Detecting malicious collusions between mobile software applications – the Android case

Irina Măriuca Asăvoae, Jorge Blasco, Thomas M. Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach and Siraj Ahmed Shaikh

Abstract Malware has been a major problem in desktop computing for decades. With the recent trend towards mobile computing, malware is moving rapidly to mobile platforms. "Total mobile malware has grown 151% over the past year", according to McAfee's quarterly threat report from September 2016. By design, Android is 'open' to download apps from different sources. Its security depends on restricting apps by combining digital signatures, sand-boxing, and permissions. Unfortunately, these restrictions can be bypassed, without the user noticing, by colluding apps for which combined permissions allow them to carry out attacks. In this chapter we report on recent and ongoing research results from our ACID project which suggest a number of reliable means to detect collusion, tackling the aforementioned problems. In the chapter we present our conceptual work on the topic of collusion and discuss a number of automated tools arising from it.

1 Introduction

One of the most fundamental principles in computer security is ensuring effective isolation of software. Programs written by different independent software vendors (ISVs) have to be properly separated to avoid any accidental data flows as well as all deliberate data leaks. Strictly speaking, even subroutines of a program need to be properly isolated and some computer systems attempt that too via, for example, protection of stack frames and memory tagging. This isolation principle helps ensure

Irina Măriuca Asăvoae - Inria Paris, France, e-mail: irina-mariuca.asavoae@inria.fr • Jorge Blasco - Royal Holloway University of London, UK, e-mail: jorge.blascoalis@rhul.ac.uk • Thomas M. Chen - City University of London, UK, e-mail: tom.chen.1@city.ac.uk • Harsha K. Kalutarage - Queen's University of Belfast, UK, e-mail: h.kalutarage@qub.ac.uk • Igor Muttik - Cyber Curio LLP e-mail: igor.muttik@cybercurio.com • Hoang Nga Nguyen and Siraj Ahmed Shaikh - Coventry University, UK, e-mail: \{hoang.nguyen,siraj.shaikh\}@coventry.ac.uk • Markus Roggenbach - Swansea University, UK, e-mail: m.roggenbach@swansea.ac.uk

both reliability of software (by limiting the effect of design flaws, insecure design, bugs, etc.) as well as protect from outright malicious code (malicious data-leaking libraries, exploiting vulnerabilities via injecting shell-code, etc.)

The era of personal computing slightly diminished the requirement for isolation. It was believed that PCs – being single-user devices – will be OK with all software running at the same privilege. First personal computers had no hardware and software support for software isolation. However, reliability and privacy demanded a better solution so these primitive OSes were replaced by multiple descendants of Windows NT, Unix and Linux. Requirements for better isolation also drove special hardware features examples include Intel SGX enclaves and ARM TrustZone.

In the cloud environments (like Docker, Amazon EC2, Microsoft Azure, etc.) which execute software from different sources and operate on data belonging to different entities, guaranteed isolation becomes even more important because any cross-container data leaks (deliberate or accidental) may be devastating. Communications across cloud containers have to be covert because no explicit APIs is provided. The problem of covert communication between programs running in time-sharing computer systems was first discussed as early as in 1970s [36].

The situation is quite different in mass-market operating systems for mobile devices such as smart phones - there is no need for covert channels at all. While corresponding operating systems (Symbian, MeeGo, iOS, Android, Tizen, etc.) were designed with the isolation principle in mind, the requirement for openness led to the ability of software in the device to communicate in many different ways. Android OS is a perfect example of such a hybrid design - apps run in sandboxes but they have documented means of sending and receiving messages to/from each other; they can also create shared objects and files. These inter-app communication mechanisms are handy but, unfortunately, also make it possible to carry out harmful actions in a collaborative fashion.

