DIGITALCOMMONS
@WAYNESTATE

Wayne State University

Wayne State University Theses

January 2019

An Empirical Study On Deterministic Collusive Attack Using Inter Component Communication In Android Applications

Tanzeer Hossain
Wayne State University

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_theses

Part of the Databases and Information Systems Commons

Recommended Citation

Hossain, Tanzeer, "An Empirical Study On Deterministic Collusive Attack Using Inter Component Communication In Android Applications" (2019). *Wayne State University Theses*. 754.
https://digitalcommons.wayne.edu/oa_theses/754

This Open Access Thesis is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Theses by an authorized administrator of DigitalCommons@WayneState.

AN EMPIRICAL STUDY ON DETERMINISTIC COLLUSIVE ATTACK USING INTER COMPONENT COMMUNICATION IN ANDROID APPLICATIONS

by

TANZEER HOSSAIN

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2019

MAJOR: COMPUTER SCIENCE

Approved By:

____

Advisor   Date

DEDICATION

This thesis is dedicated to Allah and my parents, for all their love, patience, kindness and support.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Amiangshu Bosu, who generously offered his always wise guidance. I am very thankful for his support, his patience, and his time. Without his guidance and persistent help, this thesis is not possible. I also want to thank my committee members for their time and support.

TABLE OF CONTENTS

Dedication ii

Acknowledgements iii

List of Tables vii

List of Figures viii

Chapter 1: INTRODUCTION 1

Chapter 2: BACKGROUND 4

2.0.1 Android Components 4

2.0.2 Intents 4

2.0.3 Type of Intents 4

Explicit Intents 4

Implicit Intents 5

2.0.4 Intent Resolution 5

Intent Filter 6

2.0.5 Content Provider 7

2.0.6 Broadcast 7

2.0.7 Source and Sink 7

2.0.8 ICC Entry and Exit Points 8

2.0.9 Effects of Security flaws in Android Application 8

Activity Hijacking 8

Service Hijacking 8

2.0.10 Privilege escalation 9

2.0.11 Collusive Data Leak 9

Chapter 3: Threat Model 10

3.0.1 Type I: Explicit Intents 11

3.0.2 Type II: Implicit Intents with Custom Actions 12

3.0.3 Type III: Implicit Intents with Custom Category 13

3.0.4 Type IV: Implicit Intents with Custom URI 14
3.0.5 Type V: Custom Provider 15
3.0.6 Type VI: Custom Broadcast 16
Chapter 4: Research Method 18
4.0.1 Tool Selection/Development 18
4.0.2 Data Collection 18
4.0.3 Feature Aggregation 19
4.0.4 Model Building 19
4.0.5 Evaluation 20
Chapter 5: RESULTS 21
5.0.1 Statistics of Security Threats in Android 22
5.0.2 Permission Leaks 24
5.0.3 Top Source and Sinks 25
Chapter 6: CASE STUDIES 26
6.0.1 Deterministic Collusive Data Leak 26
6.0.2 Explicit Collusion: 26
Location Data Leak 26
6.0.3 Collusive Data Leak Using Libraries 27
Deterministic collusion using Custom Provider 27
6.0.4 Same Developer Collusion 28
6.0.5 Privilege Escalation 28
Location Escalation 28
DeviceID Escalation 29
Chapter 7: DISCUSSION 30
Chapter 8: RELATED WORK 33
Chapter 9: THREATS TO VALIDITY 35
Chapter 10: CONCLUSION 38

References 39
Abstract 42
Autobiographical Statement 43

LIST OF TABLES

Table. 4.1 Categoty and Action Strings 19

Table. 4.2 Custom and Non Custom Action Strings 20

Table. 5.1 Demography 22

Table. 5.2 Top source and sinks 23

Table. 5.3 Top leaked permission 23

Table. 5.4 Collusion in Android using ICC 23

Table. 5.5 Statistic on ICC exit and Entry Leaks 24

LIST OF FIGURES

Figure. 2.1 Visualization of Intent Resolution 6

CHAPTER 1 INTRODUCTION

Android covers 80% of mobile market share. One of the major driving force behind the triumph of Android in mobile market is the ease of development, freedom of customization and overall huge number of developer community. But, these features comes at a cost. Hugely customizable features provided by android SDK creates security and privacy loophole in the total app development ecosystem. As a result, all android app markets have seen myriad amount of malicious and junk applications. To encounter this issue, Android has also evolved. It has incorporated new permission model, updated vetting mechanisms in app markets(e.g., [1], [2],[9],[16]). But, malicious applications also evolved to nullify these effects and eventually has become more and more sophisticated.

In recent times, researchers have provided evidence of collusive android applications where two or more applications can team up to conduct a malicious act (e.g., [5], [7],[13],[12]. This applications are particularly dangerous because they are seemingly benign from a single application perspective. Vetting mechanism or malware detection mechanism for single applications will label it as benign but it can be a serious security and privacy threat when it is in cooperation with other applications. So, pairwise detection mechanism is necessary to detect such malicious applications. DialDroid[6] and (IccTA+ApkCombiner and COVERT)[13] are such two efforts from research community to identify collusive malicious applications. So far, researchers are able to identify collusive attacks using different techniques but there are still few questions that need attention. Prior research were able to identify automated collusive channels between different android applications but the degree of developers involvement to establish such channels are still in the dark.

Android applications normally have many open interfaces. Though it is suggested by research community to follow security aware coding practices, it is highly unlikely that developers community follow them strictly. So, it is difficult to say weather

intent base inter app communication channels identified by static analysis actually intentionally designed by the developers. If we can identify developer's involvement in establishing collusive channels, we can understand their motivation, their point of interest and gather valuable insights to design systems to vet these malicious activities.

Primary purpose of this paper is to develop a methodology to identify developers involvement in collusive attack in two different android applications. To achieve our goal, we take an empirical approach. At first we collect most popular android applications from Google play and virusshare. Then we use state-of-the-art static analysis tool to find data flow path in each application. We store these data in highly normalized MySql database. Then we build a query model based on internal mechanism of intent based communication. Based on our model, we run queries to store collusive channels that required direct developer's effort to build . Then we do different empirical and statistical analysis to understand the motivation and nature of these collusive channels.

Our results show that there are myriad amount of collusive channels are not established just by coincidence rather required developer's effort. However, it is necessary to mention that not all of these collusive channels are malicious. Some may be created because of poor coding practices, some by just using same sdks across different apps or just for using same code base for building different applications. We summarize our contribution as follows:

1. We provide a novel methodology to deterministically identify developers involvement in establishing malicious channels between two different apps. We describe a model based on internal implementation of inter app communication and present empirical evidence on the accuracy of our model. We open sourced all our data we have gathered to build our model.

