Detection of App Collusion Potential Using Logic Programming

Jorge Blasco[a], Thomas M. Chen[b], Igor Muttik[c], Markus Roggenbach[d]

[a] *Jorge.BlascoAlis@rhul.ac.uk, Information Security Group. Royal Holloway, University of London*
[b] *Tom.Chen.1@city.ac.uk, Electronic and Electrical Engineering Department. City, University of London*
[c] *igor.muttik@cybercurio.com, Cyber Curio*
[d] *M.Roggenbach@swansea.ac.uk, Department of Computer Science, Swansea University*

Abstract

Mobile devices pose a particular security risk because they hold personal details (accounts, locations, contacts, photos) and have capabilities potentially exploitable for eavesdropping (cameras/microphone, wireless connections). The Android operating system is designed with a number of built-in security features such as application sandboxing and permission-based access control. Unfortunately, these restrictions can be bypassed, without the user noticing, by colluding apps whose combined permissions allow them to carry out attacks that neither app is able to execute by itself. While the possibility of app collusion was first warned in 2011, it has been unclear if collusion is used by malware in the wild due to a lack of suitable detection methods and tools. This paper describes how we found the first collusion in the wild. We also present a strategy for detecting collusions and its implementation in Prolog that allowed us to make this discovery. Our detection strategy is grounded in concise definitions of collusion and the concept of ASR (Access-Send-Receive) signatures. The methodology is supported by statistical evidence. Our approach scales and is applicable to inclusion into professional malware detection systems: we applied it to a set of more than 50,000 apps collected in the wild. Code samples of our tool as well as the detected malware are available.

Keywords: Android, Collusion, Malware, MoPlus

1. Introduction

Mobile devices, such as smartphones and tables are pervasive in modern everyday life. The number of smartphones in use is predicted to grow from 2.6 billion in 2016 to 6.1 billion in 2020 [1]. One reason for this fast adoption is the extensive ecosystem of apps which enable a very wide range of functions: taking photos, sending messages, financial transactions, as well as a great variety of other uses. Smartphones hold a great deal of personal information (e.g., photos, financial data, credentials, messages) making them very appealing targets for criminals who often employ malicious apps to steal sensitive information [2], extort users [3], or misuse the device services for their own purposes [4].

To mitigate these threats, mobile operating systems offer a multi-sandbox environment where each app is executed in isolation from the rest. This isolation is intended to limit the harm from any potential malicious activity of each app. An obvious attack to get around this restriction is to execute a privilege escalation exploit or, even simpler, employ social engineering and convince the user to grant additional permissions to the app. This usually works quite well because users are generally unaware of the risks associated with granting permissions to apps [5]. However, malware detection techniques keep improving, and it is getting more difficult for attackers to deploy their creations without being detected. Furthermore, attackers are interested in malware to stay longer on phones. As the short time required to identify the latest wave of clones of popular apps like "WhatsApp" [6] illustrates, social engineering attacks do not necessarily provide longer access to an attacked phone.

Attackers have a better chance of evading detection in both pre-deployment and after-deployment scenarios by using app collusion where the malicious activity is split across multiple apps and coordinated through inter-app communications (IAC). This kind of attack is possible because sandboxed systems, such as Android, are designed to prevent threats from individual apps. However, they do not restrict or monitor inter-app communications, and therefore they would fail to protect from multiple apps cooperating in order to achieve a malicious goal. Most malware analysis systems, such as antivirus software for smartphones, also check apps individually only.

Collusion attacks are more complex than single malicious app attacks. In order for such an attack to be successful, the attacker must be able to embed colluding code in more than one app, and have at least two of those

apps installed on the victim's device. However, there are strategies, such as infecting development libraries and SDKs (e.g. the XCode Ghost case [7]) that can facilitate this task.

While collusion is not a common attack vector today, we believe this is likely to grow because of constant enhancements in efficacy of static and dynamic analysis of individual apps as well as in calculating their reputations.

1.1. Contributions

In this paper, we present a methodology to detect the potential for app collusion by using logic programming. For each app, we extract what we call an ASR (Access-Send-Receive) signature. Then Prolog rules are used to characterize app collusion behaviours. These rules reflect two aspects of collusion: access to protected resources and communication channels between apps. Using the rules and the set of ASR signatures codified as Prolog facts, it is possible to obtain a list of potentially colluding app sets from large datasets of apps.

We have validated the approach against a set of colluding apps which we crafted ourselves for testing purposes. We further applied it to an unclassified set of more than 50,000 apps collected in the wild. Our approach allowed us to shed light on how apps in the Android ecosystem communicate. We could also identify a set of apps in the wild all including a malicious SDK which used collusion to maximize the effects of its payload [8].

As with any rule-based method, our approach is limited to being able to flag only known types of collusion which are represented in the rules. In principle, an alternative approach such as anomaly detection might be able to detect new types of collusion that known rules can not, but it is not possible to confirm the efficacy of anomaly detection until new collusion attacks occur in the wild (or become known).

This paper provides a thorough and detailed documentation and discussion of our approach, including a description of our detection strategy, our implementation, and how to scale the application of our tool to app sets consisting of many tens of thousands of apps.

The remainder of this paper is structured as follows. Section 2 reviews the Android security model focusing mainly on the aspects related to app collusion. In Section 3 we propose a definition of app collusion and describe possible communication channels that can be used by colluding apps. Section 4 describes our approach in detail and gives validation of our approach against a set of manually created colluding apps. Section 5 elaborates on how our

method can scale up. Section 6 offers experimental results with a relatively large dataset of apps collected in the wild. Section 7 describes a group of colluding apps which we discovered in the wild by using our method. To the best of our knowledge, this is the first example of app collusion found in the wild. Section 8 reviews previous work done to detect and protect against app collusion.

2. The Android Operating System

The Android operating system model is designed to protect users, apps, the device and the network from malicious code. By default, all third party apps are considered untrusted by the OS and run inside a sandbox that isolates them from any sensitive resources and from all other apps. Until Android 4.3, the sandboxing mechanism was implemented by assigning a different Linux user id (UID) to each app and configuring file permissions accordingly. Since Android 4.4 SELinux domains are used in addition to the Linux UID so apps can only access files inside their sandbox, as these are the only ones in their domain.

Access to sensitive resources outside the app sandbox is possible by using APIs provided by the operating system. Calls to these APIs are managed by a permission system, which has a deny-by-default policy. Apps that want to access sensitive resources, must include a permission declaration inside their `AndroidManifest.xml` file. When an app is being installed (in 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 requested by the app.

Starting with Android 6.0, apps can ask for permissions at runtime and users have a choice of granting or denying each permission individually. However, permissions must still be declared in the manifest file. For a more detailed description of Android security features, we refer the reader to [9, 10].

2.1. App Components

Android apps are built with the following components:

- *Activities* represent screens of the user interface and allow the user to interact with the app. Activities run only in the foreground. Apps are generally composed of a set of activities.

- *Services* execute operations in the background. They are generally used by other components of the app to perform long-running tasks: listening to incoming connections, downloading a file, etc.

- *Broadcast receivers* respond to messages that are sent through `Intent` objects, by the same or other apps.

- *Content providers* manage access of other apps to the app's own data. Apps with content providers enable other apps to read and write their local data.

In order to be reachable by other apps, components must be declared as *exported* in the app manifest file.

2.2. Communications

Besides the standard Unix files, sockets, etc., Android offers three inter-process communication (IPC) mechanisms:

- The *Binder* is a remote procedure call mechanism designed to enable fast and efficient IPC between processes. The Binder Framework uses a server-client architecture. It is implemented as a Linux driver, allowing communications across sandbox boundaries. This allows the operating system to mediate communication through Binder. The rest of the Android IPC (Intents and Content Providers) are, in fact, abstractions based on the Binder.

- An *Intent* is a messaging object which is used to request actions from other apps' components. These may belong to the same or different apps. Intents can be explicit or implicit. Explicit intents target specific activities or services. Implicit intents target generic actions that can be performed by many different activities (e.g. send a message, open a web link, etc.). Activities, services and broadcast receivers advertise the intents which they want to handle by declaring a set of `IntentFilters`. For activities and services, intent filters must be declared in the app's manifest. Broadcast receivers can also register their intent filters through API calls during execution.

- *Content Providers* are used to offer other apps a method to access their own structured data. Content providers store information in one or more tables, similarly to relational databases. Apps access data of

content providers using `ContentResolver` objects. A content provider offers methods, which can be called by other apps, not only to read data, but also to update, create and delete information encapsulated in the content provider object.

None of these three IPC mechanisms, which all are standard in Android, is covered by the mandatory access control offered by SELinux [11]. As a result, apps can share any kind of information by using standard IPC without any restriction. To avoid security problems, Android allows apps communicating through IPC to request specific permissions from any app that wants to communicate with them. As an example, Android includes a Contact Provider to interact with the device's contact list. Apps accessing this provider need to declare `READ_CONTACTS` or `WRITE_CONTACTS` permissions in their manifest. Similarly, apps using Intents to start phone calls require the `CALL_PHONE` permission.

Unfortunately, Android does not enforce this protection mechanism. It is left to app developers to decide if they want to apply it. Consequently, permission-protected resources are potentially exposed. This fact can be exploited to build colluding apps which access sensitive resources without permission. Apps might also communicate in order to aggregate permissions necessary to perform malicious actions which individual apps would not be able to perform.

3. App Collusion

The first demonstration of app collusion was *Soundcomber*, a proof-of-concept malware described in 2011 [12]. It was comprised of two apps which used inter-app communications to steal the user's banking credentials. Soundcomber shows the limitations of the Android permission model to protect against apps that collude to aggregate their permissions [13]. Although collusion has inherently a malicious component, sometimes it is hard to distinguish collusion from collaboration.

