Android – Collusion Conspiracy

Authors: Igor Muttik, Jorge Blasco, Tom Chen, Harsha K. Kalutarage and Siraj A. Shaikh

Abstract: We describe, analyze and demonstrate how a set of Android apps can, when working together, circumvent the current Android security model. The problem is that a set of colluding apps may be able to perform actions beyond the limitations set by the OS. These capabilities can easily go unnoticed because only individual app's permissions are shown. We demonstrate the scale of the problem based on analyzing permissions of prevalent apps. Colluding apps create a problem for both users' privacy and security; we demonstrate examples of colluding app sets: one leaking sensitive user data and a "distributed botnet". Finally, we discuss solutions to the problem.

Introduction

The Android security model was designed to protect users, data, applications, the device and the network from security threats. By default, any third party app is treated as untrusted by the OS and runs inside a sandbox that isolates it from any sensitive resource or other applications. Access to sensitive system resources is protected by the OS permissions system. If an application wants to access a sensitive resource, it must include a permission declaration inside the `AndroidManifest.xml` file. When the application is being installed (in a device running Android versions below 6.0), the system will ask the user to accept the permissions used by the app before proceeding with the installation. At this point, the user must accept or deny all permissions the app is requesting. Starting with Android 6.0 applications can ask for permissions at runtime and users have the choice of granting or denying each permission.

Apps in the sandbox can communicate with other apps via standard Unix mechanisms (files, sockets, etc.) or the Inter-Process communication (IPC) mechanisms provided by the Android OS. Generally speaking, apps accessing Android IPC mechanisms do not need to request specific permissions unless one of the apps that takes part in the communication, requires them. This must be enforced by developers and is intended to avoid apps exposing protected resources to other apps that have not requested access to those same resources. An example of this is the usage of the Phone app. Other apps can request (by using `Intents`) the Phone app to start a phone call. When this happens, the phone app opens and starts a call to a number specified in the Intent. As the access to this resource is protected, apps requesting the Phone app to start phone calls need to declare the "`android.permission.CALL_PHONE`" permission.

The Android OS does not check if an app that is accessing a permission-protected resource through another app has itself requested that permission. This task is left to the developer of the app that is exposing the access to the permission-protected resource. This lack of control can be used by apps to get access to sensitive resources they are not supposed to have. In this scenario, applications can "conspire" using communication channels to aggregate their permissions and perform malicious actions. In addition to this, malware analysis services as

well as security researchers normally focus on single apps. Therefore, they cannot detect when malicious actions are distributed over several apps installed on a device.

In this work we describe, analyze and demonstrate how a set of Android apps can break the current Android security model. The rest of the paper is structured as follows. In the next section we define the concept of application collusion and describe the privacy and security implications of colluding apps. In section three, we demonstrate the threat created by these apps with two app sets that affect different system protected resources. In section four, we describe a preliminary analysis of the most prevalent apps from Google Play and other markets. Finally, we discuss how app collusion can be detected and approaches to improve protection at both device and market level.

Application Collusion

The origin of the colluding application problem can be traced back to the *confused deputy* attack (Hardy, 1988). This attack can happen when an application provides a public interface to access some restricted resources. Under this circumstances, other application could use that interface to abuse the restricted resources. The application providing access to the protected/restricted resource is called a confused deputy. In Android, confused deputy attacks can happen in a form of *permission re-delegation attacks* (Porter Felt, Wang, Moshchuk, Hanna, & Chin, 2011). A careless developer may expose permission-protected resources when publicly enabling the component that accesses those resources for communication from other applications through IPC.

The first documented example of intentional permission re-delegation is *Soundcomber* (Schlegel, Zhang, Zhou, Intwala, Kapadia, & Wang, 2011). This proof-of-concept malware is composed of two apps (Figure 1). The first app, which requires only access to the device microphone (`RECORD_AUDIO` permission), listens for calls to telephone banking services and extracts the digits pressed by the user. The second app receives the extracted sensitive information and sends it to a remote server (`INTERNET` permission). *Soundcomber* uses Android overt (IPC) and covert channels (file locks, settings modifications, etc.) as communication channels.