2. We extend the state-of-the art open source collusive data flow analysis tool Dial-

droid to increase source-sink discovery rate to a great extend specially for large real world apps. We open sourced our code, total implementation of our methodology.

3. We analyzed 1,36,012 applications using our updated tool using cluster computers and made our generated data open source. We also report various interesting statistics regarding malicious exploitation of inter app communication channels.

CHAPTER 2 BACKGROUND

2.0.1 Android Components

Android application has four major components. Activity, Service, Broadcast Receivers and Content Providers. These four components communicate with each other with a message passing mechanism called Intent. These communication can be between components of same or different apps. In this paper, we focus on communication between components of different apps. Every android application has a *AndroidManifest.xml* file which includes all the components, permissions and other necessary details for that specific application.

2.0.2 Intents

Intents are messaging objects in android which are used to request another application component to perform certain actions. During this communication, one app can bind additional information to pass it to other application. App component can be of different types. For example, one app can use intents to start a new activity (new screen) or a new service (to run some longer running task in background). Intent has three major parameter string. They are *category*, *data* and *action*. These parameters are used to choose the right component in response to a request. Android use a technique called *Intent Resolution* for this purpose.

2.0.3 Type of Intents

There are two major type of intents. Explicit intents and Implicit intents.

Explicit Intents

Explicit intent specify which component to receive the intended intent. It is done by explicitly specifying the package name or fully qualified component class name. If explicit intents are used, only the targeted component is able to receive the intent. In *AndroidManifest.xml* all application components are declared. For each component,

AndroidManifest.xml a parameter called *exported* can be added. If this parameter is set to *false* other applications can not directly invoke this component.

Implicit Intents

Implicit intents are intents where targeted components are not specified. Rather, few parameters of the intent are added. Each component can registers for specific type of intent via *intent filter*. Based on the properties specified in the intent, operating system selects a set of component to complete an action.

```
Intent intent= new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.java.com"));
startActivity(intent);
```

In the above example, intent request for a component that can show a HTML page. Intent also specify the url of the page to be loaded. There can be more than one application who can perform this request. In that case, Operating system will present a set components that can perform the action to user.

2.0.4 Intent Resolution

As implicit intents do not have a specific receiver, operating system has to go through a process called Intent resolution to determine the best suited receiver component for the intent. Android uses three pieces of string from intent object for this purpose. They are *action*, *category* and *type*. If *action*, *category* or *type* parameters exist in the intent object, receiver component must include them in their intent filters to be a valid candidate as a receiver of the intent. If there are multiple matching component found for an intent object, operating system pops up a window and leaves the choice of choosing an component to the user. Figure 2.1 shows visualization of intent resolution technique (source: developers.android.com) in android.

resolution.PNG resolution.PNG

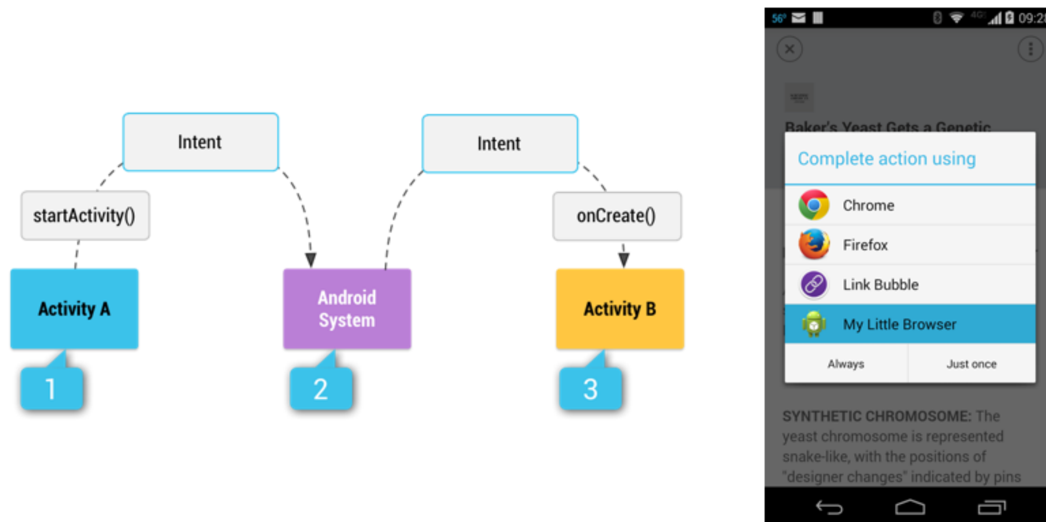

Figure 2.1: Visualization of Intent Resolution

Intent Filter

An intent filter is an expression in an application's *AndroidManifest.xml* file that specifies the type of intents that the component would like to listen to.

```
<activity android:name="ShareActivity">
<intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
</intent-filter>
</activity>
```

In this example, *ShareActivity* is an application component. It only listens to the intents that request to send plain text.

2.0.5 Content Provider

Content provider is a secure way to connect data of one process to code of another process. Content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. Android provides special security guidelines for content providers to make data sharing secure and structured. A provider application can specify permissions that a caller application requires to request data from it. Caller application must include them in *¡uses-permission¿* tag in it's *AndroidMenifest.xml* file in order to successfully retrieve data from provider application.

2.0.6 Broadcast

Broadcast is a messaging system where android system or other applications can send and receive messages. A broadcast message is normally fired when an event of interest takes place. An application component can register for listening different type of messages using *¡intent-filter¿* in its *AndroidMenifest.xml* file. An application can send broadcast messages by customizing an intent by *sendBroadcast* call. For example, a camera app after taking a picture can send a broadcast message about the event. A gallery app upon receiving the message can update is database.

2.0.7 Source and Sink

Sources are Android sdk provided methods that returns sensitive data. Sinks are the methods that creates interfaces for other application to red the data. For our experiment, we use SuSi[3] project as baseline for determining sensitive sources and sinks. A sensitive source can be *getLastKnownLocation(), getDeviceId()* to retrieve any sensitive data. A sensitive sink can be *java.io.OutputStream* or *java.net.URL* that create interfaces for other application to read that data.

2.0.8 ICC Entry and Exit Points

In any icc communication there are two parties. Sender and Receiver. Sender sends information using communication channels and receiver app receives and parses that information. ICC exit points send a data to another applications using API calls like *startActivity(), startService()* and ICC entry points receives that data using API's like *getIntent(), startActivityForResult()*.