Extreme commonality of Android as well as rapid growth of malicious and privacy-leaking apps made it a perfect target for our team to look for colluding behaviours. Authors of malicious software would be interested in flying under the radar for as long as possible. Unscrupulous advertisers could also benefit from hiding privacy-invading functionality in multiple apps. These reasons led us to believe that Android may be one of the first targets for collusion attacks. We also realised that security practitioners who analyse threats for Android desperately need tools which would help them uncover colluding apps. Such apps may be outright malicious or they may be unwanted programs which often do aggressive advertising coupled with disregard for users' privacy (like those which would use users' contacts to expand their advertising further). Having a popular OS which allowed (and to some extent even provides support to) colluding apps was a major risk.

Before we started there were no tools or established methods to uncover these attacks: discovering such behaviours is very tricky - two or more mobile apps, when analysed independently, may not appear to be malicious. However, together they could become harmful by exchanging information with one another. Multi-app threats such as these were considered theoretical for some years, but as part of this research we discovered colluding code embedded in many Android apps in the wild

[48]. Our goal was to find effective methods of detecting colluding apps in Android [6, 7, 8, **?**, 12, 13, 37]. This would potentially pave a way for spotting collusions in many other environments that implement software sandboxing, from other mobile operating systems to virtual machines in server farms.

1.1 Background

Malware has been a major problem in desktop computing for decades. With the recent trend towards mobile computing, malware is moving rapidly to mobile platforms. Total mobile malware has grown 151% over the past year, according to McAfees quarterly threat report from September 2016. Criminals are clearly motivated by the opportunity - the number of smartphones in use is predicted to grow from 2.6 billion in 2016 to 6.1 billion in 2020, predominantly Android, with more than 10 billion apps downloaded to date. Smartphones pose a particular security risk because they hold personal details (accounts, locations, contacts, photos) and have potential capabilities for eavesdropping (with cameras/microphone, wireless connections).

By design, Android is "open" to download apps from different sources. Its security depends on restricting apps by combining digital signatures, sandboxing, and permissions. Unfortunately, these restrictions can be bypassed, without the user noticing, by colluding apps for which combined permissions allow them to carry out attacks.

A basic example of collusion consists of one app permitted to access personal data, which passes the data to a second app allowed to transmit data over the network. While collusion is not a widespread threat today, it opens an avenue to circumvent Android permission restrictions that could be easily exploited by criminals to become a serious threat in the near future.

Almost all current research efforts are focusing on detection of single malicious apps. The threat of colluding apps is challenging to detect because of the myriad and possibly stealthy ways in which apps might communicate with each other. Existing Anti-Virus (AV) products are not designed to detect collusion. A review of the literature shows that detecting application collusion introduces a new set of challenges including: the detection of communication channels between apps, the exponential number of app combinations, and the difficulty of actually proving that two or more apps are really colluding.

1.2 Contribution

In this chapter we report on recent and ongoing research results from our ACID project which suggest a number of reliable means to detect collusion, tackling the

aforementioned problems. In the chapter we present our conceptual work on the topic of collusion and discuss a number of automated tools arising from it.

We start with an overview on the Android Operating System, which introduced the various security mechanism built in.

Then we give a definition for app collusion, and distinguish collusion from the closely related phenomena of collaboration and confused deputy attacks.

Based on this we address the exponential complexity of the problem by introducing a filtering phase. We develop two methods based on a lightweight analysis to detect if a set of apps has any collusion potential. These methods extract features through static analysis and use first order logic and machine learning to assess whether an analysed app set has collusion potential. By developing two methods to detect collusion potential we address the problem of collusion with two distinct approaches.

The first order logic approach allows us to define collusion potential through experts, which may identify attack vectors that are not yet being seen in the real world. Whereas the machine learning approach uses Android permissions to systematically assign the degree of collusion potential a set of apps may pose. A mix of techniques as such provides for an insightful understanding of possibly colluding behaviours and also adds confidence into filtering.

Once we have reduced the search space, we use a more computational intensive approach, namely software model checking, to validate the actual existence of collusion between the analysed apps. To this end, model checking provides dynamic information on possible app executions that lead to collusion; counter examples (witness traces) are generated in such cases.

