered as a means of communication.

*7.3.3. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies [94]*

**Objective:** FlaskDroid is policy-driven tool that provides security for kernel resources (like files, IPC, etc.) as well as middleware resources (like Intents, Content Providers, etc.). The security enforcement is through providing mandatory access control on both middleware and kernel layers of Android simultaneously. They extended Android’s middleware layer with type enforcement and present a new policy language to capture the semantics of this layer.

**Methodology:** FlaskDroid plants various **Object Managers** at middleware and kernel layer that are responsible for assigning security context to objects. Related policies are managed by security servers deployed at different layers. The object manager makes access control decisions by using security servers at their respective layer. Also the deployed policies at both the layers are synchronized meaning change of policy in one layers, automatically reflect in another layer. Following are the major components of FlaskDroid:

- **SE Android Module:** SE Android module restricts the privileges of root account to constrain the file-system privileges of the app. It is also responsible for restricting apps from bypassing middleware level policy enforcement check. For e.g., it restricts app from directly accessing the contacts database file instead, the app must access contacts via `ContactsProvider` app.



- **Userspace Security Server:** It is responsible for taking policy decisions for all userspace access control.
- **Userspace Object Managers:** In FlaskDroid, middleware services and apps act as Userspace Object Managers (USOMs) for their respective objects. Currently it comprises of 136 policy enforcement points.
- **Context Providers:** A context is the current security requirements of the device. It is derived from various criteria, such as physical, the state of apps and the system. Context Providers are the plugins to Userspace Security Server that allows control of contexts and their definitions.

**Dataset Used:** FlaskDroid is evaluated on the apps collected from Malware Genome [119] and Contagio minidump [120]. The authors also developed synthetic apps that exhibits root exploit, over-privilege, information leakage, sensory malwares, confused deputy and collusion attacks.

**Limitations:** FlaskDroid suffers from the following limitations:

- Access control rules are human user trail based. Therefore, revision of rules is time-consuming process. Also, limited human trials cannot guarantee full coverage of possible access control rules.
- Many false alarms while detecting confused deputy and collusion attacks. As FlaskDroid relies on application inputs/outputs and does not consider the information flow within apps.
- Simultaneous analysis of multiple apps is not provided by FlaskDroid.

#### 7.3.4. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks [26]*

**Objective:** XmanDroid (eXtended Monitoring on Android) is a dynamic framework that extends the monitoring mechanism of Android to detect and prevent application-level privilege escalation attacks. It is based on runtime

system-centric policies. Two types of application-level privilege escalation attacks are handled by XmanDroid viz. Confused Deputy attacks and Colluding attacks.

**Dataset Used:** XManDroid developed their own dataset that consists of seven apps. These app set exhibit privilege escalation attack through ICC communication links and three types of covert channels viz. synchronized adjustment and reading of the voice volume, change of the screen state and change of the vibration settings.

**Methodology:** XManDroid consists of three elements:

- **Application Installer:** It is responsible for installation and uninstallation of applications. It makes use of package manager for incorporating the changes and rebuild a new state, whenever any new application gets installed.
- **System Policy Installer:** It is responsible for the installation of explicitly defined list of system policies in the Android middleware.
- **Runtime Monitor:** It is responsible for enforcing mandatory access control in Android like permissions are checked at this interface, take decisions whether to allow an ICC or not based on the information about installed apps and their communication. Whenever a request for an ICC call reaches, it is either approved or disapproved by reference monitor after validating it with policies database and whether the given ICC call leads to privacy leak or not.

System representation is done using graph schema where UID assigned by the system to an app is a vertex and the information about exchanged intents are the edges. By applying the rules of graph theory, transitive transfer of information is detected during ICC. This graph is used for defining rules in system policies.

**Dataset Used:** XManDroid developed their own dataset that consists of seven apps. These app set exhibit privilege escalation attack through ICC communication links and three types of covert channels viz. synchronized adjustment and reading of the voice volume, change of the screen state and change of the vibration settings.

**Limitations:** XmanDroid suffers from the following limitations:

- False Positives: XManDroid suffers from high positives if the app under analysis is over-privileged. The defined system-policies are not tuned properly.
- Attack at kernel level: XManDroid cannot handle privilege escalation attacks done at kernel level that can exploit the system to gain root access.
- Single app analysis is missing: XManDroid cannot detect malicious app as applications within a single sandbox have equal privileges and cannot perform privilege escalation.

#### 7.4. Case Studies

This section presents various studies done to demonstrate the serious effects of information leakage in android apps.

##### 7.4.1. Case of Collusion: A Study of the Interface Between Ad Libraries and their Apps [121]

**Objective:** In this study, API calls used by Ad Libraries to communicate with host applications are analyzed. Host applications have access to sensitive and private user data. API calls are capable of transmitting demographic data about user, which is of great interest for Ad agencies. Therefore, this interface presents a serious impact of user's private information leakage.