2.0.9 Effects of Security flaws in Android Application

Security unaware coding practices create quite a few vulnerabilities. One application can intercept the intent and set itself as an default activity, can receive unauthorized data without user even notice it, can invoke an unauthorized activity or service, forcefully intercepts a broadcast etc.

Activity Hijacking

Activity hijacking happens when activity achieves the ability of forcefully opening itself. An malicious application can register many possible intents in *AndroidManifest.xml* for an application using intent filter. When a application calls for an activity using implicit intent, malicious application will always be a valid candidate and with little unawareness for the user, it can open itself instead of providing a full of options to the user.

Service Hijacking

Service hijacking is similar to activity hijacking. It happens when a service achieves the ability of forcefully starting itself. An malicious application can register many possible intents in *AndroidManifest.xml* for an application using intent filter. When a application calls for an service using implicit intent, malicious application can receive the request for service and starts itself. Service hijacking is potentially more dangerous because it doesn't involve any UI and it can go unnoticed by the

user.

2.0.10 Privilege escalation

Each android application has it's own permission set. So, an application can only access information permitted by its permission list. But, cunning applications can conspire among themselves to obtain sensitive information and distribute among themselves which may lead to Privilege escalations. Privilege escalation happens when a sender app receives a sensitive information using API call and pass this data to receiver application via ICC where sender app doesn't have the permission of requesting the data in its permission list. For example, one application has permission of obtaining user's location coordinate. If it creates ICC channels with other applications where receiving application doesn't have location permission, it is privilege escalation threat.

2.0.11 Collusive Data Leak

Collusive data leak happens when a malicious activity is distributed among multiple applications. When an application obtains a piece of information and leaks it via an ICC exit point, it's called data leak. Collusive data leak attack happens when this data leaks happens with the help multiple applications. For example, one application can obtain location information and pass it to it's helper application using ICC exit point. Receiver application can leak those information using ICC exit point to other external sources. As data leak using single application is easier to detect, distributing these activities among multiple application makes it harder to detect.

CHAPTER 3 Threat Model

Our main objective of this paper is to identify *sensitive* ICC channels among two different android applications where developers willfull involvement is required. We define a ICC channel as *sensitive channels* when a data flow path starts from a *sensitive API* call and flow thorough a ICC exit point to another application using ICC methods. To define *sensitive API's*, we use results from the SuSi project.

As we have previously stated, android application has four major building components (*activity*, *broadcast receiver*, *service content providers*). These components from different application can communicate among each other to provide various useful functionalities for the developer. There is also a file called *AndroidManifest.xml* where major components, their properties and application specific permissions are defined. Though ICC provides tremendous usability and flexibility to developers, poor developer practices can lead to privacy and security related threats *e.g., activity hijacking , service hijacking, broadcast hijacking* [8]. These security threats involve one victim application and one malicious application. When a single application is involved in malicious activity, it's easier to detect them using malware analysis. But, researchers have found existence of more complex security threats where malicious activity is distributed among multiple apps. In this paper, we are focused on two types of security threats when malicious activity is distributed beyond a single application. One is *collusive data leak* and another is privilege escalation. To be able to detect such application, we need to find all possible sensitive ICC channels from each application to any other applications. One of the major challenges of analyzing such security threats is runtime scalability of pairwise analysis (each application needs to be checked with all other application for malicious activity. Hence, it becomes a *quadratic* complexity problem.

There is another inherent problem for analyzing such ICC base collusive channels. Naturally ICC based communications are arbitrary in nature when communication

is done using implicit intent. One application can fire an implicit intent using ICC exit leak and there can be multiple application with *ICC entry points* to receive that intent. It is also not uncommon to have ICC open interfaces in android applications for poor developer practices [14]. These issues can dramatically increase number of ICC collusive channels in pairwise-analysis. It can also detect a huge number of *false positive* in detecting collusive channels (one application can fire an intent with action string *Intent.ACTION_VIEW* to display an image, there can be multiple application to display that image. In such cases, developer's are completely unaware but pairwise analysis can detect as collusive ICC channel). To encounter these problems, we propose a novel methodlogy to identify ICC channels where developer's involvement is necessary to some extent.

We define six different ways of establishing communication channels using ICC. Each of these communication channels required active effort from developer's part to establish. One very obvious type is explicit intent. Others use implicit intent. We take away arbitrary nature of implicit intents and define each category such a way each target component can be uniquely targeted even though communication is established using implicit intent.

Threat type-V is communication channels established by ICC exit point *sendBroadcast* which includes both implicit intent and explicit intent. Threat type-VI is communication channels established by *Custom Provider*. Threat type **I, II, III, IV** consists of communication channels established by other ICC exit points *(startActivity, startService, startActivityForResult* with parameter variation in implicit and explicit intents.

3.0.1 Type I: Explicit Intents

We have defined Explicit intent in previous section. In explicit intent, developer explicitly specify target application's package name and component name. So, if a communication channel is established by explicit intents, it can be said that devel-

oper's are consciously trying to communicate with that specific component. A sample explicit intent in android is called as follows:

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("com.example.myapp", "com.example.myapp.
    MainActivity"));
startActivity(intent);
```

In this example, only *MainActivity.class* of app with package name *com.example.myapp* will respond to action requested by *startActivity* method provided a parameter *exported* is not set to *false* or *permission* parameter is not set in *AndroidManifest.xml*.

To receive the intent, *AndroidManifest.xml* in the receiver app needs to be as follows:

```
<manifest package="com.example.myapp" ... >
    <application ... >
        <activity android:name=".MainActivity" ... >
            ...
        </activity>
    </application>
</manifest>
```

3.0.2 Type II: Implicit Intents with Custom Actions

Action string is one of the three fields used for intent resolution. An implicit communication can be made explicit by carefully choosing an unique action string. For example, sender app can use an implicit intent but uses a unique action string, and the receiver app uses the same string in it's intent filter. In such cases, sender

app can directly communicate with receiver application even though explicit package and class name is not mentioned in the code.

```
String custom_action = "com.unique_action";
Intent i = new Intent();
i.setAction(custom_action);
startActivity(i);
```

In this example, sender app uses an unique action string com.unique_action in it's *action* field. If receiver app defines a component with an *intent filter* that filters with that unique action string (*com.unique_action*), intent will be directly delivered to the component.

```
<manifest package="com.example.myapp" ... >
    <application ... >
        <activity android:name=".MainActivity" ... >
           <action android:name="com.unique_action" />
        </activity>
    </application>
</manifest>
```

3.0.3 Type III: Implicit Intents with Custom Category

Category string is another parameter used in intent resolution process. Like custom action strings, developers can create intent objects in the sender app with custom category strings and target specific component of the receiver app without explicitly mentioning it's name.