In order to evaluate our approaches, we have developed a set of specifically crafted apps and gathered a data set of more than 50,000 real-world apps. Some of our approaches have been validated by using the crafted app set and tested against the real-world apps.

The above effort has been demonstrated through a number of publications through the different teams involved focusing on different aspects. The chapter allows us to provide a consolidated perspective on the problem. By systematically taking related work into account, we aim to provide a comprehensive presentation of state of the art in collusion analysis.

This chapter reports previously published work [13, 6, 7, 8, 11, 37], however, we expand on these publications by providing more detail and also the singular approach into context.

2 The Android Operating System

The Android operating system consists of three software layers above a Linux kernel as shown in Figure 1. The Linux kernel is slightly modified for an embedded environment. It runs device-specific hardware drivers, and manages power and the file system. Android is agnostic of the processor (ARM, x86, and MIPS) but does

take advantage of some hardware-specific security capabilities, e.g., the ARM v6 eXecute-Never feature for designating a non-executable memory area.

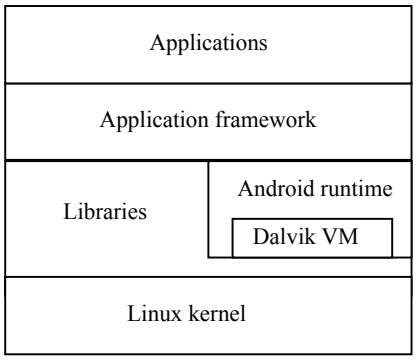

Fig. 1: Android operating system layers

Above the kernel, libraries of native machine code provide common services to apps. Examples include Surface Manager for graphics; WebKit for browser rendering; and SQLite for basic datastore. In the same layer, each app runs in its own instance of Android runtime (ART) except for some apps that are native, e.g., core Android services. A replacement for the Dalvik virtual machine (VM) since Android 4.4, the ART is designed to run Dalvik executable (DEX) bytecode on resource-constrained mobile devices. It introduces ahead-of-time (AOT) compilation converting bytecode to native code at installation time (in contrast to the Dalvik VM which interpreted code at runtime).

The application framework above the libraries offers packages and classes to provide common services to apps, for examples: the Activity Manager for starting activities; the Package Manager for installing apps and maintaining information about them; and the Notification Manager to give notifications about certain events to interested apps.

The highest application layer consists of user-installed or pre-installed apps. Java source code is compiled into JAR (Java archive) files composed of multiple Java class files, associated metadata and resources, and optional manifest. JAR files can be translated into DEX bytecode and zipped into Android package (APK) files for distribution and installation. APK files contain .dex files, resources, assets, lib folder of processor-specific code, META-INF folder containing manifest MANIFEST.MF and other files, and additional AndroidManifest.xml file. The AndroidManifest.xml file contains the necessary configuration information to install the app, notably defining permissions to request from the user.

2.1 App Components

Android apps are built composed of one or more components which must be declared in the manifest.

- *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, such as a "main" activity launched when a user starts an app.
- *Services* operate in the background to carry out long-running tasks for other apps, such as listening to incoming connections or downloading a file.
- *Broadcast receivers* respond to messages that are sent through Intent objects, by the same or other apps.
- *Content providers* manage data shared across apps. Apps with content providers enable other apps to read and write their local data.

Any component can be public or private. If a component is public, components of other apps can interact with it, e.g., start the Activity, start the Service. If a component is private, only components from the app that runs with the same user ID (UID) can interact with that component.

2.2 Communications

Android allows any app to start another app's component in order to avoid duplicate coding for the same function. However, this can not be done directly because apps are separate processes. To activate a component in another app, an app must deliver a message to the system that specifies the intent to start that component.

Intents are message objects that contain information about the operation to be performed and relevant data. Intents are delivered by various methods to application components, depending on the type of component. Intents about certain events are broadcasted, e.g., an incoming phone call. Intents can be explicit for specific recipients or implicit, i.e., broadcast through the system to any components listening. Components can provide Intent filters to specify which Intents a component is willing to handle.