**Methodology:** Initially, manual identification of *103* individual and analytic libraries are done. Then, apps are disassembled using dexdexer [122] followed by app parsing, to detect all the API calls (using package name) that occurred between apps and any ad library. Later, all the captured API calls are recorded to keep a track of calls which were actually used from the considered dataset. Frequency of each call is also calculated and recorded. The group of detected API calls are assembled to recreate API of Ad libraries. Recreation is followed by detection of privacy related API calls through manually examining each API call using method name and parameters. The study found that user private data is leaked and stored in databases. These demographic data can be correlated to map a user to a real world person.

**Dataset Used:** Dataset consists of *114,000* apps downloaded from Google Play Store. APIs of *103* ad libraries used by apps in the dataset are reconstructed. Top *20* ad libraries used in *64,000* applications have been analyzed to detect privacy leakage.

**Limitations:** The study suffers from following limitations:

- The study only considers API calls. API calls are not the only source of communication between ad libraries and apps. Communication can also be performed through direct manipulation of class variables, shared memory etc.
- In case of obfuscation, where method names are altered, this method is not capable of identifying privacy related API calls.
- The libraries which acts as a intermediate to transfer information between an app and its library are known as Ad mediation libraries. They are not considered in this study.
- It also omits small libraries which can also be a source of privacy leakage.

#### 7.4.2. Analysis of communication between colluding applications [27]

**Objective:** This paper focuses on evaluating two major channels used for collusion named covert and overt channels. This work aims to quantify the severity of app collusion attack threats by computing throughput, bit-error rate and expected synchronization for every channel.

**Methodology:** Covert and overt channels are implemented by executing them on Nexus one or Samsung Galaxy S smartphones. 5 overt channels are implemented viz. Shared Preferences, Internal Storage, Broadcast Intents, System Logs and UNIX Socket communication. 9 covert channels are implemented which viz. Single and Multiple Settings, Types of Intents, Automatic Intents, Threads Enumeration, UNIX Socket discovery, Free Space on file system, Reading /proc/stat, Timing Channels and Processor Frequency. Experiments are performed to calculate throughput (rate of transmitting data through a channel), stealth (difficulty in identifying a channel), bit-error rate (error-occurrence rate in transmission) and required synchronization (timing constraint between source and sink). In tests 4, 8 and 135 byte data is transmitted from source to sink. During transmission, if the channel under consideration is found open, the information is logged. For the channels which requires synchronization between source and sink, a synchronization protocol is applied on them. This protocol reduces noise and starts measurement on both the ends at the same time.

This paper also proposes a collusion detection approach named *Black-Box* analysis technique. This technique administers a data monitor between applications on the device. The monitor tracks and stores the data used and transmitted by an app to the colluding app. The paper claims that the existing analysis tools such as TaintDroid and XmanDroid failed to detect most of these channels. TaintDroid detects only 2 out of 5 overt channels and 0 out of 9 covert channels. XmanDroid detects 4 out of 5 overt channels and 6 out of 9 covert channels.

**Limitations:** The approach suffers from the following limitations:

- The proposed black-box approach is very preliminary, it misses many communications.
- Data monitoring cannot handle obfuscation, reflection and encryption.
- It cannot handle complex string analysis that can be used by any of the channels.

#### *7.5. Comparison among state of art approaches*

Researchers have proposed various approaches for intra and inter-app analysis varying from static [62][73], dynamic [79] to policy enforcement [9][91] based techniques. In Section 6, we explained these techniques and Section 7 presents research pieces that rely on these techniques. Table 1 summarizes each proposed approach under different criteria: (1) handled components, (2) handled Intents, (3) examines native code or not, (4) resolves reflection or not, (5) works on which code level, (6) conducts intra or inter app analysis and (7) availability of the tool. We believe this helps the reader to examine all the differences in one glance.

Most of the proposed approaches handle Android components viz. Activities(A), Services(S) and Receivers(R), whereas, Content Providers(C) are not handled by [10, 20, 22, 32, 100, 109] as shown in column (1) of the table. These approaches consider only Intents as a medium of communication. To access content providers, unique resource identifier (URI) does not use the Intent. Therefore, Intent specific approaches fails to handle content providers. In particular, there are two approaches [108] and [123] that are not handling any components other than activities. In [108], the authors have mentioned that their approach can be similarly extended for other components whereas [123] have built a prototype on activities and it is available commercially as a cloud service. In future the authors of [123] may extend their approaches to handle all the other components.

There are broadly two types of Intents viz. Implicit(I) and Explicit(E). All the proposed approaches can handle communication through Intents as they are the most popular medium of communication used in Android as shown in column (2) of the table. Although there is one approach [109] that is not considering implicit Intent. The reason narrated by the authors of [109] is that they do not want to increase false positives. In case of implicit Intent the target is not fixed. If there are multiple receivers then at the run-time one of the receivers is chosen.

Column (3) of table 1 presents the capability of proposed approaches to handle native code. Native code refers to the code written in C/C++ and used by Android app libraries for low-level interactions with the underlying Linux kernel. Native code runs directly on the processor and hence not included in Dalvik executable that runs in Dalvik virtual machine. Almost all the approaches convert dex into some intermediate representation (IR) language but native code is not get converted into IR and hence, cannot be handled by many tools. However, Flowdroid [73] can handle very limited native calls as they defined some explicit rules for common invocation of native calls present in Java. Tools like [15, 18, 108] leverage Flowdroid for analysis and therefore can handle native calls partially.