```
String custom_category = "com.unique_category";
Intent i = new Intent();
i.addCategory(custom_category);
startActivity(i);
```

In this example, sender app uses an unique category string com.unique_category in it's *category* field. If receiver app defines a component with an *intent filter* that filters with that unique category string (*com.unique_category*), intent will be directly delivered to the app component

```
<manifest package="com.example.myapp" ... >
    <application ... >
        <activity android:name=".MainActivity" ... >
           <category android:name="com.unique_category" />
        </activity>
    </application>
</manifest>
```

3.0.4 Type IV: Implicit Intents with Custom URI

Data is another piece of information that Android uses to choose the right component for an intent. Normally, an URI object contains the information about the data to be acted on. Information can be location of the data or mime type of the data. For example, if one app want to delegate the task of editing an app to another app, the delegation process goes through an intent and intent object should contain the location of the image to be edited.

```
File fileToShare = new File("/sdcard/somefile.dat");
Intent i = new Intent();
i.setAction(Intent.ACTION_SEND);
i.setData(Uri.fromFile(fileToShare));
startActivity(i);
```

3.0.5 Type V: Custom Provider

Content providers provide ways to share data between apps. It creates a layer of abstraction to developers to do complex data related tasks. Though for secure communication, content providers have different protection level provided by android operating system, co-operation between application developers can easily compromise that. Sender have can create custom content provider and implement appropriate callback methods to share the data. If developer of the collusive apps share exclusive sharing credentials among them, receiver application can call the custom providers without any arbitrariness and receive the data. To implement a custom content provider, sender application extends *ContentProvider* class provided by Android SDK and implements all the callback functions. Server app also has to declare the *provider* in the application manifest. Client application sends a request my class that implements *Cursorloader*. A sample code is as follows:

```
OncreateLoader(){
    CursorLoader cursorLoader = new
    CursorLoader ( this,
    Uri.parse( "content://om.MyProvider"), null, null, null, null); }
onLoadFinished( Cursor cursor){
    If(cursor != null){
    if(cursor.moveToFirst()){ int val = cursor.getInt(1);
} }
```

A typical declaration code is as follows:

```
<provider android:name="MyProvider"
     android:exported="true"
     android:authorities="com.MyProvider"/>
```

Receiver app can call this custom provider to get the data and even listen to data source changes in the sender app.

3.0.6 Type VI: Custom Broadcast

Broadcasts are a messaging system in android that are used to pass a message across applications. Broadcast messages uses intent as a carrier. So, all the communication variants that are possible with intent, also possible with custom broadcast. But, they key difference between communication of intents with custom broadcast and other mediums is involvement of user interface. Broadcast messages doesn't require any user action, it can be transmitted in the background. A sample code in sender application is as follows:

```
Intent i = new Intent (this,com.B.Receiver.class);
i.setAction("com.broadcast");
i.putExtra(data,12345);
sendBroadcast(i);
```

Receiving app registers for broadcast using a receiver app and listens for any broadcast message. Upon receiving the the intent, it parses the data using *onReceive()* method..

```
<application >
<receiver android:name=".Receiver ">
<intent-filter> <action android:name="com.broadcast"/>
</intent-filter> </receiver>
```