A direct frontal attack to look for collusion of pairs, triplets, and larger sets of Android apps is not practical because of the size of the search space. Thus, in this work, we develop an effective filtering system to quickly analyze relatively large app sets to detect collusion potential, and thus narrow down the search space to help a security analyst (or other more computationally expensive automatic tools) to focus their efforts on the most suspicious app sets.

3.1. Definitions

In this work, collusion refers to the ability of a set of apps to carry out an attack through collaboration. We assume that colluding apps carry out the same malicious actions as single apps do, such as information theft, money theft, service misuse, sabotage, denial of service, ransom, etc. [14].

To the best of the authors' knowledge, there is no evidence that collusion can create new threats in mobile operating systems, as they depend on the assets, which remain the same regardless of how the malicious code is sliced and diced. However, collusion can drastically change the appearance of the attack which may severely complicate its detectability during static code inspection and during dynamic behavioural analysis. In the Soundcomber scenario, collusion is simply used to make an information leakage attack more stealthy.

Additionally, malicious apps can also take advantage of collusion for coordination and synchronization. In this case, each app could be considered to be malicious on its own. When colluding apps are installed on the same device, they may coordinate their actions to improve their capabilities (compared to acting on their own). An example could be one app encrypting users' documents and photos and another one asking for a payment to reverse the action (ransomware). Considering this, we base the meaning of "collusion potential" on the following definitions:

Definition 1 (Action). *An Action a is an operation that can be executed via the operating system (Android) API such as recording audio, sending data through the Internet, receiving data from another app, etc. Actions can be categorised in three groups:*

- ***A**ccess actions involve access to system-protected resources (e.g. record audio).*
- ***S**end actions allow apps to send information to other apps on the same device (e.g. an implicit intent).*
- ***R**eceive actions allow apps to receive information from other apps (e.g declaration of a broadcast receiver).*

Definition 2 (ASR Signature). *An ASR signature is a triplet* $(A_{app}, S_{app}, R_{app})$ *where* A_{app} *represents a set of actions that allow app to access system-protected resources, and* S_{app} *and* R_{app} *represent sets of actions included in app to send*

and receive information from other apps (using inter-app communications), respectively. Note that we consider a capability as the ability to execute an action.

Definition 3 (Threat). *A threat is a sequence of actions* $\langle a_1, a_2, \cdots, a_n \rangle$. *In this work, we consider threats to be the ones that can be realised by a single app. Let* τ *denote the set of all these threats.*

Definition 4 (Collusion Potential). *We say that a set* S *consisting of at least two apps has collusion potential iff the apps in* S *together can execute a sequence Seq of actions such that:*

1. *Seq restricted to Access actions is a sequence in* τ*;*
2. *Seq is collectively executed involving all apps in* S*, i.e., each app in* S *executes at least one action in Seq; and*
3. *considering the Send and Receive actions in Seq, all apps in* S *are connected through communication channels. That is, it is possible to build a directed graph* $G = (S, C)$ *where the elements of* C *are pairs of Send and Receive actions, in which there are no unreachable nodes, i.e. apps in* S*.*

Our definition of collusion potential highlights two steps required for collusion: execution of a malicious activity (threat) and the need to communicate between the apps executing the actions of the treat. Malicious actions and types of threats that can be executed by smartphone malware have been extensively studied by researchers [14, 15]. Due to the high level description of the actions it may happen that an app set marked as having collusion potential is taking advantage of another app (via permission re-delegation attack [16]) or just collaborating with it. In this work we consider that both cases should be detected and highlighted as having collusion potential.

As already noted by other researchers [17], to effectively differentiate between malicious and non-malicious behaviours one needs to be able to inspect (apart from the app code itself, that is) the application description, developer's intentions and system implementations. This is very difficult to achieve during static analysis. It is up to the security analyst (probably with the help of other taint analysis tools) to decide whether (1) one app is executing a permission re-delegation attack over another app or (2) if they are merely collaborating (with the user knowledge as described in the documentation) or (3) if they are actually colluding. Therefore, in these

three cases, apps may exhibit exactly the same behaviour, and so must be considered to have collusion potential (i.e. in all these cases apps have the capability to maliciously collude).

In the next section we review the main communication channels that can be used by colluding apps. The development of the definition into a Prolog program to detect collusion potential is described in section 4.

3.2. Communication Channels for App Collusion

Colluding apps execute *Send* and *Receive* actions to synchronize their actions and achieve their common goal. In some cases, these apps will use standard communication APIs (as described previously in Section 2). Colluding apps may also use APIs that are not specifically meant to be used for inter-app communication but allow the creation of covert channels. The following list summarises all the actions that can be included in the *Send* and *Receive* groups:

- *Intents* can be used by colluding apps to share information. Broadcast receivers and services allow apps to exchange data without user intervention. In addition, intents used to open activities can include information that is not necessarily required to present the new activity to the user.

- Malicious apps can use *content providers* as a dropbox to exchange information. This runs the risk of being visible to the user (e.g. creating a new contact to exchange information). Access to system content providers requires apps to request permissions (e.g. `WRITE_CONTACTS` for the contact database).

- *External storage* of an Android device can also be used as a shared dropbox to exchange information. External storage is generally available through a USB connection, SD card or, sometimes, even via non-removable storage. Apps accessing the external storage need to declare the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`, depending on the required access. Files in the external storage can be accessed using the common file access API.

- *Shared preferences* are an Android feature that allows apps to store key-value pairs of data. Although it is not intended for inter-app communication, apps can use key-value pairs to exchange information if

proper flags are defined (`WORLD_READABLE` or `WORLD_WRITABLE`) when accessing and storing data. Since the adoption of SELinux (Android 5.0), apps cannot access the world readable files of other apps, as they are confined to different SELinux domains.

- Colluding apps can also use standard *Unix sockets* to communicate through the local network interface. Apps can use sockets opened to localhost to communicate as if they were communicating through the network. Communication between two apps that is mediated by an external server is not generally counted as collusion, because the communication happens outside the device domain.

- *Covert channels* may take advantage of APIs or features offered by the operating system to enable communication between processes [12, 18]. In Android, this includes publicly readable and writable settings (e.g. speaker volume level) and capturing broadcast intents generated by the system. Additionally, processes can take advantage of covert channels present in most computing systems like file locks, process enumeration, free storage or memory space and CPU usage.

Table 1 summarises the keywords that can be used to enable *Send* and *Receive* actions through overt channels in Android. The presence of these keywords indicates that an app is able to communicate to other apps within the same device.

Overt channel	Action	Object	Keywords
Intent	Send	Java	(`startActivity` *or* `startServer`) *and* (`putExtra` *or* `setData`)
	Receive	XML	`intent-filter`
BroadcastReceivers	Send	Java	`sendBroadcast` *and* (`putExtra` *or* `setData`)
	Receive	Java	`registerReceiver`
	Receive	XML	`receiver` *and* `intent-filter`
Intent Results	Send	Java	`startActivityForResult`
	Receive	Java	`provider` *and* `grantUriPermissions`
External Storage	Send	XML	`WRITE_EXTERNAL_STORAGE`
	Receive	XML	`READ_EXTERNAL_STORAGE`
Content Providers	Send	Java	`getContentResolver`
	Receive	Java	`getContentResolver`
SharedPreferences	Send	Java	`getSharedPreferences` *and* `edit` *and* `put`
	Receive	Java	`getSharedPreferences` *and* `get`

Table 1: Keywords enabling communication between apps using overt channels.

Table 2 shows a similar table for covert channels in Android documented in the literature. These API calls have two effects. First, they perform the action they were designed for (e.g. increase volume) but, at the same time, these API calls can be used to transmit information.

Covert channel	Action	Object	Keywords
Audio settings [12]	Send	Java	`Context.AUDIO_SERVICE` *and* (`adjustStreamVolume` *or* `adjustSuggestedStreamVolume` *or* `adjustVolume`)
	Receive	Java	`Context.AUDIO_SERVICE` *and*
			`getStreamVolume`
Settings broadcast [12]	Send	Java	`Context.AUDIO_SERVICE` *and* `setVibrateSetting`
	Receive	Java or XML	`RINGER_MODE_CHANGED`
Wake lock [12, 18]	Send	Java	(`Wakelock` *and* `acquire`) *or* `WakefulBroadcastReceiver`
	Receive	Java	`ACTION_SCREEN_ON` *and* `ACTION_SCREEN_OFF`
File lock [12, 18]	Send	Java	`FileLock` *and* `lock` *and* `release`
	Receive	Java	`isValid`
Proc. enumeration [18]	Send	C	`fork` *or* `pthread_create`
	Receive	C	`proc`
		XML	`GET_TASKS`
		Java	`ActivityManager` *and* `getRunningServices`
Socket discovery [18]	Send	Java	`Socket`
	Receiver	Java	`Socket` *and* `isClosed`
Free space [18]	Send	Java	*
	Receive	Java	`StatFs` *and* `getAvailableBlocks`
		Java	`MemoryInfo` *and* `availMem`
CPU Usage [18]	Send	Java	*
	Receive	Java	*

Table 2: Keywords enabling communication actions between apps using covert channels.

4. Detecting Collusion Potential

Mobile apps can be downloaded from the web or app markets such as Google Play. These apps are analyzed by market operators and anti-malware services that constantly crawl these markets. A combination of static and dynamic analysis techniques as well as statistical and machine learning methods are used to establish reputation and risk values for each app. However, all these techniques consider apps in isolation and neglect to take into account other apps that could be installed on the same device. This hinders detection of possible collusions between apps installed on the same device.