The proposed approaches based on the their ability to resolve reflection is depicted in column (4) of the table. Reflection is a language’s ability to inspect and dynamically call classes, methods, attributes, etc. at runtime. It is a dynamic phenomenon and hence it is very difficult for any static approach to handle it. Dynamic analysis approaches are needed to capture related runtime behaviour features to resolve reflection. If an API is called through reflection, it is passed as a parameter and hence become invisible for detection tools. Although some static tools like [14, 15, 18] can handle reflection partially meaning if the API calls are string constants, then they may be revealed otherwise if they are called through variable where it is obfuscated or encrypted, these tools cannot resolve such reflected calls.

Analysis tools based on the used intermediate representation (IR) for anal-

Proposed Approaches	Components Handled (1)	Intents Handled (2)	Native Code (3)	Reflection (4)	Code Level (5)	Inter-app Analysis (6)	Availability (7)
Static	MR-Droid [17]	$\langle A \ S \ R \ C \rangle$	No	No	Java Bytecode	Yes	-
	Detecting Inter-App Information Leakage Paths [32]	$\langle A \ S \ R \ - \rangle$	No	No	Java Bytecode & Smali	Yes	-
	Towards Automated Android App Collusion Detection [100]	$\langle A \ S \ R \ - \rangle$	No	No	Smali	Yes	-
	ICC Map [4]	$\langle A \ S \ R \ C \rangle$	No	No	Jimple/Source code	Yes	-
	IccTA [15]	$\langle A \ S \ R \ C \rangle$	Yes*	Yes*	Jimple	Yes <sup>+</sup>	Open-Source
	Permission Flow [13]	$\langle A \ S \ R \ C \rangle$	No	No	Java Bytecode	No	-
	FUSE[14]	$\langle A \ S \ R \ C \rangle$	No	Yes*	Java Bytecode	Yes	Commercial
	AmanDroid [22]	$\langle A \ S \ R \ - \rangle$	No	No	Java Bytecode	No	Open-Source
	DidFail [108]	$\langle A \ - \ - \ - \rangle$	Yes*	No	Java Bytecode	Yes	Open-Source
	ComDroid [10]	$\langle A \ S \ R \ - \rangle$	No	No	Java Bytecode	No	Open-Source
Dynamic	IntelliDroid [109]	$\langle A \ S \ R \ - \rangle$	Yes	Yes	Java Bytecode	No	-
	IntentDroid [117]	$\langle A \ - \ - \ - \rangle$	Yes	Yes	Java Bytecode	Yes	Commercial
	TaintDroid [2]	$\langle A \ S \ R \ C \rangle$	Yes	Yes	Java Bytecode	No	Open-Source
	DIALDroid [18]	$\langle A \ S \ R \ C \rangle$	Yes*	Yes*	Java Bytecode	Yes	Open-Source
	Intersection Automata Based Model for Android Application Collusion [20]	$\langle A \ S \ R \ - \rangle$	No	No	Java Bytecode	Yes	-
	FlaskDroid [94]	$\langle A \ S \ R \ C \rangle$	No	No	-	Yes	-
	XManDroid [26]	$\langle A \ S \ R \ C \rangle$	Yes	Yes	-	Yes	-

- A: Activity, S: Service, R: Broadcast Receiver, C: Content Provider
- E: Explicit Intent, I: Implicit Intent
- Yes\*: The details are explained in section 7.5
- Yes<sup>+</sup>: If it is used with APKCombiner

Table 1: Comparison among state of the art approaches



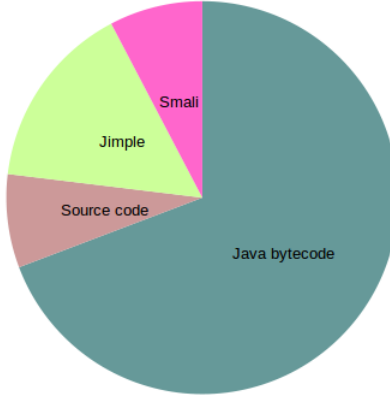


Figure 3: Intermediate Representation (IR) used for analysis

ysis are classified in column (5) of the table. Android APK file is converted to some IR prior to the analysis. There are four code levels on which analysis can be performed viz. Java source code, Java bytecode, Jimple and Smali. Java source code can be analyzed because applications are written in Java language. However, source is available only if the apps are open-sourced or self developed. Android apps are compiled into Dalvik bytecode called Dex, which is executed in Dalvik virtual machine. For analysis Dalvik should be converted to Java bytecode. This can be done by many APK to Jar converters like dex2jar [124], dex2jar [125] and Dare [98]. Jimple is a simplified version of Java bytecode. It is a typed 3-address intermediate representation. It is used by Soot [104] which is a popular static analysis framework for Java. Dexpler [126] is a plugin for the Soot framework that translates Dalvik bytecode to Jimple. Smali is another IR used by very popular reverse engineering tool developed by Google named Apk-tool [106]. Figure 3 shows that Java bytecode is used by most of the approaches, as Java source is generally not available and Jimple and Smali are tool specific representations.