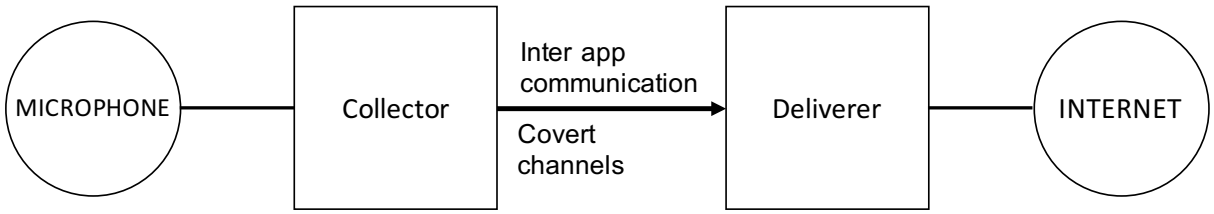

Figure 1: Apps comprising the 'Soundcomber' proof of concept malware. The Collector access the microphone and sends data to the Deliverer using overt or covert channels. The deliverer uses the INTERNET permission to extract the information from the device

Definition

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

Confused deputies (through permission re-delegation) occur only when a programmer accidentally creates a vulnerable app. Therefore, *application collusion can be defined as the act of cooperation between two or more apps to share their access to protected resources so they can execute a potentially harmful action which they could not perform separately with their own privileges*.

Misconceptions

The *Soundcomber* example and other works that have addressed the problem of app (Marforio, Francillon, & Capkun, 2011) (Bugiel, Davi, Dmitrienko, Heuser, Sadeghi, & Shastry, 2011) have focused only on scenarios where colluding apps are used for information theft. In this work we show how collusion is not only restricted to information theft by demonstrating a botnet client distributed across several apps.

Another relevant aspect of collusion is its differentiation from collaboration and confused deputies. As an example, a camera app usually saves pictures on external storage of a device. Photo sharing apps read photos from the external storage and upload them to the Internet. Although this behavior is benign, as the user has given consent, the actions of the camera app (confused deputy) could be used by a malicious app to extract the photos from the device. In a similar way, a malicious camera app could send all the pictures taken to another colluding app for extraction.. Distinguishing collaboration from collusion in this case, is a challenging task, as mentioned by other researchers (Chin, Porter, Greenwood, & Wagner, 2011), (Elish, Yao, & Ryder, 2015). It may boil down to determining whether the photos – in this example – were left accessible/unencrypted by mistake or deliberately.

Communication channels

The Android OS offers a wide range of communication options that enable apps to cooperate and share information. Colluding apps can use these to communicate and facilitate their malicious actions. To increase stealthiness against security software that may be monitoring these channels, colluding apps can also establish covert channels by exploiting the OS API calls and information leaks. This section provides an overview of the overt and covert channels available in the Android OS.

Overt Channels

The Android OS supports different channels to transfer information between apps. Each of these were designed to address specific app interaction needs:

- **Intents** are messages used to request actions from other application components (Activities, Services or BroadcastReceivers). These can belong to the same or different apps. There are two types of intents: explicit and implicit. Explicit intents target specific activities or services. Implicit intents target generic actions that can be performed by many different activities (send a message, open a web link, etc.) If the intent launches a new activity, the foreground activity is placed in the background and the activity specified by the intent is shown to the user. A service will run in the

background with no user interface and will deliver updates to the activities that are using it. Intents can also be captured by BroadcastReceivers if sent accordingly. Code caption 1 shows the code required to send information to another activity, service and broadcast receiver using different kinds of Intents.

```
// Explicit Intent creation to launch ActivityB
Intent i1 = new Intent(this, ActivityB.class);
startActivity(i1);
// Explicit Intent creation to launch ServiceB
Intent i2 = new Intent(this, ServiceB.class);
startService(i2); //bindService(i2) could also be used
// Implicit intent launch code
Intent i = new Intent();
i.setAction("myapp.action.send_info");
i.putExtra(Intent.EXTRA_TEXT, "Some text to send");
sendBroadcast(i);
```

Code caption 1: Code required to send information via different kinds of intents

Activities, services and broadcast intents declare the intents which they can handle by declaring a set of `IntentFilters`. For activities and services, intent filters must be declared in the app manifest XML file (Code caption 1Code caption 2). Broadcast receivers can also register their intent filters programmatically during runtime.