Our approach aims to extend these app analysing services by also considering the *potential* for collusion among sets of apps; the reputation and risk of an app are measured not only in terms of its own features, but also taking into account additional capabilities when installed together with other apps on the same device.

To do so, for each app we extract the actions which it is able to perform as ASR signatures. These are described as Prolog facts. In a similar way, collusion potential is described as a set of logic rules that are composed by Prolog facts. Collusion rules can be applied to query for apps that may be potentially colluding. A detailed view of the process is given in the following sections.

4.1. Extracting ASR Signatures

ASR signatures are extracted by means of a static analysis of the app manifest and app code. In this work we consider the usage of (i) implicit *intents*, (ii) *shared preferences* and (iii) *external storage* for communication, i.e., a subset of the channels listed in Section 3.2. Therefore, the ASR signature of an app is a combination of all permissions, intents, shared preferences and external storage channels that can be used by an app to send or receive information. Noted that, as described in Section 3, one mechanism can be utilized to create several communication channels. For instance, an app that stores information in two shared preferences files, would be creating two communication channels.

To analyze app code, we have extended Androguard [19], a reverse engineering tool for Android apps written in Python[1]. Our extension looks up API calls involved in the creation and broadcast of intents and broadcast receivers and the access and modification of shared preferences files. Parameters that specify the communication channel for each method are tracked back through the code. We trace back the value of the `action` parameter for each broadcast intent and corresponding intent filters. In the case of shared preferences, we track the name of the shared preferences file. As with any static analysis tool, our tool is not able to trace back values that are dynamically defined. In those cases, we return the API call path that generates the value.

[1]Our code is available at `https://github.com/acidrepo/collusion_potential_detector`

Table 3: ASR signature of the SMS app (id 4) that is part of the Botnet group.

A_{id4}	
Permissions	READ_SMS
	SEND_SMS
$S_{id4} = \emptyset$	
R_{id4}	
Intent	SMS_SENT
	sms

The app manifest is analyzed to identify usage of external storage or static broadcast receivers. We are not able to obtain the specific channel used by apps through external storage. This requires identifying all API calls that can modify the external storage file system. PScout is able to identify the Android library API calls that require `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions [20]. However, standard I/O calls are not included in this mapping; this task is left for future work.

As an example, Table 3 shows the ASR signature of a simple artificial colluding app that sends premium SMS messages. This app requires `READ_SMS` to display the user's SMS and `SEND_SMS` to send SMS. This permission is abused to send SMS messages when an intent with the action `sms` is received. Details about this app, and the rest of the apps created for validation purposes, can be found on section 4.4.

4.2. Characterizing Collusion Potential with Logic Rules

Our approach to detect collusion potential utilizes logic programming in Prolog. We have created a Prolog program, the *ACiD* (**A**pplication **C**ollus**i**on **D**etection) rule set, that defines when a set of apps shows collusion potential.

Access actions have been categorized into four high level actions: (i) accessing sensitive information (ii) using an API that can incur a financial loss (iii) controlling device services (e.g. camera) and (iv) sending information outside the device. These actions are characterised by permissions and API calls which are mapped to one or more of the four high level actions. For example, an app that declares the `INTERNET` permission will be capable of sending information outside the device:

$$access(App, P_{Internet}) \rightarrow information\_outside(App)$$

Similarly:

$$\begin{aligned} access(App, P_{Read\ contacts}) &\rightarrow sensitive\_info(App) \\ access(App, P_{Send\ SMS}) &\rightarrow money(App) \\ access(App, P_{Kill\ process}) &\rightarrow control\_service(App) \end{aligned}$$

In Android 4.3, overall 35 permissions can be used to access sensitive information; 12 to send information outside the device; 3 to execute financially-sensitive APIs (invoking actions which can cost money, e.g. making a phone call or sending an SMS); and 39 to control device services. The complete mapping of permissions to actions can be found in the project repository [2].

Send and *Receive* actions are characterised by specific API calls offered by the Android OS. We create a fact for each to describe that communication action. When using `Intents` and `SharedPreferences`, we are able to specify the communication channel using the intent actions and preference file, respectively. As an example, if an app sends a `BroadcastIntent` with an action `SEND_FILE` we consider the following:

$$\begin{aligned} &send\_broadcast(App, Intent_{send\_file}) \\ &\quad \rightarrow send(App, Intent_{send\_file}) \end{aligned}$$

We consider that two apps communicate if one of them is able to *send* and the other to *receive* via the same channel.

$$\begin{aligned} &send(App_a, channel) \wedge receive(App_b, channel) \rightarrow \\ &\quad communicate(App_a, App_b, channel) \end{aligned}$$

Note that communication is directed, i.e., information flows from App_a to App_b.

Finally, each of the threats is characterised by a sequence of actions. Our threat set τ considers information theft, money theft and service misuse. Specifically, we consider that two apps have collusion potential to execute an information theft when one of them has access to sensitive information and

[2] `https://github.com/acidrepo/collusion_potential_detector`

communicates with another app which can do external communications:

$$\begin{aligned} sensitive\_info(App_a) \quad & \wedge \\ information\_outside(App_b) \quad & \wedge \\ communicate(App_a, App_b, channel) \quad & \rightarrow \\ information\_collusion(App_a, App_b) & \end{aligned}$$

We consider that two apps have potential to collude for money theft when one app has access to cost sensitive APIs and receives information from another app:

$$\begin{aligned} money(App_b) \quad & \wedge \\ communicate(App_a, App_b, channel) \quad & \rightarrow \\ money\_collusion(App_a, App_b) & \end{aligned}$$

An internet connection in App_a would allow a server to send commands to an app with access to cost sensitive APIs:

$$\begin{aligned} information\_outside(App_a) \quad & \wedge \\ money(App_b) \quad & \wedge \\ communicate(App_a, App_b, channel) \quad & \rightarrow \\ money\_collusion(App_a, App_b) & \end{aligned}$$

In a similar sense, this same app could also send commands from a C&C server to other apps that have access to device services:

$$\begin{aligned} information\_outside(App_a) \quad & \wedge \\ control\_service(App_b) \quad & \wedge \\ communicate(App_a, App_b, channel) \quad & \rightarrow \\ service\_collusion(App_a, App_b) & \end{aligned}$$

4.3. ACiD Rule Set in Prolog

We have translated the ACiD rules into a Prolog program including rules required to identify communication paths (and specific channels) between applications. Then, once the ASR signatures have been extracted from an app set, they can be translated into Prolog facts to be part of the Prolog program that is executed to find collusion potential.

On the code level in Prolog, we represent ASR signature as facts, see Listing 1 for an excerpt. The package `predicate` relates the name of the app with the filename of the apk, the `access`, `send`, and `receive` predicates gather information on what permissions and keywords were found in the code of an app.

```
package('bbc.mobile.weather','
    BBC Weather_androidappsapk.co_215.apk').
...

access('bbc.mobile.weather',
       'android.permission.INTERNET').
access('bbc.mobile.weather',
       'android.permission.ACCESS_NETWORK_STATE').
...
send('bbc.mobile.weather',
       'i_android.intent.action.GET_CONTENT').
send('bbc.mobile.weather',
       'i_android.settings.APPLICATION_DETAILS_SETTINGS').
...
receive('bbc.mobile.weather',
       'i_android.net.conn.CONNECTIVITY_CHANGE').
receive('bbc.mobile.weather',
       'i_android.nfc.action.NDEF_DISCOVERED').
...
```

Listing 1: Facts on the BBC Weather App

In general, a Prolog fact `access(app, permission)` is created for every permission used by an app. For every channel sending information outside the app sandbox, we generate a `send(app,channel)` fact and `receive(app,channel)` for all channels receiving information from outside the sandbox. The extraction of ASR signatures is carried out by a modified version of Androguard. We created a Python script to translate the obtained signatures into Prolog facts. This same script is used to generate a version of the Prolog progam that includes both the facts and the ACiD rule set. If new apps are analyzed, the generated file can be updated with the additional facts extracted from the apps.

The complete Prolog program is structured in seven sections:

- Package facts: map the file names of the APKs to the package name of the app inside each APK file.
- Access facts: codify the accessed permissions, A_{app}, extracted from the apps in the analysed set.

- Communication facts: codify the communication facts, S_{app} and R_{app}, extracted from the apps as communication actions in *com*.
- Access to action rules: map access facts, A_{app}, to application actions, Act_{prolog}.
- Communication rules: detect when two or more apps are communicating based on the extracted S_{app} and R_{app}.
- Collusion rules: describe the possible collusion behaviours. These rules are fired when action and communications facts are fired from their respective rules.
- Channel rules: allow to detect which channels are using a set of colluding apps, once the colluding app set has been detected with the previous rules.