Besides Intents, processes can communicate by standard Unix communication methods (files, sockets), Android offers three inter-process communication (IPC) mechanisms:

- *Binder*: a remote procedure call mechanism implemented as a custom Linux driver;
- *Services*: interfaces directly accessible using Binder;
- *Content Provider*: provide access to data on the device.

2.3 App Distribution and Installation

Android apps can be downloaded from the official Google Play market or many third party app stores. To catch malicious apps from being distributed, Google uses a variety of services including Bouncer, Verify Apps, and Safety Net. Since 2012, the Bouncer service automatically scans the Google Play market for potentially malicious apps (known malware) and apps with suspicious behaviours. It does not examine apps installed on devices or apps in third party app stores. Currently, however, none of these services look for apps exhibiting collusion behaviours.

The Verify Apps service scans apps upon installation on an Android device and scans the device in the background periodically or when triggered by potentially harmful behaviours, e.g., root access. It warns users of potentially harmful apps (PHAs) which may be submitted online for analysis.

The Safety Net service looks for network-based security threats, e.g., SMS abuse, by analyzing hundreds of millions of network connections daily. Google has the option to remotely remove malicious apps.

2.4 Android Security Approach

Android security aims to protect user data and system resources (including the network), which are regarded as the valuable assets. Apps are assumed to be untrusted by default and therefore considered potential threats to the system and other apps. The primary method of protection is isolation of apps from other apps, users from other users, and apps from certain resources. IPC is possible but mediated by Binder. However, the user is ultimately in control of security by choosing which permissions to grant to apps. For more detailed information about Android security, the reader is referred to the extensive literature [27, 21, 22, 32, 59].

Android security is built on key security features provided by the Linux kernel, namely: a user-based permissions system; process isolation; and secure IPC. In Linux, each Linus user is assigned a user (UID) and group ID (GID). Access to each resource is controlled by three sets of permissions: owner (UID), group (GID), and world. The kernel isolates processes such that users can not read another user's files or exhaust another's memory or CPU resources. Android builds on these mechanisms. An Android app runs under a unique UID, and all resources for that app are assigned full permissions for that UID and no permissions otherwise. Apps can not access data or memory of other apps by default. A user with root UID can bypass any permissions on any file, but only the kernel and a small subset of core apps run with root permission.

By default, apps are treated as untrusted and can not interact with each other and have limited access to the operating system. All code above the Linux kernel (including libraries, application framework, and app runtime) is run within a sandbox to prevent harm to other apps or the system.

Apps must be digitally signed by their creators although their certificate can be self signed. A digital signature does not imply that an app is safe, only that the app has not been changed since creation and the app creator can be identified and held accountable for the behaviour of their app.

A permissions system controls how apps can access personal information, sensitive input devices, and device metadata. By default, apps collecting personal information restricts data access to themselves. Access to sensitive user data is available only through protected APIs. Other types of protected APIs include: cost sensitive APIs that might generate a cost to the user; APIs to access sensitive data input devices such as camera and microphone; and APIs to access device metadata. App permissions are extracted from the manifest at install time by the PackageManager.

The default set of Android permissions is grouped into four categories as shown in Table 1.

Table 1: The default set of Android permissions

Category	Description	Examples
Normal	Can not cause real harm	ACCESS_NETWORK_STATE INTERNET SET_WALLPAPER
Dangerous	Possibly causing harm	READ_CONTACTS ACCESS_FINE_LOCATION READ_PHONE_STATE
Signature	Automatically granted if the app is signed by the same digital certificate as the app that created the permission	ACCESS_VR_MANAGER WRITE_BLOCKED_NUMBERS BIND_TRUST_AGENT
SignatureOrSystem	Similar to Signature except automatically granted to the Android system image in addition to the requesting app	GET_APP_OPS_STATS MANAGE_DEVICE_ADMINS ACCESS_CACHE_FILESYSTEM

The permissions system has known deficiencies. First, apps tend to request for excessive permissions. Second, users tend to grant permissions to apps without fully understanding the permissions or their implications in terms of risk. Third, the permissions system is concerned only with limiting the actions of individual apps. It is possible for two or more apps to collude for a malicious action by combining their permissions, even though each of the colluding apps is properly restricted by permissions.