```
public void onReceive ( Context context, Intent i ){
int val = i.getExtra(data,-1);
}
```

Using the above code, sender can send a broadcast message to *MyReceiver* class of receiver app without user knowing it. If receiver app properly declares that in it's manifest file, broadcast message will be seamlessly delivered.

Based on above type of communication channels, we define for type of security threats related to ICC communication.

CHAPTER 4 Research Method

Main goal of our study is to develop an automated process to identify developer's involvement in any malicious collusive channel. We take an empirical approach. On a high level overview, we collect most popular android applications from Android app market as our sample data, extract data flow path using static analysis, use our theoretical understanding described in **??** to establish collusive path that requires developer's involvement and finally do manually investigation to verify our approach and to understand the motivation of building such applications. We complete our experiment in five different stages.

4.0.1 Tool Selection/Development

Success of our experiment depends on accurate extraction of data flow path directly from android apk files. There are few research tools available for that. For our experiment, we choose Dialdroid, state-of-the-art tool for collusion detection in android. Though, Dialdroid has all the features available to serve our need, we observe few key pragmatic limitations of this tool. Dialdroid fails in ICC exit and Entry point extraction phase in many large real world applications. It has a timeout of 30 mins. But, for large real world applications (apk size more than 50 MB like Facebook, Angry Birds) this time limit is not enough. So, we implement a incremental update feature in current Dialdroid implementation. After this update, Dialdroid is able to save any data flow path it discovered before analysis timeout takes place.

4.0.2 Data Collection

Choosing right data source is extremely important for validity of an empirical study. To validate our finding we collect data from most reliable sources. We collected 1,20,181 most popular android applications from google playe store from 24 different categories. As we are considering malicious activities in android applications, we also consider downloading 796 known malicious applications for virusshare. So, overall we

Table 4.1: Categoty and Action Strings

action string		category string	
custom	non custom	custom	non custom
110251	571	40474	232

collect 1,20,977 applications from 15 different categories.

4.0.3 Feature Aggregation

To understand security threats using ICC, we need to extract properties from the source code of the apks. We use static analysis for this purpose. Feature extraction phase consists of several phases.

At first, we extract Entry/Exit points from the source code. Given the apk file, Dialdroid parses the *permission list* and *intent filters* form *AndroidManifest.xml*. Dialdroid use static taint analysis to determine ICC entry and exit points. To improve performance of taint analysis, Dialdroid uses dynamic precision configuration technique. It has two different configurations. *High Precision Configuration* and *Low Precision Configuration* . In *High Precision Configuration*, it uses a context sensitive taint path of maximum length 3 and in *Low Precision Configuration*, it uses a taint path of length 2. By default, Dialdroid uses *High precision Configuration* and if taint analysis fails within a specified timelimit, it runs with a low precision configuration. We analyze each of the 1,20,977 applications using Dialdroid. Analyzing this huge number of applications requires tremendous computing power. So, we look forward to high performance computing machines. The cluster computer was distributed over 40 nodes. Each node had Intel Haswell 2-thread 10-core chips with 64 GB memory which combinedly provides computing power of total 800 nodes. We write Linux shell scripts to automate our process and remotely submit our computation job over SSH.

4.0.4 Model Building

Major challenge of our study is to find properties of collusive channels that definitely proof developer's participation in any collusive task over ICC. So, we take help

Table 4.2: Custom and Non Custom Action Strings

Custom	Non Custom
com.distriqt.extension.notifications.NOTIFICATION	android.intent.action.MAIN
air.com.tinychat.mobile.intent.action.OPEN_ROOM	android.intent.action.VIEW
aviary.intent.action.EDIT	android.intent.action.PICK
aviary.intent.action.CDS_DOWNLOAD_START	android.intent.action.SEND

from the internal mechanism of ICC communication. We study the documentation of Android SDk and find the coding practices that must be present if developer of the application consciously participate in the collusion. We described such criterion in section **??**. Easiest proof of developer's participation in collusion is the explicit intent. A simple matching of sink and source of two different applications using explicit intent proofs developers' participation. But, to proof developers' participation using implicit intent is challenging. Implicit intent can be tweaked to target a specific component. In our previous phase of data aggregation, we collect all unique category and action strings. Some of the action and category strings are provided by android. For example, *android.intent.action.VIEW*, *android.intent.action.MAIN*, *android.intent.category.LAUNCHER*. We manually label each category and action strings as custom and non custom. Table 4.1 shows the summary of our labeling. Some samples of custom and non custom actions are listed in Table 4.2

4.0.5 Evaluation

In data aggregation step, we collect data flow path of each application. We match pair wise data flow path using their connecting ICC component. If we discover that the connecting path is using any customized intent, we can definitely say that this communication is under conscious consideration of the developer. Our approach discover many collusive channels that require developer's participation. We further investigate the source code of the suspected collusive app pairs to extract additional knowledge. We reverse engineer the apks and verify collusive app pairs and report interesting case studies.

CHAPTER 5 RESULTS

In our experiment, we analyzed 1,36,951 android applications using our custom ICC threat detection tool. Our tool saves extracted data in MySQL database. We run different SQL queries to compute sensitive ICC channels. From our experiment, we want to answer following research questions:

1. can we develop tools to determine developer's potential involvement in Privilege escalation and Collusion attack?

2. How prevalent is ICC based security threats in Android? Which kind of applications are mostly infected by sensitive collusive activity?

3. What are the possible motivations/reasons of establishing such collusive channels ?

4. What are the most vulnerable permissions?

5. What are top source and sinks methods of collusive channels?

In chapter 3, we described how developers can target a specific component of another app using ICC even with implicit intent. While explicit intent is direct proof of developer's effort to build a communication channel, properly tuned custom intent is also proves developers' willfull involvement in establishing such channels. We extract our data based on the methodology described in chapter 3 and calculate the data leak path using SQL queries. As many of the queries involve multiple tables and tables are quite large, we create multiple indexes for keeping the computation time to a reasonable limit. We manually investigate data leak paths to determine the accuracy of our tool. In all the cases, we find that all the data leak paths reported by our tool is deterministic by our definition in 3. We explain most compelling few cases in later sections. From our manual investigation, we can say that, it's possible to build tools that can identify collusive ICC channels where developers involvement is necessary. We have made our data,tools and codes open source for research community.

Table 5.1: Demography

Category	%apps	%collusion	% deterministic collusion	% of privilege escalation
Social	4%	12%	8%	0.06%
Tools	4%	8.5%	1.8%	0.33%
Entertainment	3%	8.9%	3.7%	0.12%
Personalization	14%	28.84%	3.12%	0.12%
Finance	4%	6.23%	1.32%	0.03%
Education	4%	7.88%	3.5%	0.15%
Communication	2%	7.66%	2.28%	0.30%
Medical	1%	7.6%	3.83%	0.2%
Lifestyle	5%	12.50%	4.47%	0.08%
Shopping	2%	6.73%	3.52%	0.02%
Photography	4%	11.41%	4.9%	0.02%
Transportation	3%	8%	2.63%	0.04%
Sports	5%	15.82%	6.12%	0.04%
Others	22%	13.60%	**6.11%**	0.23%
Productivity	4%	8.78%	1.55%	0.20%
Travel and Local	4%	10%	2.94%	0.18%
Weather	2%	9.9%	4.42%	0.16%
Libraries and Demo	2%	17.35%	4.63%	0.10%
Music and Audio	1%	8.16%	2.78%	**0.62%**
Media and Video	3%	7.78%	3.2%	0.15%
Comics	1%	10%	3.56%	0.19%
Business	2%	5.7%	1.7%	0.11%
News and Magazine	3%	8.71%	3.72%	0.06%
Health and Fitness	2%	12.8%	6.07%	0.09%
Virus	0.001%	7%	**20.1%**	**0.75%**

5.0.1 Statistics of Security Threats in Android

We extract ICC exit and entry leaks using *DialDroid* as described in chapter 1:research-method. We also compute collusive data leak path and privilege escalation. We establish collusive channels using the six methods we described in chapter 3. In our dataset, we have found total 3,50,891 channels among app pairs. Among them 9,747 channels are collusive in nature (receiver app leaks data using exit points). To answer our research question 2, we analyze collusive and privilege escalation channels for each of 26 categories. Table 5.1 shows ICC threats for each category. Category

Table 5.2: Top source and sinks

source name	percentage	sink name	percentage
getLastKnownLocation	**15.5%**	**startActivity**	**47.7%**
getDeviceId	**15.3%**	startService	24%
getLatitude	12%	android.content.Intent setAction	8.1%
getLongitude	12%	setResult	6.1%
query	10.4%	bindService	4.7%
getNetworkOperator	8.7%	sendBroadcast	4%
getConnectionInfo	6%	android.database.Cursor query	1.7%
getName	5.5%	android.content.ContentResolver: int delete	1.5%
getSubscriberId	4.6%	android.content.Intent setComponent	1.4%
getLine1Number	3%	android.content.ContentResolver: int update	0.1%

Table 5.3: Top leaked permission

Permission Name	**Percentage**
android.permission.ACCESS_NETWORK_STATE	66%
android.permission.ACCESS_FINE_LOCATION	8%
android.permission.ACCESS_COARSE_LOCATION	7%
android.permission.READ_PHONE_STATE	3%
android.permission.ACCESS_WIFI_STATE	0.76%
android.permissionget.ACCESS_WIFI_STATE	0.18%

Virus has highest percentage of privilege escalation and collusive data leak. As these applications are experimentally chosen malicious applications, it's expected to have high number of ICC treats. Apart from these, we persona

Table 5.4: Collusion in Android using ICC

Type	Name	number of channels	Same Developers	Collusion	Privilege Escalation
I	Broadcast	84%	97.90%	50.3%	88%
II	Explicit	8%	1.50%	47.9%	0.40%
III	Intentional Data	4%	1.7%	0.04%	1.70%
IV	Custom Provider	4%	0%	95%	9%
V	Custom Action	0.1%	0.0%	0%	0%
VI	Custom Category	0.01%	0.0%	0%	0%

Table **??** shows number of collusive and privilege escalation results we have found. Collusion using broadcast is very prominent in number. As we have discussed, broadcast can be sent and received without involvement of user interface, it's is heavily used and hence heavily misused.