A Prolog predicate (`q :- p`) describes a logical rule of the form $p \rightarrow q$. Prolog uses *modus ponens* to evaluate queries and look for results. If `p` is true, then it will consider `q` to be also true. The identification of communication paths between apps is performed by using recursive Prolog predicates (Listing 2). The base case (first rule) identifies when two apps are communicating. This is, if `AppA` sends information through `Channel` and `AppB` receives information from the same channel, it means they communicate (`comm_l(AppA,AppB,2,_,[])`). The recursive predicates (last two rules) add more apps to the communication path. To avoid circular paths, all rules check if the app that is being analysed is already a member of the path (`nonmember`).

```
comm_l(AppA,AppB,2,_,[]) :− trans(AppA,Channel), receive(AppB,Channel), AppA\=AppB.
comm_l(AppA,AppB,Length,[],[AppD|Rest]) :− Length > 2,send(AppA,Channel),
      receive(AppD,Channel), AppA\=AppD, PrevL is Length −1,
      comm_l(AppD,AppB,PrevL,[AppA],Rest), AppA\=AppB.
comm_l(AppA,AppB,Length,Visited,[AppD|Rest]):− Length > 2, send(AppA,Channel),
      receive(AppD,Channel), AppA\=AppD, nonmember(AppD,Visited), PrevL is Length − 1,
      comm_l(AppD,AppB,PrevL,[AppA|Visited],Rest), AppA\=AppB.
```

Listing 2: Communication rules

The channel rules allow, once a collusion path has been obtained, to extract the list of communication channels used by the apps (Listing 3). Similarly, the first predicate (first rule) saves the channel used when two apps are communicating. The second predicate looks recursively for the rest

of the channels. These rules facilitate the security analyst task to investigate how the potential collusion can happen.

```
chanl(AppA,AppB,[],Channel) :- send(AppA,Channel),receive(AppB,Channel), AppA\=AppB.
chanl(AppA,AppB,[AppD|Rest],[Channel|Channels]) :- send(AppA,Channel), receive(AppD,Channel),
    chanl(AppD,AppB,Rest,Channels).
```

Listing 3: Channel identification rules

4.4. Validation

We performed an initial validation by running our tool against a set of eleven specifically developed artificial apps that include colluding and non-colluding apps[3] which are summarized in Table 4. We decided to use apps developed by ourselves for two reasons. First, to the best of our knowledge, no colluding apps had been identified in the wild at that time. Thus, we lacked a set of previously known positive examples. Second, even if there were apps identified as colluding, we could not be 100% certain about the quality of a non-colluding set: even apps downloaded from a reputable market might be colluding, i.e., we lacked a set of previously known negative examples.

There are nine colluding apps that have been developed to cover all collusion scenarios described in Section 3. They belonged to three groups:

- The **Document Extractor** group is comprised of two apps. One of the apps in the group looks for sensitive documents (txt, pdf, db, xls, etc.) on the external storage (app_1). This information is shared with app_2 using the shared preferences. The information received is sent to a remote server.

- The **Botnet** group is comprised of four apps. One of the apps (app_4) acts as a relay that receives orders from the command and control center. The other colluding apps execute commands depending on their requested permissions. They are capable of sending SMS messages (app_4), stealing the user's contacts (app_5) and starting and stopping tasks (app_6). This group uses intents as communication channel.

- The **Contact Extractor** group is comprised of three apps. This group sends the device's address book to a remote server. The first app (app_7) reads the contacts from the address book, the second (app_8) forwards

[3]Due to the malicious nature of the apps, they are only available upon request.

Table 4: Summary of colluding app groups included in our basic app set.

Group	Id	Threats	Permissions	Coll. with	Channel
Document Extractor	1	Inf. theft	READ_EXTERNAL_STORAGE	2	Shared Prefs.
	2		INTERNET	1	Shared Prefs.
Botnet	3	Inf. theft	INTERNET	4,5,6	Intents
	4	Service misuse	READ_SMS SEND_SMS	3	Intents
	5		READ_CONTACTS	3	Intents
	6	Money theft	GET_TASKS KILL_BACKGROUND_PROCESSES	3	Intents
Contact Extractor	7	Inf. theft	READ_CONTACTS	8,9	Intents
	8		WRITE_EXTERNAL_STORAGE	7,9	Ext. Storage
	9		INTERNET READ_EXTERNAL_STORAGE	7,8	Intents Ext. Storage
Non Colluding	10	-	-	-	-
	11	-	INTERNET	-	-

them to the third (app_9), which sends them to the Internet. This group uses intents and the external storage as communication channels.

In addition to the colluding apps, the validation set included two non-colluding apps. These are a document viewer (app_{10}) and an information sharing app (app_{11}). The first app displays different file types on the device screen and uses other apps (through an intent with the action `android.intent.action.SEND`) to share their uniform resource identifier. The second app receives text (through the same action) and sends it to a remote server.

4.5. Validation Run

Table 5 shows the results obtained from analyzing the crafted colluding app set. "Dark red club" entries show when we detect collusion potential. As an example, the entry in row 1, column 2 means: the program detects that app_1 sends information to app_2, and these two apps collude to perform "information theft". As we take communication direction into consideration, the resulting matrix is non-symmetric, e.g., there is no entry in row 2, column 1. Additionally, our approach is able to identify transitive collusion attacks (i.e. app_7 colluding with app_9 through app_8).

Table 5: Collusion Matrix of the Prolog program. ♣ = Information theft. $ = Money theft. ♠ = Service misuse. ♣, $, ♠ = Benign showing collusion potential.

app	1	2	3	4	5	6	7	8	9	10	11
1		♣								♣	♣
2											
3				$♣	♠	♠					
4											
5			♣							♣	♣
6			♣	♣							
7		♣						♣	♣	♣	♣
8									♣		
9											
10											♣
11											

"Gold club" symbol marks apps flagged as having some collusion potential according to our approach but not colluding in reality. For instance, the entry in row 1, column 10 means: the program flags collusion of type "information theft" though the set $\{app_1, app_{10}\}$ is clean. However, they are just exchanging information. As stated in our definition of collusion potential, some benign apps can share access to sensitive resources (e.g. a location being shared from a maps app to a social media app). As we consider all channels available in Android as suitable for collusion in our first approach, apps using common channels such as intents with `VIEW` or `SEND` actions (which are very frequently used in Android apps) are also considered to have collusion potential. However, it is unlikely to see apps using these channels for collusion as other apps could have registered to receive the same information. We have taken this fact into consideration when scaling up our methodology (Section 5).

Overall, our approach identifies all colluding app sets but it also flags eight cases with collusion potential where apps are just collaborating.

5. Scaling up

Our initial methodology takes an app set and finds all potentially colluding app sets. This works as long as the app set to analyze is of a reasonable size (i.e., the number of apps that are regularly installed on a regular smart-

phone, about 20 to 30). However, if the number of applications to analyze is larger, scalability becomes a more serious issue.

5.1. Collusion Potential and Computational Complexity

Figure 1 shows an estimate of the maximum number of potentially colluding sets depending on the size of the app set to be analyzed. These estimates correspond to the possible combinations of k apps in a group of n apps.

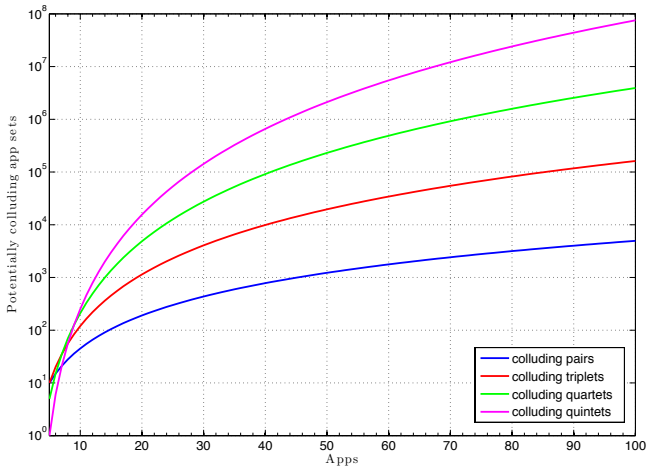

Figure 1: Maximum number of potentially colluding app sets that can be found depending on the size of the number of apps analyzed.

As the number of apps increases, the number of potentially colluding sets rapidly increases. Therefore, if the analyzed apps show a high degree of communication capabilities, the number of potentially colluding app sets will become unmanageably large.

5.2. Managing Complexity

To address scalability, we improved the Prolog-based methodology taking into account the philosophy of Android: it strongly promotes the use of intents and other IPC communications in order to improve user experience. Consequently, many communication paths detected by our method will be benign, flagging app sets that have collusion potential, but are just collaborating. If we were able to identify and remove these benign communications,

then the corresponding potentially colluding sets would be drastically reduced in size.

We analyzed communication signatures generated by more than 50,000 apps included in our experiment dataset (Section 6). Up to 40% of the analyzed apps have the capability to read from and write to external storage. Our approach does not identify specific files accessed in the external storage. Consequently, our crude initial approach would consider all these apps to be capable of communicating with each other. However, this is not the case and fails to represent their real behaviour. We decided to leave out external storage as a communication channel when scaling up our approach. Identification of specific files opened by each app (via tracking their locations and names) is left for future work.

Similarly, we filtered out some common intents used by apps to exchange information. Specifically, we removed the following intent-based communications[4]:

- Intents that can only be generated by the operating system. These can be found in the Android Open Source Project Git page[5]. We have identified 253 intent actions in this category.

- Intent actions that are created by common and trusted third party applications such as Facebook, Dropbox, etc. These are sent by applications that want to interact with these apps, but only the apps from the same developer (Facebook, Dropbox, etc.) receive them. We can detect them by inspecting intents sent and received by the *clean* apps of our data set – c.f. Section 6.1 for details. Intents exhibiting this behaviour will be received by one (e.g. Facebook) or a small number of apps (e.g. apps implementing the Facebook API). To rule out such intents we measured the amount of apps that send the intent divided by the ones that are able to receive it:

  $$c_{intent} = \frac{apps\_sending_{intent}}{apps\_receiving_{intent}}$$

  Any intent action included in the aforementioned apps or with a $C_{intent} \geq 5$ has been included in this list. We have obtained 693 intent actions to filter by using this approach.

[4]The full list of such intents can be found in our github repository.