3 App Collusion

ISO 27005 defines a threat as “A potential cause of an incident, that may result in harm of systems and organisations.” For mobile devices, the range of such threats includes [62]:

- Information theft happens when information is sent outside the device boundaries.
- Money theft happens, e.g., when an app makes money through sensitive API calls (e.g. SMS).
- Service or resource misuse occurs, for example, when a device is remotely controlled or some device function is affected.

As we have seen before, the Android OS runs apps in sandboxes, trying to keep them separate from each other, especially that no information can be exchanged between them. However, at the same time Android has communication channels between apps. These can be documented ones (overt channels), or undocumented ones (covert channels). An example of an overt channel would be a shared file or intent; an example of a covert channel would be volume manipulation (the volume is readable by all apps) in order to pass a message in a special code.

Broadly speaking, app collusion is when, in performing a threat, several apps are working together, i.e., they exchange information which they could not obtain on their own.

This informal definition is close to app collaboration, where several apps share information (which they could not obtain on their own), in order to achieve a documented objective.

A typical example of collusion is shown in Figure 2, where two apps perform the threat of information theft: the `Contact_app` reads the contacts database to pass the data to the `Weather_app`, which sends the data outside the device boundaries. The information between apps is exchanged through shared preferences.

In contrast, a typical example of collaboration would be the cooperation between a picture app and an email app. Here, the user can choose a picture to be sent via email. This requires the picture to be communicated over an overt channel from the picture app to the email app. Here, the communication is performed via a shared image file, to which both apps have access.

These examples show that the distinction between collusion and collaboration actually lies in the notion of intention. In the case of the weather app, the intent is malicious and undocumented, in the case of sending the email, the intent is documented, visible to the user and useful.

To sharpen the argument, it might be the case that the picture app actually makes the pictures readable by all apps, so that harm can be caused by some malicious app sending pictures without authorisation. This would provide a situation, where a bug or a vulnerability of one app is abused by another app, leading to a border case for collusion. In this case one would speak about “confused deputy” attack: the picture app has a vulnerability, which is maliciously abused by the other app, however, the picture app was – in the way we describe it here – not designed with the intention to collude. An early reference on such attacks is the work by Hardy [34].

This discussion demonstrates that notions such as “malicious”, intent, and visibility (including app documentation - external and built-into the app) play a role when one wants to distinguish between collusion, cooperation, and confused deputy. This is typical in cyber security, see e.g. Harley’s book chapter “Antimalware evaluation and Testing”, especially the section headed “Is It or Isn’t It?”, p.470–474, [35]. It

is often a challenge, especially for borderline cases, to distinguish between benign and malicious application behaviours. One approach is to use a pre-labeled "malicious" data set of APKs where all the aforementioned factors have been already accounted for. Many security companies routinely classify Android apps into clean and malicious categories to provide anti-malware detection in their products and we had access to such set from Intel Security (McAfee). All apps classified as malicious fall into three mentioned threat categories. Now, collusion can be regarded as a camouflage mechanism applied to conceal these basic threat's behaviours. After splitting malicious actions into multiple individual apps they would easily appear harmless when checked individually. Indeed, even permissions of each such app would indicate it cannot pose a threat in isolation. But in combination, however, they may realise a threat. Taking into account all the details contributing to "maliciousness" – deceitful distribution, lack of documentation, hidden functionality, etc. – is practically impossible to formalise.

In our book chapter, we aim to apply purely technical methods to discover collusion. Thus, we will leave out of our definition all aspects relating to psychology, sociology, or documentation. In the light of the above discussion our technical definition of collusion thus applies to all three identified cases, namely collusion, cooperation, and confused deputy. If someone aims to distinguish between these, then further manual analysis would be required, involving the distribution methods, documentation, and all other surrounding facts and details.