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

Code caption 2: Example Intent filter declared for an example Activity in the app manifest

For security reasons, Android services cannot be accessed unless the tag 'android:exported" is set to true in the app manifest.

- **Content Providers** are used in Android to transmit structured data across different apps. Content providers store information in one or more tables, in a similar way as relational databases do. Apps access data of content providers using *ContentResolver* objects. They offer methods not only to read data, but also to update, create and delete information from the content provider object offered by the other app. Apps requiring read or write access to content providers must declare necessary permissions in their manifest file (if the content provider requires them). For instance, an app that wants to access the contact list of a user (Code Caption 3) will need to declare the READ_CONTACTS permission in the manifest file.

```
Cursor c = null;
   c = getContentResolver().query(
   ContactsContract.Data.CONTENT_URI, // URI of the contacts table
   mProjection, // The columns to return for each row
   mSelectionClause, // null, or variable to perform matching in each row.
   mSelectionArgs, // Either empty, or the value to match
   mSortOrder); // The sort order for the returned rows
```

Code caption 3: Code required to execute queries over the content provider

- **External Storage** is a specific Android storage option available in Android devices through an USB connection, SD card or 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. Files on external storage can be accessed using common File access API.

- **Shared Preferences** is an OS feature that allows apps to store key-value pairs of data. Its purpose is to be used to store preferences information. Although it is not intended for inter-app communication, apps can use key-value pairs to exchange information if proper permissions are defined when accessing and storing data (before Android 4.4). Code caption 4 shows how to save information into shared preferences, so it can be read by other apps.

```
SharedPreferences                    sp                    =
getSharedPreferences("PrefsFile",MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sp.edit();
editor.putString("key", "value");
```

Code caption 4: Code required by an app to write a world readable preference file

Code caption 5 shows the code required to read the same information from another app. This code would work on any Android device below version 4.3.

```
Context otherAppsContext = createPackageContext("com.other.package.example", 0);
sp = otherAppsContext.getSharedPreferences("PrefsFile",  MODE_WORLD_READABLE);
String data = sp.getString("key","nothing");
```

Code caption 5: Code required to read data from a preference file located in other app package

Covert Channels

Covert channels take advantage of some of the APIs or features offered by the OS to enable communication between processes. There is no formal method to obtain all covert channels that may exist in a computer system. Examples of Android covert channels include: audio settings (they can be read and modified without permissions); broadcast events triggered by setting changes; File locks; process enumeration; Unix socket discovery; amount of free RAM/storage space and CPU utilization among others (Marforio, Ritzdorf, Francillon, & Capkun, 2012) (Schlegel, Zhang, Zhou, Intwala, Kapadia, & Wang, 2011). Table 1 provides a summary of some of these channels and some keywords that can be used to identify them through static analysis.

Table 1: Non-exhaustive list of covert channels and keywords to identify their usage

Covert Channel	Colluding Role	Location	Keywords

Audio Settings	Sender	Java	`Context.AUDIO_SERVICE`
			`adjustStreamVolume`
			`adjustSuggestedStreamVolume`
			`adjustVolume`

	Receiver	Java	`Context.AUDIO_SERVICE` `getStreamVolume`
Settings Broadcast	Sender	Java	`Context.AUDIO_SERVICE` `setVibrateSetting`
	Receiver	Java or Manifest	`RINGER_MODE_CHANGED`

Wake Lock	Sender	Java	`Wakelock`
			`acquire`
			`WakefulBroadcastReceiver`

	Receiver	Java	`ACTION_SCREEN_ON` `ACTION_SCREEN_OFF`

File Lock	Sender	Java	`FileLock`
			`lock`
			`release`

	Receiver	Java	`isValid`

Proc. Enumeration	Sender	C	`fork`
			`pthread`
			`create`

	Receiver	C	`proc`

		Manifest	`GET_TASKS`
		Java	`ActivityManager getRunningServices`
Socket Enumeration	Sender	Java	`Socket`
	Receiver	Java	`Socket isClosed`
Free space	Sender	Java	*Not possible to limit*
	Receiver	Java	`StatFs & getAvailableBlocks MemoryInfo & availMem`
CPU Usage	Sender	Java	*Not possible to limit*
	Receiver	Java	*Not possible to limit*

Demonstration of App Sets

Previous research on application collusion has focused only on app sets specifically designed to steal information.