Apart from this, Figure 4, shows the distribution of different app repositories used by the state of art approaches.

This brings us to the following conclusions:

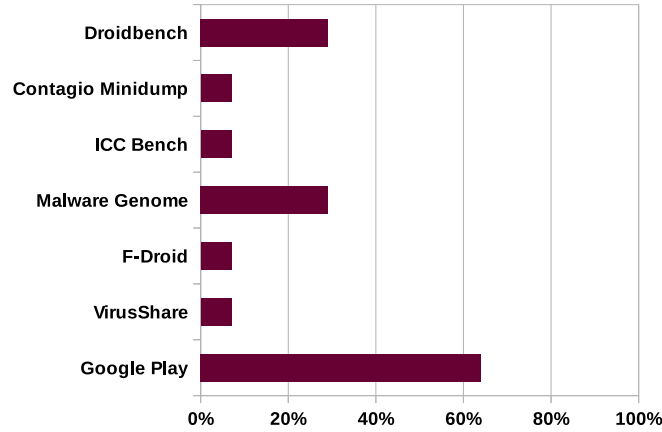


Figure 4: App repositories used for analysis

- Intents are considered to be most commonly used ICC communication channels. Therefore data sharing through Content Providers may become an attractive target for attacker to exploit.
- Native Calls and Reflection are handled by very few researchers and therefore can be used by malware developers to evade the maliciousness of their code.
- There are two ways to register Broadcast Receiver viz. static and dynamic. If any static approach that cannot handle reflection but can handle dynamic Broadcast Receiver only works if, registration of Receiver is not reflected.
- Most of the approaches work on bytecode and therefore maliciousness posed by native code remains untouched.
- Approaches that analyzes only single app cannot detect inter app leakages.

#### 7.6. *Lessons Learned*

In this subsection, we mention some recommendations on which future research needs to focus to stay ahead of smart malware or vulnerabilities.

Analysis approaches should be combination of static, dynamic and policy based analysis techniques to overcome the limitations of all. Other communication channels like Content Providers, Shared Preferences, AIDL etc. should be considered while analysis. Analysis methods should cover different code forms to compensate the losses posed by converting dex to any IR language. Also, analysis methods should consider native code level. There is an urgent need to develop engines that can resolve reflected calls beforehand, so that they are examined during analysis. To capture compositional vulnerabilities like collusion, analysis approaches should examine multiple apps simultaneously. Many proposed approaches are either commercial or not available for public use. It is very difficult for the researchers to evaluate and compare their findings and results. Therefore, it is highly recommended that the tool implemented from the proposed approach along with the tested dataset (if self developed) should be available for free.

To prevent unintentional collusion i.e. to avoid the situations when some malicious app can exploit the benign app and use it for collusion, developers need to take special care while signing applications with same certificate and keep their certificate private. Developers should also protect the component that is sending sensitive information to the outside world with specific permissions.

## 8. Conclusions

Android is a modern operating system for smartphones with expanding market share. The main security mechanisms of Android are application sandboxing, application signing, and a permission framework to control access to (sensitive) resources. Android's security framework exhibits serious shortcomings: The burden of approving application permissions is delegated to the end-user who in general does not care much about the impact of prompted permissions on his privacy and security. Hence, malware can be installed on end-user devices such as unauthorized sending of text messages or leaking of sensitive data in the background of running games. With the growing use of Android and

the awareness of its security vulnerabilities, a number of research contributions have led to tools for the intra-app analysis of Android apps. Unfortunately, these state of the art approaches, and the associated tools, have long left out the security flaws that arise across the boundaries of single apps, in the interaction between several apps. We provide in this survey a definition of the collusion in Android, the major security risks on Android, as well as a summary of the main tools for detecting inter and intra app analysis. The collusion attack is worth investigation.

This survey provides a comprehensive assessment of the strengths and shortcomings of state-of-art approaches. It provides a platform to researchers and practitioners towards proposing technique that can analyze multiple apps simultaneously to detect Android app collusion attacks.

## 9. Acknowledgments

This study has been carried out with financial support from the Department of Information Technology, Government of India Project Grant ‘Security Analysis Framework for Android Platform’ and the French National Research Agency (ANR) of the French State in the frame of the ‘Investments for the future’ Programme IdEx Bordeaux - CPU (ANR-10-IDEX-03-02).

## References

## References

- [1] IDCReport: Press Release, <http://www.idc.com/getdoc.jsp?containerId=prUS41425416>, [Online; accessed 18-June-2016].
- [2] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, ACM Transactions on Computer Systems (TOCS) 32 (2) (2014) 5.