[5]`https://android.googlesource.com`

- Intents that are used to execute common tasks such as view a document (`android.intent.action.VIEW`); send something (`android.intent.action.SEND`); or open an application (`android.intent.action.MAIN`) are widely used in the Android ecosystem. Some of these are defined in the Android documentation[6]. These kinds of intents are widely sent and received by apps. As they are declared by many apps, in most cases, the user will be asked to select the app to handle the intent, making the collusion attack infeasible. We have identified 208 intent actions matching these characteristics.

6. Experiments

We have used our methodology to look for collusion potential in a set of 50,174 apps provided by McAfee. The goal of this analysis was to shed light on the way Android apps communicate and to test whether our approach can deal with large numbers of apps found in the wild. As our approach focuses on specific, selected communication channels, it might happen that apps not flagged by our approach could be colluding as they use a channel which our analysis does not track.

6.1. Dataset Description

The dataset contains 50,174 Android apps collected from February 2012 up to February 2016. These apps have been categorized by McAfee into three app categories: *malicious*; *potentially unwanted*; and *clean*. Potentially unwanted apps are typically related to excessive advertising, mild privacy invasions and other misbehaviours which cannot be classified as outright malicious. Apps that are known to lack any malicious behaviour are labelled as clean. Table 6 shows a summary of the three groups.

Table 6: Summary of app sets used in our analysis.

	Malware	**Unwanted**	**Clean**
# of apps	13,805	13,991	22,378
# of overall installs	3,696,720	7,656,755	21,205,724,533
Avg size in KB	3,007.9	7,394.52	10,208.3

[6] `http://developer.android.com/reference/android/content/Intent.html`

6.2. Usage of Communication Channels

We first checked if there is a difference between the usage of intents and shared preferences as communication channels. Figure 2 shows the distribution of individual channels found in each of the analyzed app sets (after filtering out common intents). Our first observation is that intent based communication is more predominant in the three analyzed app sets. This is an expected result because intent-based communications are the suggested method for inter-app communication in Android.

6.2.1. Shared Preferences based Channels

We found a significant difference in the amount of individual channels that use shared preferences for the malicious and unwanted app sets. Shared Preferences are not originally intended for application communication. If a developer wants to make a shared preference file accessible outside of the sandbox, this needs to be done explicitly by changing the default flag value (`WORLD_READABLE` or `WORLD_WRITABLE`). Therefore, it is likely that the presence of such channels indicates deliberate information sharing rather than a mistake during app development.

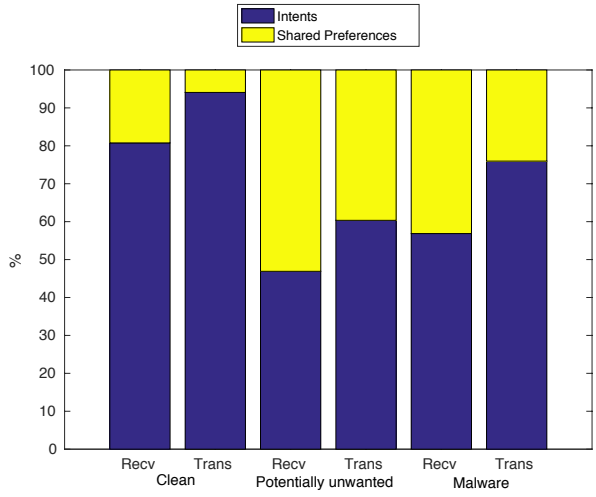

Figure 2: Distribution of unique shared preference and intent based communication channels. *Recv* channels are used to receive information. *Trans* channels used to transmit information.

We have further analyzed how the most common shared preference channels are used. Figure 3 shows the number of apps in each set using each of the top ten identified channels to send or receive information.

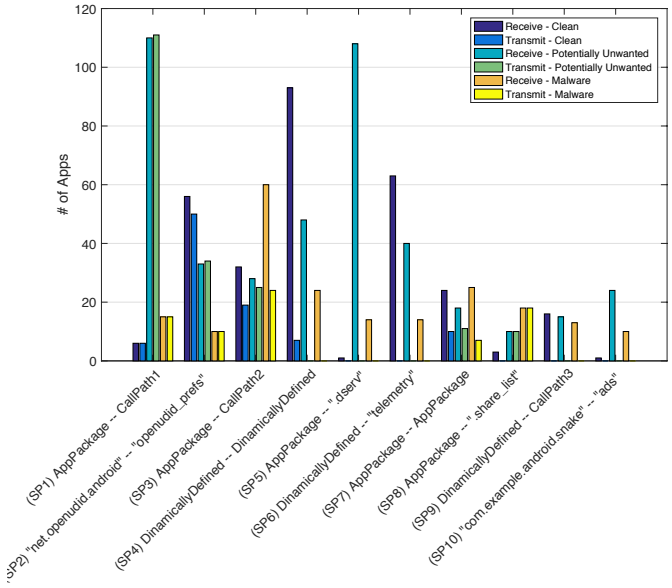

Figure 3: Apps using any of the top 10 shared preference channels. The string before the '-' specifies the app package of the shared preference file. The second string specifies the file name used to store the information. "AppPackage" specifies the application package. "CallGraphX" describes a call graph that is being used by different apps.

Next we manually analyzed samples of the apps employing these communication channels. We found that apps using shared preferences as channels fall into three categories. First, there are some apps that dynamically define the package and preference file they write to or read from ($SP4$). In this case, each app uses the preference file for a specific purpose and it is not possible to extract a behavioural pattern. Second, some apps access preference files that have the same name as their app package ($SP7$). These apps also exhibit different behaviours so it is not possible to extract a pattern from them. The third category consists of the rest of the channels, which are the ones that can be directly mapped to a string value or a call graph. We have found out that all these channels were related to using specific software libraries.

The most used shared preference channels in Figure 3 were traced back

to five different libraries. Channels $SP1$, $SP5$ and $SP8$ are included inside a SDK that belongs to the Chinese company *Play.cn*. While most apps including it are categorized as malicious or potentially harmful, some of them are considered clean. These apps should be further analyzed to determine their behaviour. Channel $SP2$ is created by the *OpenUDID* library. This library, which is now discontinued, was used to generate a unique identifier that could be shared between different apps. This behaviour puts the user's privacy at risk: it can be used by different apps to correlate if they are installed on the same device. $SP3$ and $SP9$ belong to a library developed by the Chinese company *Baidu*. We have been able to identify a colluding behaviour by apps using this SDK, which is described in detail in Section 7. Channel $SP6$ belongs to apps including the Adobe Air SDK. The preference file is read by a method named `getTelemetrySettings`. We did not find any app writing into that file in any of our three app sets. Finally, channel $SP10$ belongs to apps including the *Heyzap* advertising library. Again, no apps in our three datasets were found writing data into that file.

6.2.2. Intent based Channels

Figure 4 shows the number of apps that use the most frequent intents in all the three sets of apps. Analyzing intent communication is more challenging than shared preferences communications. Depending on the app component (activity, service or broadcast receiver) used to match an intent, it is not possible to see by static analysis if it is intended for the same app or a different one. Additionally, as seen with $I3$ and $I4$, we have not detected any receiver that uses intent actions defined programatically. This is because components that receive intents are generally defined statically with strings inside the `AndroidManifest.xml` file.

We found that intent-based communications can also be used to help with app classification. In Figure 4, malicious and unwanted apps use inter-app communication channels that are not being used by clean apps.

As with the shared preferences, we analyzed the origin of the predominant communication channels found in our analysis. Most of them belong to libraries provided by advertisement companies. $I1$ is included in an aggressive ad library known as *apperhand*. $I2$, $I4$, $I8$ and $I9$ belong to the *AirPush* advertisement library. The *Sendroid* ad library includes the communication channel $I3$. Finally, $I5$, $I6$ and $I7$ are included inside the *Startapp* ad library while $I10$ appears in a Chinese library called *umeng*.

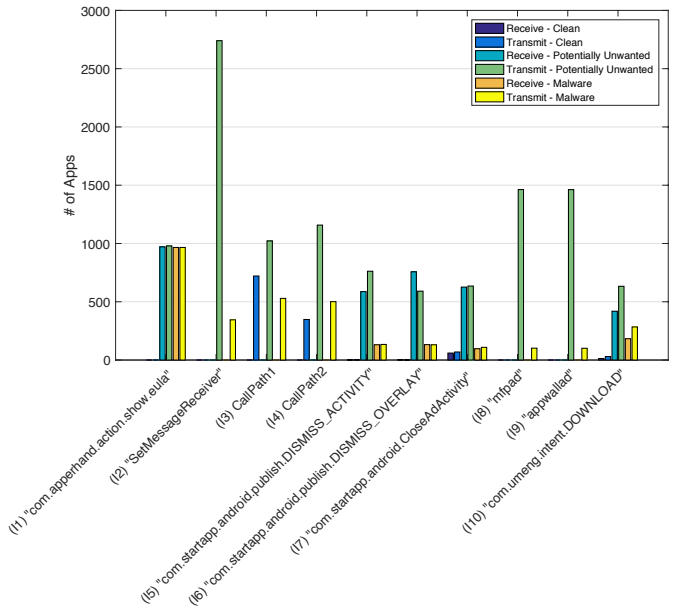

Figure 4: Apps using any of the top ten used intent enabled channels. CallGraphX describes a call graph that is being used by different apps. Quoted values are strings.