As part of our research, we have developed several colluding app sets to demonstrate broader possible threats. We present two such colluding app sets. The first one is a colluding app set that uses several applications to steal information from the user. This set is different from the ones already present in the literature as it is comprised of three applications instead of only two. The second demonstration app set is a botnet client distributed across several apps. Depending on the amount of apps installed by the infected user, command and control server would have access to different bot functionality on the user device.

Contact Extractor Demonstration App Set

This app set is comprised of three apps (Table 2). This group sends the device's address book to a remote server. The first app reads the contacts from the address book. This information is shared with the second colluding app via `SharedPreferences` library. The second app acts as a message forwarder and sends the received information to the third app by using the external storage. The third app of the colluding set uses its Internet connection to transmit gathered information to a remote server. A graphical representation of these apps is shown in Figure 2.

Table 2: Summary of apps included in the contact extractor demonstration app set

App Package	Permissions	Colludes with	Channel
com.acid.trans	`READ_CONTACTS`	com.acid.fwgame	Sh. Preferences
com.acid.fwgame	`WRITE_EXTERNAL_STORAGE`	com.acid.recv2	Ex. Storage
com.acid.recv2	`READ_EXTERNAL_STORAGE INTERNET`		

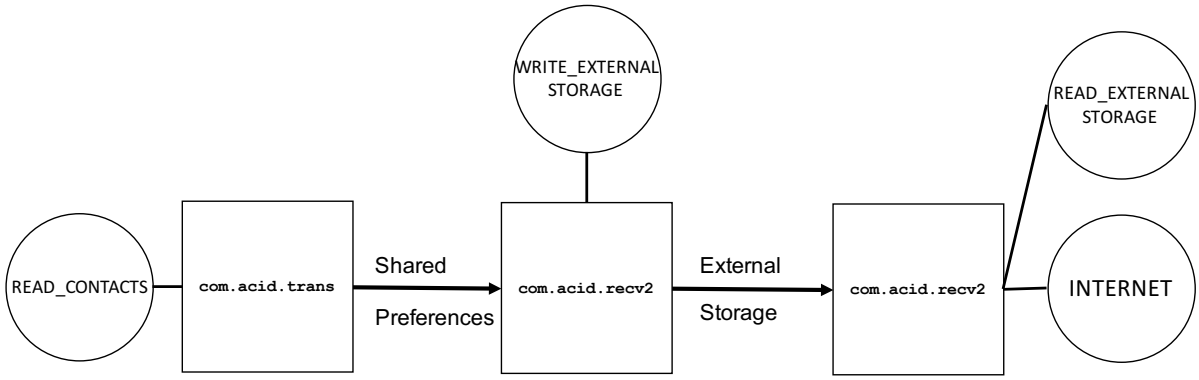

Figure 2: Graphical representation of the apps part of the contact extractor demonstration app set

Botnet Client Demonstration App Set

This demonstration group is comprised of four apps (Table 3). One of the apps acts as a relay receiving orders from a command and control (C&C) center. The other colluding apps execute commands received from the C&C depending on their requested permissions. The companion apps are capable of sending SMS messages, stealing the device address book and starting and stopping tasks. This group uses `BroadcastIntents` as a communication channel between apps.

Table 3: Summary of apps included in the botnet client demonstration app set

App Package	Permissions	Colludes with	Channel
com.acid.weatherapp	`INTERNET`	com.acid.enhancesms com.acid.contactmanager com.acid.taskmanager	Intents
com.acid.enhancesms	`READ_SMS` `SEND_SMS`		
com.acid.contactmanager	`READ_CONTACTS`	com.acid.weatherapp	Intents
com.acid.taskmanager	`GET_TASKS` `KILL_BACKGROUND_PROCESS`	com.acid.weatherapp	Intents

The C&C server can be used to transmit commands to the “Weather” app. All commands are then forwarded to the respective companion app. If the app is installed, the command will be executed via a Broadcast Intent (Figure 3).