When analysing the APKs of various apps for collusion, we look at the actions that are being executed by these APKs. Actions are operations provided by the Android API (such as record audio, access file, write file, send data, etc.). We denote the set of all actions by Act. Note that this set also includes actions describing communication. Using an overt channel in Android requires an API call.

An action can be can be characterised by a number of static attributes such as permissions, e.g., when an app needs to record audio, the permission RECORD_AUDIO needs to be set in the manifest while the permission WRITE_EXTERNAL_STORAGE needs to be set for writing a file.

Technically, we consider a threat to be a sequence of actions. We consider a threat to be realised by collusion if it is distributed over several apps, i.e.,

Definition 1. there is non-singleton set S of apps such that:

- each app in S contributes the execution of at least one action to the threat,
- each app in S communicates with at least one other app.

This definition will be narrowed down further when discussing concrete techniques for discovering collusion.

To illustrate our definition we present an abstract example[1].

Example 1 (Stealing contact data). The two apps graphically represented in Figure 2 perform information theft: the `Contact_app` reads the contacts database to pass the data to the `Weather_app`, which sends the data outside the device boundaries. The information is sent through shared preferences.

[1] Concrete examples are available on request.

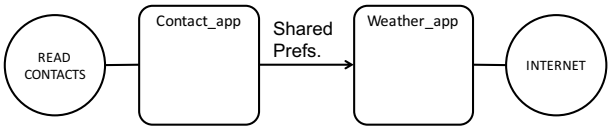

Fig. 2: An example of colluding apps

Using the collusion definition we can describe the actions performed by both apps as:

- $Act_{\texttt{Contact\_app}} = \{a_{read\_contacts}, send_{shared\_prefs}, \}$ and
- $Act_{\texttt{Weather\_app}} = \{a_{send\_file}, recv_{shared\_prefs}, \}$

with the permissions $pms\ (a_{read\_contacts}) = \{Permission\_contacts\}$ and $pms(a_{send\_file}) = \{Permission\_internet\}$. The information threat T is given by

$$T = \langle a_{read\_contacts}, send_{shared\_prefs}, recv_{shared\_prefs}, a_{send\_file} \rangle$$

.

This data leakage example is in line with the collusion definitions given in most existing work [52, 42, 46, 40, 17, 5] which regards collusion as the combination of inter-app communication with information leakage. However, our definition of a threat is broader, as it includes also financial and resource / service abuse.

3.1 App Collusion in the Wild

We present our analysis of a set of apps in the wild that use collusion to maximise the effects of their malicious payloads [11]. To the best of our knowledge, this is the first time that a large set of colluding apps have been identified in the wild. This does not necessarily mean that there are no more colluding apps in the while, as one of the main problems (that we are addressing in our work) is the lack of tools to identify colluding apps. We identified these sets of apps while looking for collusion potential on a set of more than 40,000 apps downloaded from App markets. While performing this analysis we found 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 synchronise the execution of a potentially harmful payload. Both the colluding and malicious payload were included inside a library, the MoPlus SDK, embedded in all apps. This library has been known to be malicious since November 2015 [58]. However, the collusion behaviour of the SDK was hitherto unknown. In the rest of this section, we briefly describe this colluding behaviour.

The detected colluding behaviour looked different from the behaviour predicted by most app collusion research [57, 47] so far. 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 C&C server, maximising the effects of the malicious payload.

The MoPlus SDK includes the MoPlusService and the MoPlusReceiver components. In all analysed apps, the service is exported. In Android, this is considered to be a dangerous practice, as also 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. To establish the app priority (Figure 3, the MoPlus SDK executes a number of checks, including the verifying if the app embedding the SDK has granted the INTERNET, READ_PHONE_STATE, ACCESS_NETWORK_STATE, WRITE_CONTACTS, WRITE_EXTERNAL_STORAGE or GET_TASKS permissions.