6.3. Collusion Potential

Using Prolog to analyze collusion potential provides a great deal of flexibility, by simply modifying the Prolog rules to define collusions we are looking for. We focused on analyzing collusion potential of app sets formed by 2 or 3 apps that may try to extract the accounts, SMS messages or the user contact list. We have limited the size of app sets to two and three for two reasons. First, it is unlikely that an attacker has the resources to make the user to install more than 3 apps. Second, larger app sets will be composed of smaller subsets. Finding them is just a matter of combining smaller colluding app sets. Listing 4 shows the Prolog rules required to identify apps with collusion potential that may affect the accounts, SMS messages or the contacts of the device.

```
coll_accounts(AppA,AppB,Path,Length):- uses(AppA,'GET_ACCOUNTS'),
    comm(AppA,AppB,Length,_,Path), out_comm(AppB).
coll_contacts (AppA,AppB,Path,Length): uses(AppA,'READ_CONTACTS'),
    comm(AppA,AppB,Length,_,Path), out_comm(AppB).
coll_sms(AppA,AppB,Path,Length): uses(AppA,'READ_SMS'), comm(AppA,AppB,Length,_,Path),
    out_comm(AppB).
```

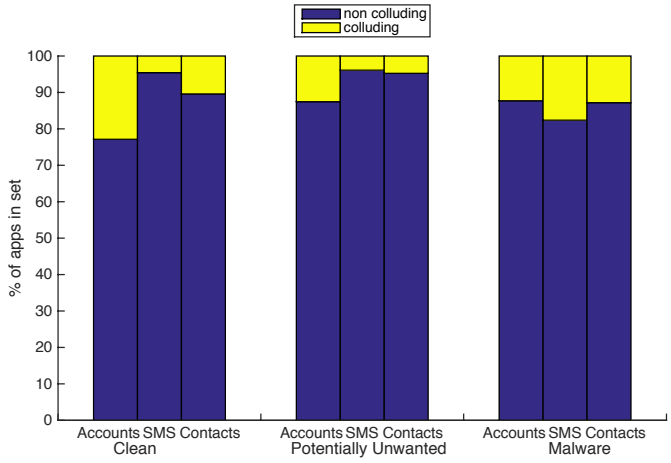

Figure 5: Percentage of applications that access accounts, SMS and contacts with and without collusion potential in each of the datasets.

Listing 4: Selection of collusion potential Prolog rules

Figure 5 shows the percentage of apps inside each set that exhibit collusion potential for any of the analyzed permission-protected resources (accounts, SMS and contacts). Results show that at least 75% of apps in each of the datasets do not exhibit collusion potential regarding the analyzed resources. This greatly reduces the number of possible apps requiring further analysis. Apps inside the malware group exhibit more collusion potential than apps in the other categories (with the exception of apps that access accounts in the clean dataset). This is because malware apps generally request more permissions [21] and the inclusion of advertisement libraries, as we saw in the previous section.

The potentially unwanted app set includes apps with less collusion potential. At first sight this is an unexpected result. However, when analyzing the amount of apps that have colluding potential inside each of the groups, we found an explanation. Figures 6 and 7 show the number of apps that can receive each sensitive protected resource from an app that has been identified to have collusion potential. Although apps inside the unwanted group have a smaller number of apps capable of leaking sensitive information, they are able to share them with a much higher number of apps than apps in the other categories. This is because apps in this group include aggressive

advertisement libraries such as the ones described in the previous section.

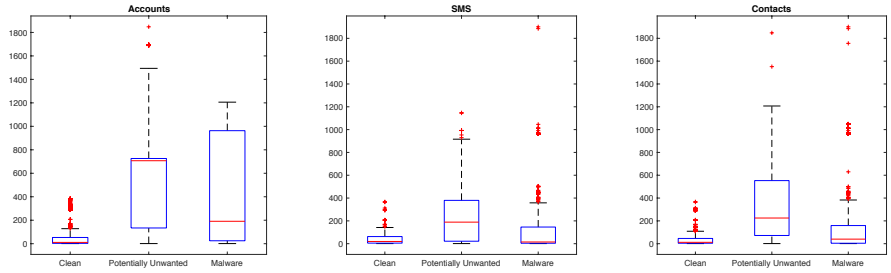

Figure 6: Number of potentially colluding app pairs obtained for each app that exhibited collusion potential in each of the sets for each of the analyzed permission-protected resources.

Apps in the clean set have the smallest number of potentially colluding pairs, while malware apps have a higher number of potentially colluding app pairs regarding accounts (Figure 6). This happens because clean apps request account related permissions more often but communicate with fewer apps, while malware apps require slightly less access to accounts, but communicate with many more apps.

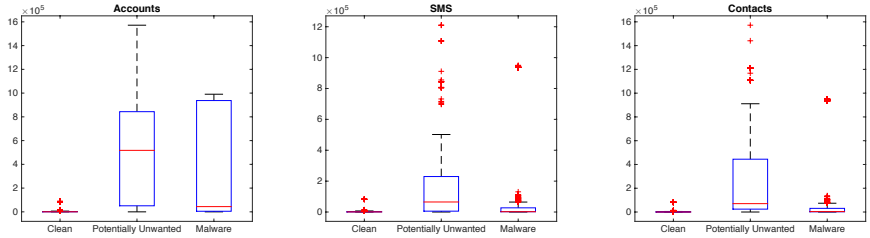

Figure 7: Number of the potentially colluding app triplets obtained for each app that exhibited collusion potential in each of the sets for each of the analyzed permission-protected resources.

This pattern remains when analyzing the number of potentially colluding triplets generated by apps with access to the analysed resources (Figure 7). The main difference is the magnitude on the number of colluding triplets, as the number of possible combinations increases.

6.4. Time Efficiency

The process required to find app sets with collusion potential is split in two phases: extracting the ASR signatures and executing the Prolog program. The first phase needs to be executed only once per app, as signatures can be stored in a database. The second phase is executed every time the fact database is updated (i.e. when a new app is analyzed). It should be noted that our analysis is not bidirectional as we identify the direction of the information flow.

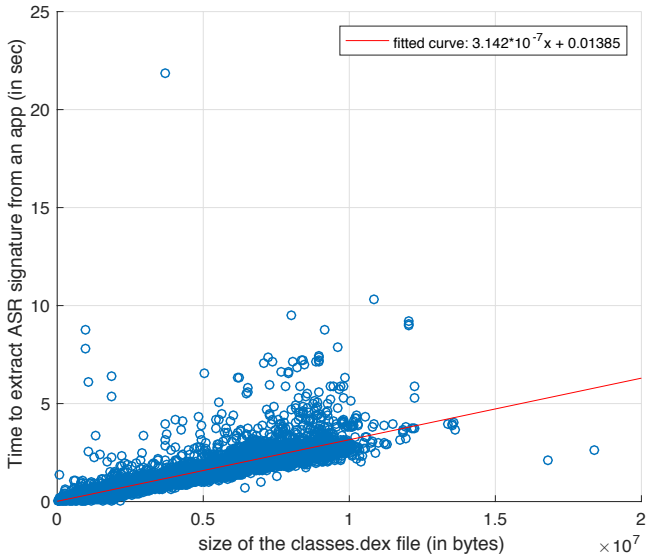

Figure 8: Time required to extract ASR signatures from an app, depending on the size of the *classes.dex* file.

Figure 8 shows the time required to extract the ASR signatures from an app, depending on the size of the *classes.dex* file. All experiments were executed on a commodity PC with an Intel Core i5 2.7 GHz processor and 8GB DDR3 RAM. As expected, time grows with the amount of code to be analyzed. Obtained times fit with a linear function with an R-Square of 76%. For example, a 9.5 Mbyte *dex* file requires around 4 seconds to process. Note that we have not put an emphasis on optimizing the ASR extraction code.

Figure 9 plots the time required to list all apps colluding with a specific app. This time depends on the number of colluding sets found for the queried app. Queries for apps that do not exhibit any colluding behaviour take 30 ms

on average. When looking for potentially colluding app pairs, the maximum query time obtained during our experiments was 216 ms. The higher times shown in Figure 9 were obtained when looking for colluding app triplets. Obtained times fit with a polynomial of grade 2 with a R-square of 71%. Analysis time could be reduced by stopping queries at the first match; this way, only apps with a match would be analyzed further.

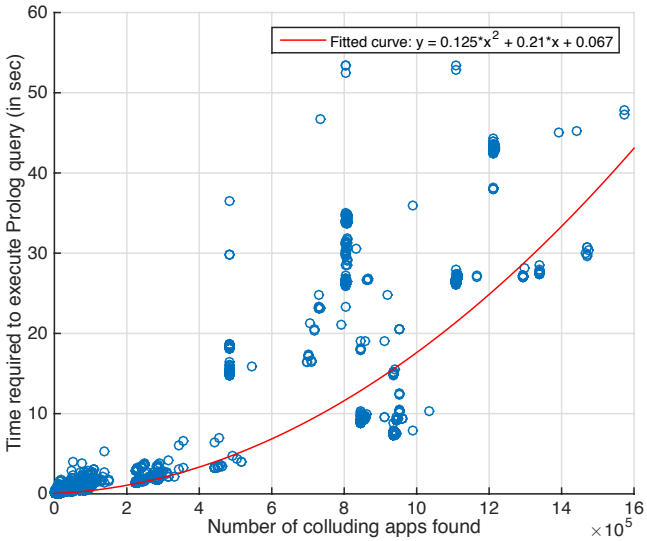

Figure 9: Time required to execute a Prolog query depending on the amount of potentially colluding app sets the app belongs to.

7. Colluding Behaviour of MoPlus SDK

During our experiments querying for potentially colluding app pairs, we identified a group of apps that was communicating using both intents and shared preference files. A manual review of the flagged apps revealed that they were sharing information through shared preferences files to synchronize execution of a potentially harmful payload. This payload was embedded into all the apps through a library, the MoPlus SDK. MoPlus is included in more than 5,000 Android installation packages (APKs). This library has been known to be malicious since November 2015 [22]. However, the collusion behaviour of the SDK was unknown. In the rest of this section, we briefly describe the malicious behaviour of the SDK and provide a more detailed

analysis of its colluding behaviour. To the best of our knowledge, this is the first instance of collusion found in the wild.