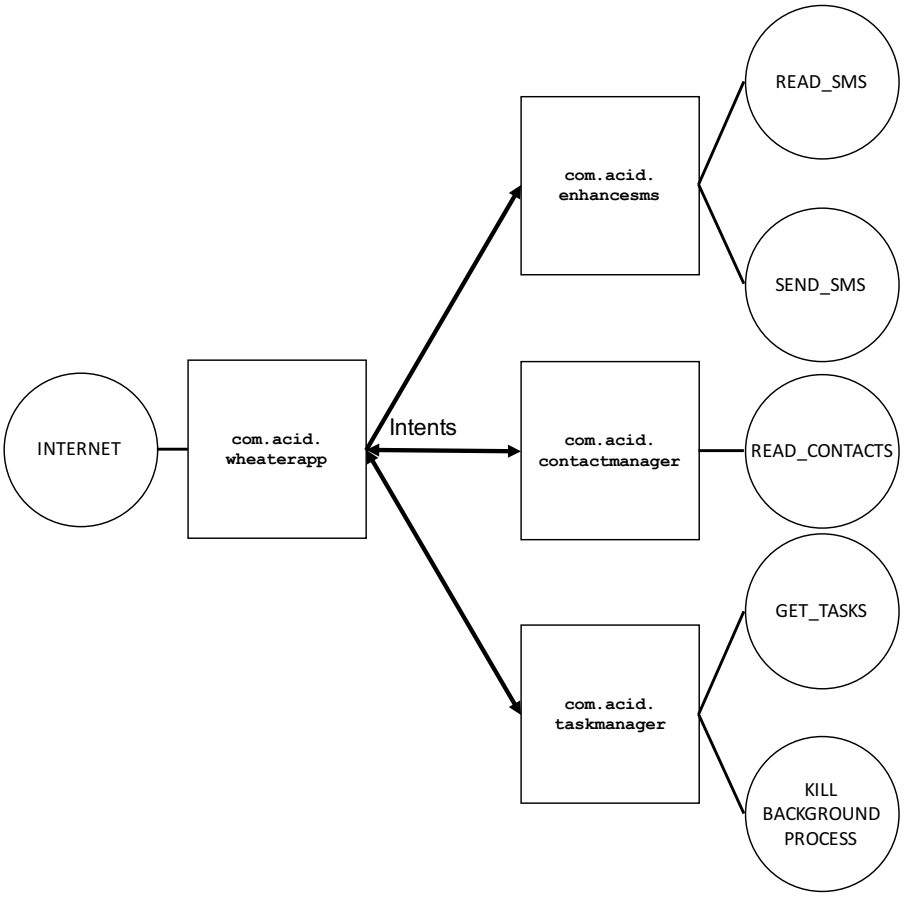

Figure 3: Graphical representation of the botnet client demonstration app set

Analysis of Prevalent Market apps

We have performed a preliminary analysis of a dataset of more than 9000 apps from the Intel knowledge base. These apps are classified as 'clean' . This is confirmed by both their prevalence and by Intel Security.

Most of the existing risk-evaluation metrics for apps look for permissions, reflection capabilities and presence of native code. In our preliminary analysis we look for non-overlapping permissions and inter-app communication capabilities that could enable collusion.

Permission Analysis

We have analyzed the conditional occurrence of permissions available by default to third-party apps from the 9000 app dataset. This is, how many times a permission p_x appears in an app when the permission p_y is declared. This shows if a permission is usually declared with other permissions or if the declaration of one permission reduces the possibility of having others declared.
Figure 4 shows a color map of the conditional occurrence of default permissions available to third party apps. A white coordinate (value 1) means that every time the permission p_y (in

row *y*) appears, the permission p_x (in column x) is also declared. A black coordinate indicates that permission p_x never appears with permission p_y in our dataset.