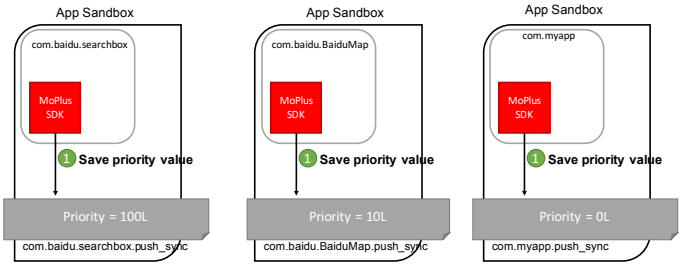

Fig. 3: Phase 1 of the colluding behaviour execution. Each app saves a priority value that depends on the amount of access it has to the system resources. Priority values are shown for the sake of explanation.

After the priority has been obtained and stored, each service inspects the contents of the shared preference files to get its priority, returning the package name of the one with highest priority. Then, each service cancels previous intents being registered (to avoid launching the service more than once) and sends an intent targeting only the process with the higher previously saved priority (Figure 4).

3.1.1 Discussion

It is important to notice that although the applications already include a malicious payload that could be executed on their own, if two apps with the Moplus SDK were to be installed in the same device, they would not be able to execute their individual malicious payloads. Although this assumption may seem unrealistic at

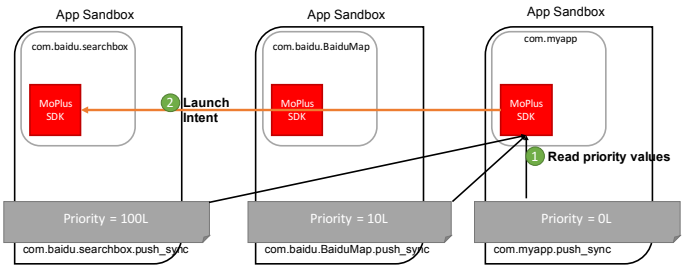

Fig. 4: Phase 2 of the colluding behaviour execution. Each app checks the WORLD_READABLE SharedPreference files and sends and intent to the app with highest priority

first, implementing these kinds of behaviours inside SDKs makes this much more likely to happen. If we consider this assumption, then, the colluding behaviour allows two things: first, it enables the execution of the malicious payload avoiding concurrency problems between all instances of the SDK running. Second, it allows the SDK instance with highest access to resources to be the one executing, maximising the result of the attack. This introduces an important remark in how colluding applications have to be analysed. This is, having the static features that allow them to execute a threat doesn't mean they will be able to achieve that threat in all scenarios, like the one presented in our case. This means, that when considering app collusion we must look not only to the specific features or capabilities of the app, but also how those capabilities work when the app is being executed with other apps. If we are considering collusion it does not make much sense to consider the capabilities of an app in isolation with respect to other apps, we have to consider the app executing in an environment where there are other apps installed.

3.1.2 Relation with Collusion Definition

This set of apps found in the wild relates to our collusion definition in the following way. Consider a set of apps $S = \{app_1, app_2, \cdots, app_n\}$ that implements the MoPlus SDK. As, they embed the MoPlus SDK, the attacks that can be achieved by them includes writing into the contacts database, launching intents and installing applications without user interaction among others. This set of threats was identified by TrendMicro researchers [58].

Consider now the installation of an application without the user interaction as a threat $T_{install}$. As all apps embed the MoPlus SDK, all apps include the code to potentially execute such threat, but only apps that request the necessary permissions are able to execute it. If app_i is the only app installed in the device, and has the necessary permissions, executing $T_{install}$ will require the following actions $\{Open\ server_i, Receive\ command_i, Install\ app_i\}$, the underscore being the app executing the action.