Table 5.5 sheds light on our first research question. It shows the existence of security threats using ICC. Uses of ICC in android is prolific that even though the

Table 5.5: Statistic on ICC exit and Entry Leaks

Exit leaks	Exit Leaks in Collusion	Entry leaks	Entry Leaks in Collusion
12,66,537	20,332	13,50,641	11,964

percentage of ICC exit or entry leaks participating in collusive channel is not large, it can pose significant security threat.

Table 5.4 shows, summary of collusive channels we have found from our data set. We find 3,51,941 potential collusive channels. Category *Virus* has most percentage of vulnerable applications. It tops in security threats both in collusion and privilege escalation. However, it's quite natural because they are specifically designed to be malicious applications. Apart from this, collusive data leak and privilege escalation is prevalent in all the categories somewhat uniformly.

Then we investigate developers' involvement in establishing these channels. Based on our methodology, we run queries to identify developers active effort. We find total 9,747 collusive channels where developers' active effort is required. We further do empirical investigation to strengthen our claim.

5.0.2 Permission Leaks

Privilege Escalation is one of the motivation of a collusive attack. Malicious app can receive sensitive data from other applications using collusive channels for which receiver have no authorization for. Thus, creating a privacy hazard. To answer our research question 3, we investigate top permissions that are violated by collusive apps. Table 5.3 shows that location based permission are among the sensitive permissions that are of interest of malicious apps. Though permission related to network information has incredibly high percentage of interest, our empirical investigation suggests that they are mainly because of internet access related application flow control which are not malicious and commonly used across apps.

5.0.3 Top Source and Sinks

Source method is the starting point of a data flow. A flow starts form the access point of an API. On the other hand, sink methods are the end point of a data flow. Sink method can be the point where sender sends a piece of information to other apps using ICC or to other consumer points like a network socket. Source and Sink tell a lot about a collusive channel. Table 5.2 shows top source and sinks methods found in our study. Our results suggests that information that can be used to target *e.g., location, deviceId* a user is heavily prone to collusive attack.

CHAPTER 6 CASE STUDIES

6.0.1 Deterministic Collusive Data Leak

In previous sections, we described our methodology to automatically determine developer's involvement in establishing a collusive channel and provided results based on our detest. In this section, we manually check few of the compelling cases reported by our tool and try to understand the nature of those channels and motivation of the developers. For this purpose, we decompile our apk with *ByteCodeViewer* and investigate the source code.

6.0.2 Explicit Collusion:

Explicit intents uses specific component name to communicate with the receiver. Our tool found existence of cases where sender app uses fully specified package and component name for communication. This communication takes place between **com.floaters.search** and **com.newsflashapp.usnews**. **com.floaters.search** is the sender application and **com.newsflashapp.usnews** is the receiver application. From our investigation, this communication takes place because both the application uses same sdk. Sender app sends specific data using explicit intent. In the receiver app, *AndroidManifest.xml* defines a *broadcast receiver*. Using the class *com.tooleap.sdk.TooleapReceiver*, it receives the data and pass it further using *sendBroadcast* sink method. From our investigation, it we conclude that this communication channel is needed required developers involvement even though it is not malicious in nature.

Location Data Leak

Location of an user is a sensitive piece of information. Our tool detect 3,857 deterministic collusive location data leak paths. One of the app with package *com.team4win.tugroom* that offers room searching facility at a university campus. This app doesn't have any permissions to retrieve location, but still it listens to

any incoming intent with data scheme *geo* and any url set to *maps.google.com*. Upon receiving data from intent, it calls an ICC sink *statActivityForResult()*. Our tool detect an app with package name *com.comdataclc.hotellocator.android* as an potential sender for this app. Sender app obtains Location data using ¡android.location.LocationManager: android.location.Location getLastKnownLocation(java.lang.String) api call. After a click event on a button, sender app calls stratActivity with custom intent which contains location data.

6.0.3 Collusive Data Leak Using Libraries

Android ecosystem provides many libraries to provide different facilities to the developers. Many of these libraries are used by many popular applications. Our tool find many of these applications that uses same library contribute to collusive data leaks. We observe that two applications with package name com.clc.hotellocator.android and **com.homelessdevelopers.fue**. Like the previous case, sender app send data by customizing the data parameter of the intent. Upon receiving the intent in *MainActivity* class, receiver app sends location the data over email.

cl.movistar.android and *net.bluumi.ForempFormacion* uses a library provided by **xtify**. These library internally uses ICC communication channels with same parameters. As same library is used across applications, collusive data leak paths are created where one client application can listen to other client applications communications.

Deterministic collusion using Custom Provider

Our results shows that 95% of total deterministic collisions uses custom provider. We manually investigate most compelling apis that are subject to collusive attack. com.creativemobile.dr4x4 - com.facebook.katana , com.kimtips.app - com.facebook.katana : Both of these applications calls **android.telephony.TelephonyManager: java.lang.String getSubscriberId()** api using custom provider sends the information to com.facebook.katana. Upon Receiv-

ing this information , facebook app starts a service which leads to collusion attack. We find similar pattern in **android.net.wifi.WifiInfo: java.lang.String getSSID()** and **android.telephony.TelephonyManager: java.lang.String getSimOperatorName()** as deterministic collusive attack.

6.0.4 Same Developer Collusion

Collusion using ICC mostly depends on effective contract between components of applications. When developer of all participants app are same, it is easier to create these contracts. So, we specifically investigate existence of collusion between same developer app pairs. One such pair is **com.pd.gsapro** and **com.pd.golfapp**. Sender application sends data using broadcast like any another collusive channels and receiver application upon receiving leaks the data. But we observe that almost all of this collusion stems from the fact that they use almost same third party library. As these libraries use same action strings for communication, we creates collusive leak. Even though these leak paths may not be designed to leak the data, it can be easily exploited and far from security aware practice.

We also observe similar effect in same developer app pair **com.mobipath.ailemnerede** and **com.mobipath.caminerede**. We observe there is a collusive path that leaks *deviceId*. After, further investigation we observe that this collusive path exists because both of these application uses same paypal library for some of it's functionality.

6.0.5 Privilege Escalation

Privilege escalation is a classic example of collusive attack. We have found quite e few interesting cases of privilege escalation.

Location Escalation

Location is an important piece of information. Android application requires two different set of permissions. One is *ACCESS_COARSE_LOCATION* and another is

.ACCESS_FINE_LOCATION. android.permission.ACCESS_FINE_LOCATION permissions allows to access location from both gps and network provider while ACCESS_COARSE_LOCATION allows only access to network provider. We have found total 2550 communication channels where there is potential scope of privilege escalation using location data. We find that an application **com.ichueca.pebblealarm** retrives location data both from gps and network provider with necessary permissions. Then it sends the data using broadcast using custom intent **com.getpebble.action.SEND_NOTIFICATION** . We observe that receiver application registers a receiver with same action in its intent filter. Receiver application **com.matejdro.pebblenotificationcenter** doesn't have the *android.permission.ACCESS_FINE_LOCATION* permission to acquire accurate location but using the communication channel it is receiving the data which leads to privilege escalation.

DeviceID Escalation

Android **deviceId** is an important information to target an user phone. DeviceId is also an integral part of push notification based advertisement system. An application requires permission *"android.permission.READ_PHONE_STATE"* to retrieve this value. We have found few communication channels where this information is potentially delivered to unauthorized application. One popular application with package name **com.acme.android.powermanager** acquire required permissions for retrieving battery information including deviceId and saves it as a text file in **external storage**. After that, it sends a system wise broadcast message with custom action. Another application **com.iridium.mailandweb** registers that action using intent filter where it can easily read the unauthorized data from external storage. In this communication channel no direct data is transferred rather a middle layer is created and just signal of new data is passed.

CHAPTER 7 DISCUSSION

In this study, our main focus is to develop methods to determine developers involvement in collusive attack. We collected a significant number of applications and generated a good amount of data to empirically justify our findings. our results show that it is possible to determine developers involvement in creating collusive channels among apps with acceptable accuracy. But, there are few limitation that will require further research. For example, currently we can determine active participation of developers to create ICC channels, but it's hard to say that they are definitely malicious.

Our novel methodoloy can dramatically decrease false positive in identifying collusive channels (any ICC channel that are created arbitrarily are not collusive). A big part of increasing accuracy of our model relies on safely identifying custom action and category strings. Though we have developed our custom list of action and category list, it can be further refined. For example, many popular sdk (e.g., firebase, urbanairship) uses some custom strings. If they are used in multiple applications, they will be detected as collusive channels.

We have found that many of the collusive threats we have discovered are mostly due to poor development practices. Google provides guidelines about secure coding practices in intent based communication. In Case studies 6 we talked about collusive data leak path due to using same libraries across. We observe that in many of the popular libraries use the same security unaware code.