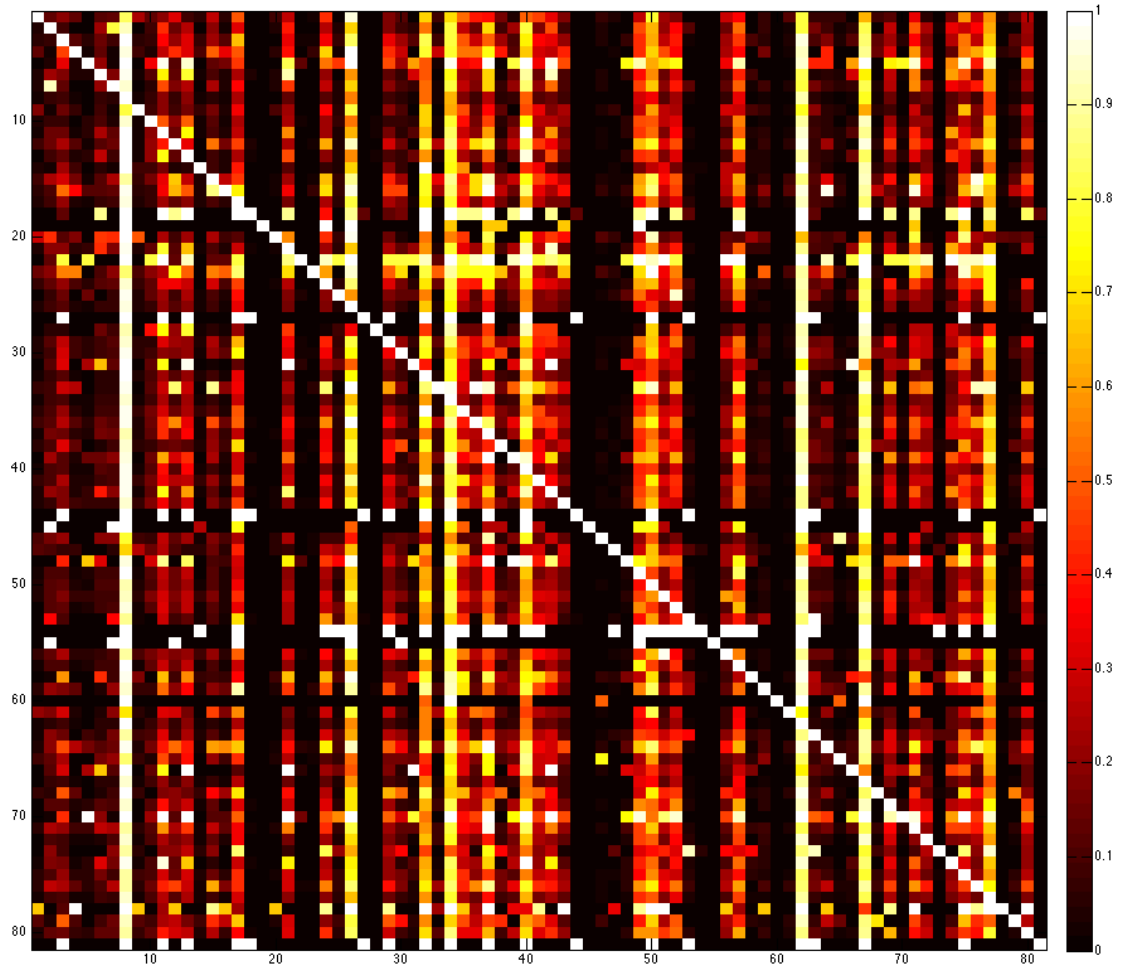

Figure 4: Color map of conditional occurrence of permissions given another permission. The color at position y,x represents the ratio of times that permission of column x appears everytime a permission of row y appears on an app.

For instance, permissions in rows 27 and 44 (READ_BOOKMARK_HISTORY and WRITE_BOOKMARK_HISTORY) match all their conditional occurrences. The WRITE_BOOKMARK_HISTORY, does not allow reading the history, so apps that want to manage bookmarks must declare both of them always. Seeing an app declaring only one of these permissions would be really suspicious.

The analysis of the app dataset also shows that Permission 8 (INTERNET), is declared almost anytime another permission is declared, except for a few cases (Table 4). However, it is not declared by all apps. Around 22% of the apps that declare the READ_USER_DICTIONARY permission do not require access to the Internet. Those apps could potentially collude with other apps that have Internet access to steal the user's dictionary. A similar circumstance happens with the access to external storage. 6% of the apps that declare this permission do not require access to the Internet.