- [3] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, Apkcombiner: Combining multiple android apps to support inter-app analysis, in: *ICT Systems Security and Privacy Protection*, Springer, 2015, pp. 513–527.
- [4] K. O. Elish, D. D. Yao, G. R. Barbara, On the need of precise inter-app icc classification for detecting android malware collusions, in: *Proceedings of the Security and Privacy Workshops*, 2015, pp. 116–127.
- [5] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, D. S. Wallach, Quire: Lightweight provenance for smart phone operating systems., in: *USENIX Security Symposium*, 2011, p. 24.
- [6] L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang, Chex: statically vetting android apps for component hijacking vulnerabilities, in: *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 229–240.
- [7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, B. Shastry, Practical and lightweight domain isolation on android, in: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2011, pp. 51–62.
- [8] S. Shekhar, M. Dietz, D. S. Wallach, Adsplitt: Separating smartphone advertising from applications., in: *USENIX Security Symposium*, 2012, pp. 553–567.
- [9] W. Enck, Defending users against smartphone apps: Techniques and future directions, in: *Information Systems Security*, Springer, 2011, pp. 49–70.
- [10] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ACM, 2011, pp. 239–252.

- [11] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, X. Wang, Soundcomber: A stealthy and context-aware sound trojan for smart-phones., in: NDSS, Vol. 11, 2011, pp. 17–33.
- [12] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: Attacks and defenses, in: In 20th USENIX Security Symposium, 2011.
- [13] D. Sbirlea, M. Burke, S. Guarnieri, M. Pistoia, V. Sarkar, Automatic detection of inter-application permission leaks in android applications, IBM Journal of Research and Development 57 (6) (2013) 10:1–10:12. doi : 10.1147/JRD.2013.2284403.
- [14] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, L. Casburn, Multi-app security analysis with fuse: Statically detecting android app collusion, in: Proceedings of the 4th Program Protection and Reverse Engineering Workshop, ACM, 2014, p. 4.
- [15] L. Li, A. Bartel, T. F. D. A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Iccta: detecting inter-component privacy leaks in android apps, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015), 2015.
- [16] A. Sadeghi, H. Bagheri, S. Malek, Analysis of android inter-app security vulnerabilities using covert, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, 2015, pp. 725–728.
- [17] F. Liu, H. Cai, G. Wang, D. Yao, K. Elish, B. Ryder, Mr-droid: A scalable and prioritized analysis of inter-app communication risks, in: Proceedings of the Mobile Security Technologies (MoST), in conjunction with IEEE Symposium on Security and Privacy, San Jose, CA, May 2017.
- [18] A. Bosu, F. Lio, D. Yao, G. Wang, Collusive data leak and more: Large-scale threat analysis of inter-app communications, in: Proceedings of the

ACM Asia Conference on Computer and Communications Security (ASIACCS), 2017.

- [19] H. Chen, D. He, S. Zhu, J. Yang, Toward detecting collusive ranking manipulation attackers in mobile app markets, in: Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS), 2017.
- [20] S. Bhandari, V. Laxmi, A. Zemmari, M. S. Gaur, Intersection automata based model for android application collusion, in: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2016, pp. 901–908.
- [21] S. Bhandari, R. Gupta, V. Laxmi, M. S. Gaur, A. Zemmari, M. Anikeev, Draco: Droid analyst combo an android malware analysis framework, in: Proceedings of the 8th International Conference on Security of Information and Networks, ACM, 2015, pp. 283–289.
- [22] F. Wei, S. Roy, X. Ou, et al., Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1329–1341.
- [23] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, Drebin: Effective and explainable detection of android malware in your pocket., in: NDSS, The Internet Society, 2014.  
URL <http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ArpSHGR14>
- [24] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, M. Rinard, Information-flow analysis of android applications in droidsafe, in: Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society, 2015.

- [25] A. P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A survey of mobile malware in the wild, in: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2011, pp. 3–14.
- [26] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, Xmandroid: A new android evolution to mitigate privilege escalation attacks, *Technische Universität Darmstadt*, Technical Report TR-2011-04.
- [27] C. Marforio, H. Ritzdorf, A. Francillon, S. Capkun, Analysis of the communication between colluding applications on modern smartphones, in: *Proceedings of the 28th Annual Computer Security Applications Conference*, ACM, 2012, pp. 51–60.
- [28] T. Markmann, D. Gessner, D. Westhoff, Quantdroid: Quantitative approach towards mitigating privilege escalation on android, in: *Communications (ICC), 2013 IEEE International Conference on*, IEEE, 2013, pp. 2144–2149.
- [29] Z. Fang, W. Han, Y. Li, Permission based android security: Issues and countermeasures, *computers & security* 43 (2014) 205–218.
- [30] H. Bagheri, A. Sadeghi, J. Garcia, S. Malek, Covert: Compositional analysis of android inter-app permission leakage, *IEEE transactions on Software Engineering* 41 (9) (2015) 866–886.
- [31] D. Ocateau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, Y. Le Traon, Combining static analysis with probabilistic models to enable market-scale android inter-component analysis, in: *ACM SIGPLAN Notices*, Vol. 51, ACM, 2016, pp. 469–484.
- [32] S. Bhandari, F. Herbreteau, V. Laxmi, A. Zemmari, P. S. Roop, M. S. Gaur, Poster: Detecting inter-app information leakage paths, in: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ACM, 2017, pp. 908–910.