7.1. Malicious Payload

The MoPlus SDK has the ability to open a local HTTP server on the user device. This enables the attacker to perform a series of malicious operations including:

- Send arbitrary intents received via the command and control (C&C) server.
- Obtain sensitive information from the users device, including the user location and the IMEI (International Mobile Equipment Identity).
- Install apps silently in rooted devices.
- Add contacts received from the C&C server.

The malicious payload embedded inside the MoPlus SDK inherits all permissions requested by the app. As these are chosen by the app developer, which may differ from the SDK developer, it is possible that an app including the SDK does not have the necessary permissions to execute all the library's malicious payload. The colluding behaviour of the MoPlus SDK aims to avoid this problem by identifying which of the apps that include the MoPlus SDK and are installed on a device has the most comprehensive access to system resources.

7.2. Colluding Behaviour

The detected colluding behaviour differs from the standard colluding behaviour studied in most app collusion research [12, 18]. In a nutshell, all apps including the MoPlus SDK that are running on a device will talk to each other to check which of the apps has the most privileges. This app will then be chosen to execute the local HTTP server able to receive commands from the external C&C server, maximizing the effects of the malicious payload.

The MoPlus SDK includes the `MoPlusService` and the `MoPlusReceiver` components. In all analyzed apps, the service is exported. In Android, this is considered to be a dangerous practice, as other apps will be able to call and access this service. However, in this case it is a feature used by the SDK to enable communication between its apps.

The colluding behaviour is executed when the MoPlusService is created (`onCreate` method). This behaviour is triggered by the MoPlus SDK of each app and can be divided in two phases: establishing app priority and executing the malicious payload. In the next sections, we will describe this behaviour in detail with reconstructed code samples. These have been obtained by disassembling part of the code from the Baidu Searchbox app (MD5=062f91b3b1c900e2bc710166e6510654). Locations of different payloads may differ from app to app, as code is generally obfuscated by using Proguard.

7.2.1. Establishing app priority

During SDK initialization, the `MoPlusService` is created inside each app with the MoPlus SDK. The service executes three checks (Listing 5):

1. The version of the MoPlus SDK is checked against a value stored in a preference file (lines 3 to 5).
2. The SDK looks for the tag `DisableService` inside the AndroidManifest (`!a(Context)`, line 8). If it is found, it will not continue to execute.
3. The SDK checks if the app executing the SDK has all the necessary components of the SDK and the minimum permissions required by the SDK have been granted (`j(Context)`, line 8). The minimum permissions required to continue execution are: `INTERNET`, `READ_PHONE_STATE`, `ACCESS_NETWORK_STATE`, `BROADCAST_STICKY`, `WRITE_SETTINGS`, `WRITE_EXTERNAL_STORAGE`, `SET_ACTIVITY_WATCHER`, `GET_TASKS`.

```
1  SharedPreferences localSharedPreferences = paramContext.getSharedPreferences("pst", 0);
2  int i = c(paramContext, paramContext.getPackageName());
3  int j = localSharedPreferences.getInt("pr_v", 0);
4  SharedPreferences.Editor localEditor1;
5  if ((j < i) || (paramBoolean)){
6     Log.d("Utility", "oldVCode=" + j + " vcode=" + i + " isForce " + paramBoolean);
7     localEditor1 = paramContext.getSharedPreferences(paramContext.getPackageName() +
            ".push_sync", 1).edit();
8     if ((!a(paramContext)) && (j(paramContext)))
9        break label197;
10    localEditor1.putLong("priority", 0L);
11 }
```

Listing 5: Code used to check for execution conditions. This code is included in the class com.baidu.android.moplus.util.a.

If any of these checks fail, the service assigns itself a zero priority inside a preference file readable by the rest of the apps installed in the system

(line 10). The name of the preference file is created adding the extension `.push_sync` to the app package name. The SDK uses the `WORLD_READABLE` flag to save the file so other apps can access it.

If the three checks hold, the service executes the method `f(Context)`. This method computes a priority to the app that depends on several factors (Listing 6).

```
1  public static long f(Context paramContext){
2      long l1 = 0L;
3      if (paramContext == null)
4          return l1;
5      if (!g(paramContext, paramContext.getPackageName()))
6          l1 += 1L;
7      long l2 = l1 << 1;
8      if (!i(paramContext))
9          l2 += 1L;
10     long l3 = l2 << 1;
11     if (!f(paramContext, paramContext.getPackageName()))
12         l3 += 1L;
13     long l4 = l3 << 1;
14     if (d(paramContext, paramContext.getPackageName()))
15         l4 += 1L;
16     long l5 = l4 << 1;
17     if (p(paramContext))
18         l5 += 1L;
19     long l6 = l5 << 1;
20     if (b(paramContext, paramContext.getPackageName()))
21         l6 += 1L;
22     return 0x79000000000000 | (l6 | 0xFF & i(paramContext, "moplus_addon_priority") << 40);
23 }
```

Listing 6: Code used by MoPlus SDK to assign priority execution to each app MoPlusService. This code is included in the class com.baidu.android.moplus.util.a.

These include, from lowest to highest priority:

1. Several meta-data values from the manifest (lines 3 to 15): `DisableLocalServer`, `DisableStatistic`, `DisableApplist`, `isBaiduApp`.
2. Access to the contact lists (lines 17 and 18).
3. If the app is part of the system image (l. 20 and 21).
4. A priority value included in the manifest (line 22).

The obtained priority is saved in the preference file with `push_sync` extension. This behaviour is executed by all apps including the MoPlus SDK (Figure 10).

7.2.2. Executing the malicious payload

After the priority has been obtained and stored, the service method `OnCreate()` calls the method `a(Context,long)` (Listing 7) to create and broad-

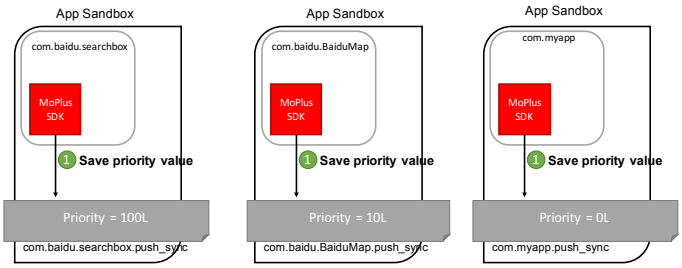

Figure 10: Phase 1 of the colluding behaviour execution. Each app saves a priority value that depends on the degree of access it has to the system resources. Shown priority values are examples.

cast a new intent object.

```
1  public static void a(Context paramContext, long paramLong){
2      Context localContext = paramContext.getApplicationContext();
3      Intent localIntent = c(localContext);
4      localIntent.setPackage(d(localContext));
5      a(localContext, localIntent, paramLong);
6  }
```

Listing 7: Creation of a new intent object. The method `a` is used to broadcast it. This code is included in the class com.baidu.android.moplus.util.a.

The `localIntent` value is obtained from the execution of the method `c(Context)` (line 3). This method creates the intent that will start the `MoPlusReceiver` (Listing 8).

```
1   public static Intent c(Context paramContext){
2     Intent localIntent = new Intent("com.baidu.android.moplus.action.START");
3     localIntent.addFlags(32);
4     localIntent.putExtra("method_version", "V1");
5     return localIntent;
6   }
```

Listing 8: Creation of a new intent object. The method `a` is used to broadcast it. This code is included in the class com.baidu.android.moplus.util.a.

The call to `d(Context)` (line 4) looks for the app package with highest priority through the method `a(Context,String,String)` (Listing 9).

```
1  public static String d(Context paramContext){
2      return a(paramContext, ".push_sync", "priority");
3  }
```

Listing 9: Method that returns the app package with highest priority. This code is included in the class com.baidu.android.moplus.util.a.

The method `a(Context,String,String)` looks for all the packages that are able to answer the Intents included in the MoPlus SDK (Listing 10, lines 3 to 6). These include both `com.baidu.android.moplus.action.START` and `com.baidu.android.pushservice.action.BIND_SYNC`.

```
1  public static String a(Context paramContext, String paramString1, String paramString2){
2    List localList = h(paramContext);
3    if ((localList == null) || (localList.size() <= 1)){
4      localObject1 = paramContext.getPackageName();
5      return localObject1;
6    }
7    long l1 = paramContext.getSharedPreferences(paramContext.getPackageName() + ".push_sync",
         1).getLong("priority", 0L);
8    String str = paramContext.getPackageName();
9    Iterator localIterator = localList.iterator();
10   while (localIterator.hasNext()){
11     localObject2 = ((ResolveInfo)localIterator.next()).activityInfo.packageName;
12       SharedPreferences localSharedPreferences2 =
             paramContext.createPackageContext((String)localObject2,
             2).getSharedPreferences((String)localObject2 + paramString1, 1);
13     ...
14   }
15 }
```

Listing 10: Method that inspects all shared preference files of packages that answer the MoPlus SDK actions. This code is included in the class com.baidu.android.moplus.util.a.

For each package found, it inspects the contents of the `push_sync` file to get its priority, returning the package name of the one with highest priority (Listing 10, line 10 to end). The intent to be launched is configured so that only receivers listed in the returned package can receive it (Listing 7, line 4). Finally, the method a(Context, Intent, long) (Listing 11) cancels previous intents being registered (to avoid launching the service more than once) and sends the intent after a delay passed as a parameter.

```
1  public static void a(Context paramContext, Intent paramIntent, long paramLong){
2    PendingIntent localPendingIntent = PendingIntent.getBroadcast(paramContext, 0, paramIntent,
         268435456);
3    AlarmManager localAlarmManager = (AlarmManager)paramContext.getSystemService("alarm");
4    localAlarmManager.cancel(localPendingIntent);
5    localAlarmManager.set(3, paramLong + SystemClock.elapsedRealtime(), localPendingIntent);
6  }
```

Listing 11: Method that cancels previous intents matching the service, and registers a new intent to be launched.