Table 4: Conditional occurrence values for some of a subset of the permissions analyzed

	p_8	p_{19}	p_{46}	p_{62}	p_{67}	p_{78}
INTERNET (p_8)	1	0	0.01	0.91	0.76	0
BIND_INPUT_METHOD (p_{19})	1	1	0	1	0	0
READ_USER_DICTIONARY (p_{46})	0.78	0	1	0.80	0.78	0.79
ACCESS_NETWORK_STATE (p_{62})	0.98	0	0.01	1	0.79	0
WRITE_EXTERNAL_STORAGE (p_{67})	0.94	0	0.01	0.90	1	0
USE_SIP (p_{78})	1	0	0	1	0.44	1

Inter app communication analysis

In our preliminary study, we focus on external storage and intent-based communication only. External storage communication is detected by checking the applications' manifest files. Intent-based communications is detected using the *didfail* (Burket, Flynn, & Klieber, 2015) tool.

Due to the longer processing time required to extract the inter-app communication channels, we tested only 124750 random pairs selected from the clean set for intent based communications.

Our collusion analysis is based on a policy-based model. We use a predefined set of rules that describes the main threats created by mobile malware. The security policy is taken from (Enck, Machigar, & McDan, 2009) and the main rules are described in Table 5.

Table 5: Sample Kirin Rules to mitigate malware

Rule	An application must not have…
1	the SET_DEBUG_APP permission label
2	PHONE_STATE, RECORD_AUDIO and INTERNET permission labels
3	PROCESS_OUTGOING_CALL, RECORD_AUDIO and INTERNET permission labels
4	ACCESS_FINE_LOCATION, INTERNET and RECEIVE_BOOT_COMPLETE permission labels
5	ACCESS_COARSE_LOCATION, INTERNET and RECEIVE_BOOT_COMPLETE permission labels
6	RECEIVE_SMS and WRITE_SMS permission labels
7	SEND_SMS and WRITE_SMS permission labels
8	INSTALL_SHORTCUT and UNINSTALL_SHORTCUT permission labels
9	SET_PREFERRED_APPLICATION permission label and receive intents for CALL action

Results

Our policy-based model classified marked 7% of app pairs as having collusion potential. (Figure 5). These figures may include a lot of false positives. At the moment we do not inspect the details of potential inter-app communication. Therefore, our results do not identify the app pairs that might be already colluding, but the ones with potential to do it. As part of our future work, we have started to use the permission analysis, which is less computationally expensive, to optimize the pair selection for the communication analisys, which is more costly. With this method we aim to reduce the cost when looking for colluding apps, so more analysis and research efforts can be given to the more suspicious subset of app pairs.