- [33] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, Android taint flow analysis for app sets, in: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, ACM, 2014, pp. 1–6.
- [34] <https://en.wikipedia.org/wiki/Android-operating-system>, [Online; accessed 20-Feb-2016].
- [35] <http://developer.android.com/ndk/index.html>, [Online; accessed 20-Feb-2016].
- [36] <https://source.android.com/devices/tech/dalvik/index.html>, [Online; accessed 21-Feb-2016].
- [37] <https://en.wikipedia.org/wiki/Android-application-package>, [Online; accessed 21-Feb-2016].
- [38] <http://developer.android.com/tools/publishing/app-signing.html>, [Online; accessed 01-Jan-2016].
- [39] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, ACM, New York, NY, USA, 2011, pp. 627–638. doi:10.1145/2046707.2046779.  
URL <http://doi.acm.org/10.1145/2046707.2046779>
- [40] <http://developer.android.com/guide/topics/security/permissions.html>, [Online; accessed 21-Feb-2016].
- [41] A. Egners, U. Meyer, B. Marschollek, Messing with android’s permission model, in: Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, TRUST-COM '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 505–514. doi:10.1109/TrustCom.2012.203.  
URL <http://dx.doi.org/10.1109/TrustCom.2012.203>

- [42] D. Barrera, H. G. Kayacik, P. C. van Oorschot, A. Somayaji, A methodology for empirical analysis of permission-based security models and its application to android, in: Proceedings of the 17th ACM conference on Computer and communications security, ACM, 2010, pp. 73–84.
- [43] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: Attacks and defenses., in: USENIX Security Symposium, Vol. 30, 2011.
- [44] K. W. Y. Au, Y. F. Zhou, Z. Huang, D. Lie, Pscout: Analyzing the android permission specification, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, 2012, pp. 217–228.
- [45] D. Sbirlea, M. G. Burke, S. Guarnieri, M. Pistoia, V. Sarkar, Automatic detection of inter-application permission leaks in android applications, IBM Journal of Research and Development 57 (6) (2013) 10–1.
- [46] S. Rasthofer, S. Arzt, E. Lovat, E. Bodden, Droidforce: Enforcing complex, data-centric, system-wide policies in android, in: Availability, Reliability and Security (ARES), 2014 Ninth International Conference on, IEEE, 2014, pp. 40–49.
- [47] <http://developer.android.com/training/articles/security-tips.html>, [Online; accessed 01-Jan-2016].
- [48] W. Enck, M. Ongtang, P. McDaniel, Understanding android security, IEEE security & privacy (1) (2009) 50–57.
- [49] OWASP Mobile Checklist Final 2016 , <https://drive.google.com/file/d/0Bx0Pagp1jPHWYmg3Y3BfLVhMcmc/view>, [Online; accessed 02-march-2016].
- [50] Q. A. Chen, Z. Qian, Z. M. Mao, Peeking into your app without actually seeing it: Ui state inference and novel android attacks., in: USENIX Security, Vol. 14, 2014, pp. 1037–1052.

- [51] L. Li, A. Bartel, J. Klein, Y. Le Traon, Automatically exploiting potential component leaks in android applications, in: Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on, IEEE, 2014, pp. 388–397.
- [52] D. Kantola, E. Chin, W. He, D. Wagner, Reducing attack surfaces for intra-application communication in android, in: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2012, pp. 69–80.
- [53] S. Heuser, M. Negro, P. K. Pendyala, A.-R. Sadeghi, Droidauditor: Forensic analysis of application-layer privilege escalation attacks on android, Tech. rep., Technical report, TU Darmstadt (2016).
- [54] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, B. Shastri, Towards taming privilege-escalation attacks on android., in: NDSS, 2012.
- [55] W. Enck, D. Ocate, P. McDaniel, S. Chaudhuri, A study of android application security., in: USENIX security symposium, Vol. 2, 2011, p. 2.
- [56] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, Automatically securing permission-based software by reducing the attack surface: An application to android, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 274–277.
- [57] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, A. Ribagorda, Evolution, detection and analysis of malware for smart devices, Communications Surveys & Tutorials, IEEE 16 (2) (2014) 961–987.
- [58] A. Armando, G. Costa, A. Merlo, Bring your own device, securely, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, pp. 1852–1858.
- [59] N. Hardy, The confused deputy:(or why capabilities might have been invented), ACM SIGOPS Operating Systems Review 22 (4) (1988) 36–38.