```
Intent localIntent = new Intent("com.some-action");
localIntent.putExtra("text", "sometext");
sendBroadcast(localIntent);
```

We find the exact similar code snippet in myriad amount of applications during

our manual investigation. This code snippet seems apparently benign, but when used across applications, it can create collusive path because broadcast sent by SDk method call *sendBroadcast()* is transmitted across applications. So, other application can listen to it. Android documentation provides safer way to send intra-app ICC using *LocalBroadcastmanager*, but our investigation says that this practice is rare among developers.

```
LocalBroadcastManager localBroadcastManager = LocalBroadcastManager.getInstance
    (context);
    Intent localIntent = new Intent("com.some-action");
    localIntent.putExtra("text", "sometext");
    localBroadcastManager.sendBroadcast(localIntent);
```

To mitigate the application level security issues using collusion it is required to improve developer's knowledge about secure coding styles. IDE's acan also be made security aware.For example, *exported* or *permission* parameter can be integrated as auto-fill in Android Studio. Thus, developer's can be more informed about secure coding practices.

We have found existence of Privilege escalation in our dataset. We also suggest structural changes in Android sdk to overcome this threat. Like *permission* parameter in *AndroidManifest.xml* is used to control access to a component from other app. It can limit how a certain component can can called from other app *(e.g., startActivity(), startActiityForResult()*. A similar strategy can be used for ICC regarding data transfer.

One concern with this method is maintaining scalability and deployablity of the proposed technique. If we want to deploy this system for real world use, we have to maintain a large database. Though using proper query planning and indexing, it

should be able to process very large amount of data, for further improvement we can introduce machine learning base technique.

Validity of this method heavily depends on careful selection of sensitive API. We primarily choose our sensitive API from SuSi project. To increase speed of calculation, we choose only the most sensitive ones. To increase the depth of the analysis, complete sensitive api list can be used.

CHAPTER 8 RELATED WORK

ICC security has always been challenged by security researchers since its birth. Researches on ICC security can be divided into two types. In early days, researchers were concerned about single application security analysis. Over the years, sophisticated application level security threats are discovered and researchers have also investigated possibility of developing salable tools to nullify those threats. Two techniques are employed by researchers to analysis security threats. Static analysis and Dynamic Analysis.

Comdroid[8] is the first complete study on ICC security vulnerabilities in android. It is a single app analysis technique which focused on component hijacking threats (*e.g., activity hijacking, broadcast theft*) 2. But, as it is a single app analysis technique, it suffers from high number of false positives. Droidsafe[11] and AmanDroid [18] are two solutions that uses static analysis for finding ICC threats in single app. TainDroid[10] is a dynamic analysis technique for single application to detect privacy leaks.Though these are comprehensive studies on ICC security, pairwise app analysis is required to increase accuracy and to understand the overall security landscape.

Researchers have used both static and dynamic analysis for pairwise app analysis. XmanDroid[7] was the first attempt of dynamic analysis in this regard. FlaskDroid[17] is dynamic analysis tool for collusive threat and privilege escalation detection. However, these dynamic analysis techniques had few limitations. They were not scalable and only works for small dataset.

DialDroid[6] uses static analysis and relational database to build a scalable solution for pairwise app analysis. It establishes collusive channels from sender application to receiver application using dataflow analysis to find collusive threats among apps. But this approach is more generalized and can not tell about degree of developer's involvement in establishing such channels. PRIMO[15] is another project that uses

probabilistic models to build sensitive ICC channels for providing complementary information on pairwise ICC threats.

CHAPTER 9 THREATS TO VALIDITY

In this study, our main focus is to develop methods to determine developers involvement in collusive attack. We collected a significant number of applications and generated a good amount of data to empirically justify our findings. our results show that it is possible to determine developers involvement in creating collusive channels among apps with acceptable accuracy. But, there are few limitation that will require further research. For example, currently we can determine active participation of developers to create ICC channels, but it's hard to say that they are definitely malicious.

Our novel methodoloy can dramatically decrease false positive in identifying collusive channels (any ICC channel that are created arbitrarily are not collusive). A big part of increasing accuracy of our model relies on safely identifying custom action and category strings. Though we have developed our custom list of action and category list, it can be further refined. For example, many popular sdk (e.g., firebase, urbanairship) uses some custom strings. If they are used in multiple applications, they will be detected as collusive channels.

We have found that many of the collusive threats we have discovered are mostly due to poor development practices. Google provides guidelines about secure coding practices in intent based communication. In Case studies 6 we talked about collusive data leak path due to using same libraries across. We observe that in many of the popular libraries use the same security unaware code.