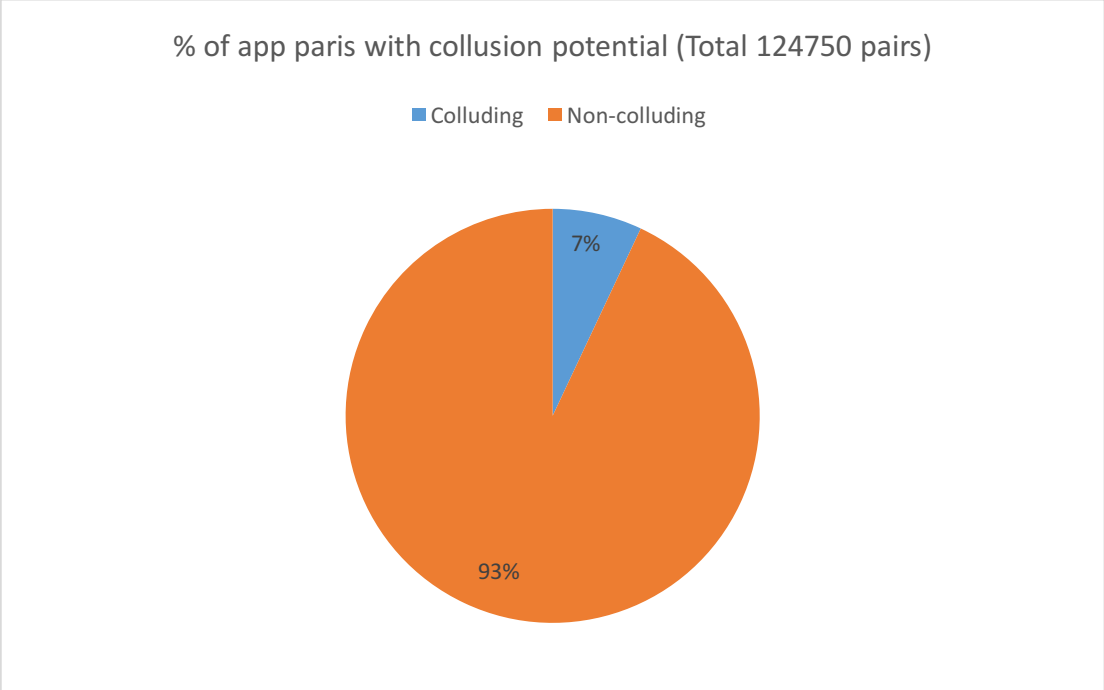

Figure 5: Percentage of app pairs with collusion potential

Detecting App Collusion

We have described a possible emerging threat for mobile devices where applications collude to achieve their goals. We have presented several examples of collusion and a preliminary study on publicly available apps. We show that a significant percentage of apps show collusion potential. Given the relatively small amount of apps analyzed (in comparison with the amounts of apps already available) there is still a need to refine the detection process, so analysis efforts can be focused on those app sets that are suspicious.

Some ways of improving our preliminary analysis would be to filter our app sets for those with specific permissions. For instance, an app that is missing the INTERNET permission would have more interest in colluding with other apps that already have that permission granted. Additionally, we have focused our efforts in the standard Android communication channels. Given that collusion is an attack designed to overpass current security measures, it would not be surprising if malware developers spent more effort on using stealthier communication channels such as covert `SharedPreferences` method.

The analysis of app sets is a much more complex and computationally expensive task. Instead of only looking at one app, it is also necessary to look for other apps that can potentially collude with it. Given the high number of Android apps already available and new apps created every day, overcoming this issue is a challenge. Performing device-level analysis would reduce this complexity. Every time an app is going to be installed, security software could check the collusion potential with the rest of the apps stored in the same user device. This way, the number of app combinations may be drastically reduced.

Our ongoing work is pursuing in two directions. First, we are creating more tools and methods to inspect the nature of the communications between apps. Second, we are aiming to reduce the complexity of analyzing different combinations of apps to detect collusion potential. In this way, we will be able to protect against these kinds of attacks before they are deployed.

References

Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., & Shastry, B. (2011). Practical and lightweight domain isolation on Android. *roceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (pp. 51-62). ACM.

Burket, J., Flynn, L., & Klieber, W. (2015). *Making Didfail Succeed: Enhancing the Cert Static Taint Analyzer for Android App Sets.* Carnegie Mellon University, Pittsburgh.

Chin, E., Porter, A., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communications in Android. *Proceedings of the 9th international conference on Mobile systems, applica- tions, and services,* (pp. 239-252). ACM.

Elish, K. O., Yao, D. D., & Ryder, B. G. (2015). On the need of precise inter-app icc classi- fication for detecting android malware collusions. *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy.*

Enck, W., Machigar, O., & McDan, P. (2009). On lightweight mobile phone application certification. *Proceedings of the 16th ACM conference on Computer and communications security* (pp. 235-245). ACM.

Hardy, N. (1988). The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review , 22* (4), 36–38.

Jerzy, N., & Pearson, E. S. (1992). *On the problem of the most efficient tests of statistical hypotheses.* New York: Springer.

Marforio, C., Francillon, A., & Capkun, S. (2011). Application collusion attack on the permission-based security model and its implications for modern smartphone systems.

Marforio, C., Ritzdorf, H., Francillon, A., & Capkun, S. (2012). Anal- ysis of the communication between colluding applications on modern smart- phones. *Proceedings of the 28th Annual Computer Security Applications Con- ference* (pp. 51-60). ACM.

Porter Felt, A., Wang, H. J., Moshchuk, A., Hanna, S., & Chin, E. (2011). Permission re-delegation: Attacks and defenses. *USENIX Security Symposium.* USENIX.

Schlegel, R., Zhang, K., Zhou, X.-y., Intwala, M., Kapadia, A., & Wang, X. (2011). Soundcomber: A stealthy and context-aware sound trojan for smartphones. *NDSS*, *11*, pp. 17-33.