- [60] D. Maier, M. Protsenko, T. Müller, A game of droid and mouse: The threat of split-personality malware on android, *Computers & Security*.
- [61] D. Maier, T. Muller, M. Protsenko, Divide-and-conquer: Why android malware cannot be stopped, in: *Availability, Reliability and Security (ARES)*, 2014 Ninth International Conference on, IEEE, 2014, pp. 30–39.
- [62] D. J. Tan, T.-W. Chua, V. L. Thing, et al., Securing android: a survey, taxonomy, and challenges, *ACM Computing Surveys (CSUR)* 47 (4) (2015) 58.
- [63] A. Bhavani, Cross-site scripting attacks on android webview, *arXiv preprint arXiv:1304.7451*.
- [64] J. Clarke-Salt, *SQL injection attacks and defense*, Elsevier, 2009.
- [65] D. Ceara, M.-L. POTET, G. I. ENSIMAG, L. MOUNIER, Detecting software vulnerabilities-static taint analysis, *Vérimag-Distributed and Complex System Group*, Polytechnic University of Bucharest.
- [66] S. Schmeelk, J. Yang, A. Aho, Android malware static analysis techniques, in: *Proceedings of the 10th Annual Cyber and Information Security Research Conference, CISR '15*, ACM, New York, NY, USA, 2015, pp. 5:1–5:8. doi:10.1145/2746266.2746271.  
URL <http://doi.acm.org/10.1145/2746266.2746271>
- [67] The AndroidManifest.xml File, <http://lyle.smu.edu/~coyle/cse7392mobile/handouts/s01.The%20AndroidManifest.pdf>, [Online; accessed 21-May-2015].
- [68] H. S. Karlsen, E. R. Wognsen, M. C. Olesen, R. R. Hansen, Study, formalisation, and analysis of dalvik bytecode, in: *Informal proceedings of The Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2012)*, Citeseer, 2012.

- [69] B. Scholz, C. Zhang, C. Cifuentes, User-input dependence analysis via graph reachability, in: Proceedings of Eighth IEEE International Working Conference on Source Code Analysis and Manipulation.
- [70] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1995, pp. 49–61.
- [71] Z. Yang, M. Yang, Leakminer: Detect information leakage on android with static taint analysis, in: Software Engineering (WCSE), 2012 Third World Congress on, IEEE, 2012, pp. 101–104.
- [72] Analyzing Data flow, <https://www.jetbrains.com/help/idea/2016.1/analyzing-data-flow.html>, [Online; accessed 25-April-2016].
- [73] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: ACM SIGPLAN Notices, Vol. 49, ACM, 2014, pp. 259–269.
- [74] M. Das, S. Lerner, M. Seigle, Esp: Path-sensitive program verification in polynomial time, in: ACM Sigplan Notices, Vol. 37, ACM, 2002, pp. 57–68.
- [75] D. Callahan, The program summary graph and flow-sensitive interprocedural data flow analysis, Vol. 23, ACM, 1988.
- [76] E. M. Myers, A precise inter-procedural data flow algorithm, in: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1981, pp. 219–230.
- [77] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis.

- [78] A. J. Spyridi, A. A. Requicha, Accessibility analysis for the automatic inspection of mechanical parts by coordinate measuring machines, in: Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on, IEEE, 1990, pp. 1284–1289.
- [79] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, S. Ioannidis, Rage against the virtual machine: hindering dynamic analysis of android malware, in: Proceedings of the Seventh European Workshop on System Security, ACM, 2014, p. 5.
- [80] Droidbox, <http://code.google.com/p/droidbox/>;, [Online; accessed 10-Oct-2015].
- [81] M. Backes, S. Bugiel, S. Gerling, P. von Styp-Rekowsky, Android security framework: Enabling generic and extensible access control on android, arXiv preprint arXiv:1404.1395.
- [82] V. Costamagna, C. Zheng, Artdroid: A virtual-method hooking framework on android art runtime, Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS) (2016) 24–32.
- [83] S. Gupta, P. Pratap, H. Saran, S. Arun-Kumar, Dynamic code instrumentation to detect and recover from return address corruption, in: Proceedings of the 2006 international workshop on Dynamic systems analysis, ACM, 2006, pp. 65–72.
- [84] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, Y.-M. Wang, Fine-grained power modeling for smartphones using system call tracing, in: Proceedings of the sixth conference on Computer systems, ACM, 2011, pp. 153–168.
- [85] P. Machado, J. Campos, R. Abreu, Mzoltar: automatic debugging of android applications, in: Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, ACM, 2013, pp. 9–16.

- [86] L.-K. Yan, H. Yin, Droidscape: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis., in: USENIX security symposium, 2012, pp. 569–584.
- [87] M. Sun, M. Zheng, J. Lui, X. Jiang, Design and implementation of an android host-based intrusion prevention system, in: Proceedings of the 30th Annual Computer Security Applications Conference, ACM, 2014, pp. 226–235.
- [88] G. Sarwar, O. Mehani, R. Boreli, D. Kaafar, On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices, Nicta.
- [89] D. Amalfitano, A. R. Fasolino, P. Tramontana, A gui crawling-based technique for android mobile application testing, in: Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, IEEE, 2011, pp. 252–261.
- [90] P. Szor, The art of computer virus research and defense, Pearson Education, 2005.
- [91] W. Enck, M. Ongtang, P. McDaniel, Mitigating android software misuse before it happens.
- [92] R. Xu, H. Saïdi, R. J. Anderson, Aurasium: practical policy enforcement for android applications., in: USENIX Security Symposium, Vol. 2012, 2012.
- [93] M. Conti, V. T. N. Nguyen, B. Crispo, Crepe: Context-related policy enforcement for android, in: International Conference on Information Security, Springer, 2010, pp. 331–345.
- [94] S. Bugiel, S. Heuser, A.-R. Sadeghi, Flexible and fine-grained mandatory access control on android for diverse security and privacy policies., in: Usenix security, 2013, pp. 131–146.