```
Intent localIntent = new Intent("com.some-action");
localIntent.putExtra("text", "sometext");
sendBroadcast(localIntent);
```

We find the exact similar code snippet in myriad amount of applications during

our manual investigation. This code snippet seems apparently benign, but when used across applications, it can create collusive path because broadcast sent by SDk method call *sendBroadcast()* is transmitted across applications. So, other application can listen to it. Android documentation provides safer way to send intra-app ICC using *LocalBroadcastmanager*, but our investigation says that this practice is rare among developers.

```
LocalBroadcastManager localBroadcastManager = LocalBroadcastManager.getInstance
    (context);
    Intent localIntent = new Intent("com.some-action");
    localIntent.putExtra("text", "sometext");
    localBroadcastManager.sendBroadcast(localIntent);
```

To mitigate the application level security issues using collusion it is required to improve developer's knowledge about secure coding styles. IDE's acan also be made security aware.For example, *exported* or *permission* parameter can be integrated as auto-fill in Android Studio. Thus, developer's can be more informed about secure coding practices.

We have found existence of Privilege escalation in our dataset. We also suggest structural changes in Android sdk to overcome this threat. Like *permission* parameter in *AndroidManifest.xml* is used to control access to a component from other app. It can limit how a certain component can can called from other app *(e.g., startActivity(), startActiityForResult()*. A similar strategy can be used for ICC regarding data transfer.

One concern with this method is maintaining scalability and deployablity of the proposed technique. If we want to deploy this system for real world use, we have to maintain a large database. Though using proper query planning and indexing, it

should be able to process very large amount of data, for further improvement we can introduce machine learning base technique.

Validity of this method heavily depends on careful selection of sensitive API. We primarily choose our sensitive API from SuSi project. To increase speed of calculation, we choose only the most sensitive ones. To increase the depth of the analysis, complete sensitive api list can be used.

CHAPTER 10 CONCLUSION

In this paper, we only focus on intentional icc communication channels . But there are few other ways to initiate a collusive privilege escalation attack. For example, one app can store data in external file system in plain text or in encrypted form and other app having the knowledge of location of the file and keys in case of encrypted text can obtain sensitive information. In our future study, we want to explore these areas.

Kotlin is Java alternative as a primary development language for Android. It is getting popular among the developer community. As Kotlin is also a JVM language, we plan to extend our tool for application developed i Kotlin.

Android wear is getting popular among android user base. Developers are trying to combine features from different wearable and provide users with unique experiences. To provide functionalists among different applications distributed across devices requires inter application communications. We also plan to explore this area of wearable gadgets for security threats.

We also wish to customize our tool to make it mass deployable. We want to create a rich of database of malicious coding practices in android community. So that, we can develop future tools to refrain developers from pushing security unaware codes in production.

REFERENCES

[1] Aafer, Y., Du, W., and Yin, H. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems* (2013), Springer, pp. 86–103.

[2] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss* (2014), vol. 14, pp. 23–26.

[3] Arzt, S., Rasthofer, S., and Bodden, E. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114* (2013).

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices 49*, 6 (2014), 259–269.

[5] Bagheri, H., Sadeghi, A., Garcia, J., and Malek, S. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering 41*, 9 (2015), 866–886.

[6] Bosu, A., Liu, F., Yao, D. D., and Wang, G. 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* (2017), ACM, pp. 71–85.

[7] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A.-R. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[8] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (2011), ACM, pp. 239–252.

[9] Elish, K. O., Shu, X., Yao, D. D., Ryder, B. G., and Jiang, X. Profiling user-trigger dependence for android malware detection. *Computers & Security 49* (2015), 255–273.

[10] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 5.

[11] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. Information flow analysis of android applications in droidsafe. In *NDSS* (2015), vol. 15, p. 110.

[12] Klieber, W., Flynn, L., Bhosale, A., Jia, L., and Bauer, L. 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* (2014), ACM, pp. 1–6.

[13] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., and Le Traon, Y. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference* (2015), Springer, pp. 513–527.

[14] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the*

37th International Conference on Software Engineering-Volume 1 (2015), IEEE Press, pp. 280–291.

[15] Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., and Le Traon, Y. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, pp. 469–484.

[16] Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., and Molloy, I. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 241–252.

[17] Tan, Y.-a., Xue, Y., Liang, C., Zheng, J., Zhang, Q., Zheng, J., and Li, Y. A root privilege management scheme with revocable authorization for android devices. *Journal of Network and Computer Applications 107* (2018), 69–82.

[18] Wei, F., Roy, S., Ou, X., et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1329–1341.

ABSTRACT

An Empirical study on Deterministic Collusive Attack in Android Application

by

TANZEER HOSSAIN

July 2019

Advisor: Dr. Amiangshu Bosu

Major: Computer Science

Degree: Master of Science

Security threats using intent based inter component communication (ICC) channels in Android are under constant scrutiny of software engineering researchers[1], [2] [4][8]. Though prior research provides empirical evidence on the existence of collusive communication channels in popular android apps, little is known about developers' willful involvement and motivation to exploit these channels. *To shed light on this matter, in this paper we devised a novel methodology to deterministically identify developers' involvement in establishing collusive inter app communication channels.* We incorporate static analysis and relational database technology to discover sensitive collusive channels and domain knowledge of the Android SDK to build a model to identify deterministic inter component channels between two different apps. Our results provide empirical evidence that a properly tuned model built on internal mechanism of intent based communication can accurately determine developers' potential involvement in establishing malicious communication channels. We also report various intriguing statistics, performance improvement of state-of-the art ICC resolution/data-flow analysis tool and interesting case studies regarding developers involvement in sensitive collusive inter app communication.

AUTOBIOGRAPHICAL STATEMENT

Tanzeer Hossain

EDUCATION

- Masters Candidate (Computer Science)
  Wayne State University, Detroit, MI, USA

- Bachelor of Science (Computer Science and Engineering), 2016
  Bangladesh University of Engineering and Technology, Dhaka