The described behaviour is executed by all apps that include the MoPlus SDK libraries (Figure 11).

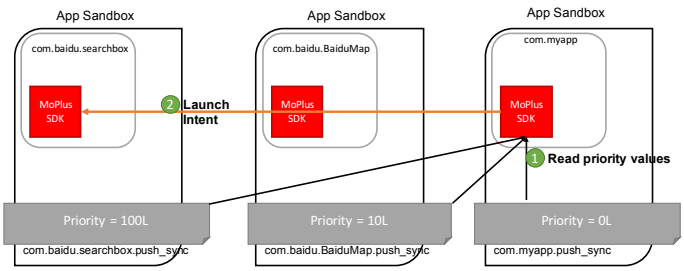

Figure 11: Phase 2 of the colluding behaviour execution. Each app checks the `WORLD_READABLE` `SharedPreference` files and sends and intent to the app with highest priority

8. Related Work

Android malware detection has been an attractive and active research area during the last few years. As a result, techniques for detecting Android malware are readily available [23, 24, 25]. These can be categorised into static analysis, dynamic analysis or a combination of both. In static analysis, certain features of the app binary are extracted and analysed using different approaches such as machine learning techniques. Examples of these are [26], using hardware components, requested permissions, critical and suspicious API calls, and network addresses. Conversely, dynamic analysis detects malware at run-time. It employs suitable app monitoring to track actions and features that indicate malicious behaviours. An example of this is [25], which first uses static analysis to guide the dynamic analysis while tracking network traffic and API calls among others.

Detecting colluding apps involves not only obtaining features that appear if an app carries out a malicious action, but also revealing whether communication between apps occurs during those actions. Previously described malware detection techniques focus only on detecting whether a single app is malicious, but do not analyse their communication channels. This approach limits their usage for collusion detection. Approaches based on taint analysis like Amandroid [27] and FlowDroid [28] could be extended for collusion detection. These are focused on analyzing single apps to detect information leaks through inter-component communications, ICC, (i.e. a location leaking from a service to an activity within the same app). This limits its usefulness for detecting collusions. First, they are only able to analyse single apps. This means that, although they are able to detect leaks to other apps

through inter-app communications[7], (i.e. and activity/service from one app sending information to an activity/service from another app), they are not able to tell which other app is taking part in the collusion. In addition to this, colluding apps may use other communication channels for collusion (i.e. covert channels) rather than standard IAC channels.

To overcome this limitation, there are approaches like APKCombiner [29] which joins two applications into a single APK. This way, a security analyst can use information flow tools to analyse the inter-app communication mechanisms. Their evaluation over a set of 3000 apps shows that the approach is valid, as it is capable of joining together 88% of the possible app pairs. The average time required to join two apps is three minutes. This makes it hard to use for practical large-scale app analysis.

Countermeasures specifically designed to mitigate collusion attacks currently employ either static analysis or they utilize Android OS extensions. ComDroid [30] is a static analysis tool that looks for confused deputies through *Intents*. $\mathbb{K}$droid [31] detects collusion via software model-checking a set of Android apps utilising the $\mathbb{K}$ framework. PermissionFlow uses taint analysis to automatically detect inter-application permission leaks [32]. In their work they found that more than 50% of the top 313 apps (in 2012) actively used inter-component information flows and four of them leaked permissions to other apps. Our work differs from PermissionFlow in our lack of taint analysis and our consideration of channels that may be used specifically for collusion (e.g. shared preferences). Taint analysis allows PermissionFlow to be more precise, but at the same time it is more computationally costly. Our system could be used to filter out app sets without colluding potentially, focusing the more computationally complex analysis on those that exhibit collusion potential.

In contrast to these, XManDroid [13], TrustDroid [33] and [34] extend the Android OS by providing finer control over app communications. These extensions identify possible communication paths between apps and allow to define policies that control how they exchange information. These are similar to the ones provided by the *Intent Firewall* included as a component, not enabled, in recent Android versions [35]. TrustDroid provides additional controls to monitor the file system and network connections. However, none of them provides a monitoring system for shared preference files or covert chan-

[7]In Android, IAC and ICC are implemented through very similar APIs.

nels. As we have found during our research, these communication channels are also a viable means of communication between colluding apps.

9. Conclusions

Detecting app collusion on a large scale is a challenging task due to the sheer amount of possible app combinations and communication channels. We have presented a method to analyze large sets of apps to look for collusion potential. Our method is based on a lightweight analysis of apps that extracts *ASR* signatures. These are transformed into Prolog facts, so logic programming can be used to identify collusion potential between apps in an efficient way.

We have validated our approach against an artificial set of apps and tested it against a large dataset of "in the wild" apps. Our results show that malicious apps use inter-app communications in a different way than clean ones. Malware classification methods could take advantage of this fact to increase their accuracy and to detect collusion.

A manual analysis of some of the apps flagged by our detection system allowed us to identify the first known case of collusion in the wild. This discovery demonstrated the risk of using untrusted or maliciously modified SDKs. The designers of these app communication scheme even considered the possibility of the SDK being included as part of a system image. Identified colluding apps synchronize with each other (by sharing a priority value) activating only the service within the more privileged app. This finding demonstrates the need to cover more communication methods when looking for colluding apps. Although intent-based communications are more common, other means of communication (such as the shared preferences and covert channels) should also be considered. Our future direction of work aims to explore these communication channels and to use formal verification methods to automatically analyze apps flagged by our system [36].

Acknowledgments

This work has been supported by UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/L022699/1.

References

[1] I. Lunden, "6.1B Smartphone Users Globally By 2020, Overtaking Basic Fixed Phone Subscriptions," http://techcrunch.com/2015/06/02/6-1b-smartphone-users-globally-by-2020-overtaking-basic-fixed-phone-subscriptions/#.pkatr9:RPIH, accessed: 10-11-2015.

[2] R. Lipovsky, "Eset analyzes first android file-encrypting, tor-enabled ransomware," http://www.welivesecurity.com/2014/06/04/simplocker/, 2014.

[3] E. K. o, "Malware abuses android accessibility feature to steal data," http://www.securityweek.com/malware-abuses-android-accessibility-feature-steal-data, 2015.

[4] C. Page, "Mkero: Android malware secretly subscribes victims to premium sms services," http://www.theinquirer.net/inquirer/news/2425201/mkero-android-malware-secretly-subscribes-victims-to-premium-sms-services, 2015.

[5] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 2012, p. 3.

[6]

[7] R. Unuchek and V. Chebyshev, "Mobile malware evolution 2015," *AO Kapersky Lab*, 2014.

[8] J. Blasco, I. Muttik, M. Roggenbach, and T. M. Chen, "Wild android collusions," in *VirusBulletin 2016*. Virus Bulletin, 2016.

[9] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE security & privacy*, no. 1, pp. 50–57, 2009.

[10] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing android-powered mobile devices using selinux," *IEEE Security & Privacy*, no. 3, pp. 36–44, 2009.

[11] S. Mutti, E. Bacis, and S. Paraboschi, "An selinux-based intent manager for android," in *Communications and Network Security (CNS), 2015 IEEE Conference on*. IEEE, 2015, pp. 747–748.

[12] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones." in *NDSS*, vol. 11, 2011, pp. 17–33.

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

[14] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 961–987, Second 2014.

[15] M. La Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *Communications Surveys & Tutorials, IEEE*, vol. 15, no. 1, pp. 446–471, 2013.

[16] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *USENIX Security Symposium*, 2011.

[17] Z. Fang, W. Han, and Y. Li, "Permission based android security: Issues and countermeasures," *computers & security*, vol. 43, pp. 205–218, 2014.

[18] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 51–60.

[19] A. Desnos, "Androguard," https://github.com/androguard/androguard, 2015, accessed: 15-10-2015.

[20] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[21] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[22] S. Shen, "Setting the record straight on moplus sdk and the wormhole vulnerability," http://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/, accessed: 04/0/2016.

[23] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in *NDSS*, 2015.

[24] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," in *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*. IEEE, 2014, pp. 3–17.

[25] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1808–1815.

[26] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: http://www.internetsociety.org/doc/drebin-effective-and-explainable-detection-android-malware-your-pocket

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

[28] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *ACM SIGPLAN Notices - PLDI'14*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[29] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Apkcombiner: Combining multiple android apps to support inter-app analysis,"

in *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 513–527.

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

[31] I. M. Asavoae, H. N. Nguyen, M. Roggenbach, and S. Shaikh, "Utilising $\mathbb{K}$ semantics for collusion detection in android applications," in *FMICS/AVoCS'16*. Springer, 2016.

[32] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, "Automatic detection of inter-application permission leaks in android applications," *IBM Journal of Research and Development*, vol. 57, no. 6, pp. 10–1, 2013.

[33] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 51–62.

[34] Y. Jing, G.-J. Ahn, A. Doupé, and J. H. Yi, "Checking intent-based communication in android with intent space analysis," in *Proceedings of The 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[35] C. Yagemann, "Intent firewall," Web, July 2016.

[36] I. M. Asavoae, H. N. Nguyen, M. Roggenbach, and S. A. Shaikh, "Software model checking: A promising approach to verify mobile app security: A position paper," in *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017*. ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3103111.3104040