- [95] OWASP, [https://www.owasp.org/index.php/Access\\_Control\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Access_Control_Cheat_Sheet), [Online; accessed 10-March-2017].
- [96] A. Ubale Swapnaja, G. Modani Dattatray, S. Apte Sulabha, Analysis of dac mac rbac access control based models for security, *Analysis* 104 (5).
- [97] DroidBench 3.0, <https://github.com/secure-software-engineering/DroidBench/tree/develop>, [Online; accessed 02-February-2017].
- [98] D. Ocateau, S. Jha, P. McDaniel, Retargeting android applications to java bytecode, in: *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, ACM, 2012, p. 6.
- [99] D. Ocateau, D. Luchaup, M. Dering, S. Jha, P. McDaniel, Composite constant propagation: Application to android inter-component communication analysis, in: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.
- [100] I. M. Asavoa, J. Blasco, T. M. Chen, H. K. Kalutarage, I. Muttik, H. N. Nguyen, M. Roggenbach, S. A. Shaikh, Towards automated android app collusion detection, in: *Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS16*, London, UK, 06-April-2016.
- [101] K. O. Elish, D. Yao, B. G. Ryder, User-centric dependence analysis for identifying malicious mobile apps, in: *Workshop on Mobile Security Technologies*, 2012.
- [102] K. O. M. Elish, User-intention based program analysis for android security.
- [103] J. Oberheide, C. Miller, Dissecting the android bouncer, *SummerCon2012*, New York.
- [104] P. Lam, E. Bodden, O. Lhoták, L. Hendren, The soot framework for java program analysis: a retrospective, in: *Cetus Users and Compiler Infrastructure Workshop (CETUS)*, 2011.



- [105] D. Oteau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. Le Traon, Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis, in: USENIX Security, 2013.
- [106] APKTool, <http://ibotpeaches.github.io/Apktool/>, [Online; accessed 21-March-2016].
- [107] L. O. Andersen, Program analysis and specialization for the c programming language, Ph.D. thesis, University of Copenhagen (1994).
- [108] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, Android taint flow analysis for app sets, in: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, SOAP '14, ACM, New York, NY, USA, 2014, pp. 1–6. doi: 10.1145/2614628.2614633.  
URL <http://doi.acm.org/10.1145/2614628.2614633>
- [109] M. Y. Wong, D. Lie, Intellidroid: A targeted input generator for the dynamic analysis of android malware, Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS 2016).
- [110] K. Tam, S. J. Khan, A. Fattori, L. Cavallaro, Copperdroid: Automatic reconstruction of android malware behaviors., in: Proceedings of the Network and Distributed System Security Symposium (NDSS15), San Diego, California, USA.
- [111] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, andromaly: a behavioral malware detection framework for android devices, Journal of Intelligent Information Systems 38 (1) (2012) 161–190.
- [112] <http://code.google.com/p/xml-apk-parser/>, [Online; accessed 16-Nov.-2015].
- [113] <http://wala.sourceforge.net>, [Online; accessed 11-Jan.-2016].

- [114] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
- [115] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: Security and Privacy (SP), 2012 IEEE Symposium on, IEEE, 2012, pp. 95–109.
- [116] D. Barrera, J. Clark, D. McCarney, P. C. van Oorschot, Understanding and improving app installation security mechanisms through empirical analysis of android, in: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2012, pp. 81–92.
- [117] R. Hay, O. Tripp, M. Pistoia, Dynamic detection of inter-application communication vulnerabilities in android, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, 2015, pp. 118–128.
- [118] A. Bosu, F. Liu, D. D. Yao, G. Wang, Collusive data leak and more: Large-scale threat analysis of inter-app communications, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM, 2017, pp. 71–85.
- [119] Malware Genome Project, <http://www.malgenomeproject.org>, [Online; accessed 20-April-2015].
- [120] Contagio Minidump, <http://contagiomindump.blogspot.com/>, [Online; accessed 23-April-2015].
- [121] T. Book, D. S. Wallach, A case of collusion: A study of the interface between ad libraries and their apps, in: Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices, ACM, 2013, pp. 79–86.

- [122] Dedexer, <http://dedexer.sourceforge.net/>, [Online; accessed 10-May-2016].
- [123] R. Hay, O. Tripp, M. Pistoia, Dynamic detection of inter-application communication vulnerabilities in android, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, ACM, New York, NY, USA, 2015, pp. 118–128. doi:10.1145/2771783.2771800. URL <http://doi.acm.org/10.1145/2771783.2771800>
- [124] Dex2Jar, <https://github.com/pxb1988/dex2jar>, [Online; accessed 10-May-2015].
- [125] D. Octeau, W. Enck, P. McDaniel, The ded decompiler, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0140-2010.
- [126] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, Dexpler: converting android dalvik bytecode to jimple for static analysis with soot, in: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, ACM, 2012, pp. 27–38.