However, if another MoPlus SDK app, app_j, is installed in the same device but doesn't have the permissions required to achieve $T_{install}$ the threat won't be realised because of concurrency problems, both apps share the port where they receive the commands. To avoid these, the MoPlus SDK includes the previously described leader selection mechanisms that uses the SharedPreferences. In this setting, we can describe the set of actions required by both apps to execute the threat as $Act_{Moplus} = \{Check\ permissions_i, Check\ permissions_j, Save\ priority\ i_i, Save\ priority\ j_j, Read\ priority\ i_j, Read\ priority\ j_i, Launch\ service\ i_j, Open\ server_i, Receive\ command_i, Install\ app_i\}$. Considering *Read priority* x_y and *Save priority* x_y as actions that make use of the SharedPreferences as a communication channel, we can consider that the presented set of actions follows under our collusion definition as (1) there is a sequence of actions that execute a threat executed collectively by app_i and app_j and (2) both apps communicate with each other.

4 Filtering

A frontal attack on detecting collusions to analyse pairs, triplets and even larger sets is not practical given the search space. An effective collusion-discovery tool must include an effective set of methods to isolate potential sets which require further examination.

4.1 Rule based collusion detection

Here, in a first step we extract information about app communications and access to protected-resources. Using rules in first order logic codified in Prolog, the method identifies sets of apps that might be colluding.

The goal of this is to serve as a fast, computationally cheap filter that detects potentially colluding apps. For such a first filter it is enough to be based on permissions, In practical work on real world apps this filter turns out to be effective to detect colluding apps in the wild.

Our filter (1) uses Androguard [20] to extract facts about the communication channels and permissions of all single apps in a given app set *S*, (2) which is then abstracted into an over-approximation of actions and communication channels that could be used by a single app. (3) Finally the collusion rules are fired if the proper combinations of actions and communications are found in *S*.

4.1.1 Actions

We utilise an action set Act_{prolog} composed out of four different high level actions: accessing sensitive information, using an API that can directly cost money, control-

ling device services (e.g. camera, etc.), and sending information to other devices and the Internet. To find out which of these actions an app could carry out, we extract its set of permissions pms_{prolog} with Androguard. For each found permission, our tool creates a new Prolog fact in the form $uses(app, permission)$. Then permissions extracted are mapped to one of the four high level actions. This is done with a set of previously defined Prolog rules. The mapping of all Android permissions to the four high-level actions can be found in the project Github repository[2]. As an example, an app that declares the INTERNET permission will be capable of sending information outside the device:

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

4.1.2 Communications

The communication channels established by an app are characterised by its API calls and the permissions declared in its manifest file. We cover communication actions (com_{prolog}) that can be created as follows:

- *Intents* are messages used to request tasks from other application components (activities, services or broadcast receivers). Activities, services and broadcast receivers declare the intents they can handle by declaring a set of intent filters.
- *External Storage* is a storage space shared between all the apps installed without restrictions. Apps accessing the external storage need to declare the

  READ_EXTERNAL_STORAGE

  permission. To enable writing, apps must declare

  WRITE_EXTERNAL_STORAGE.

- *Shared Preferences* are an OS feature 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 permissions are defined (before Android 4.4).

We map apps to sending and receiving actions by inspecting their code and manifest files. When using intents and shared preferences we are able to specify the communication channel using the intent actions and preference files and packages respectively. If an application sends a broadcast intent with the action SEND_FILE we consider the following:

$$send\_broadcast(App, Intent_{send\_file}) \rightarrow send(App, Intent_{send\_file})$$

We consider that two apps communicate if one of them is able to *send* and the other to *receive* through the same channel. This allows to detect communication paths composed by an arbitrary number of apps:

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

$$send(App_a, channel) \wedge receive(App_b, channel) \rightarrow communicate(App_a, App_b, channel)$$

4.1.3 Collusion Potential

To identify collusion potential in app sets, we put together the different communication channels found in an app and their high-level actions as identified by their permissions. Then, using domain knowledge we created a threat set that describes some of the possible threats that could be achieving with a collusion attack. Our threat set τ_{prolog} considers information theft, money theft and service misuse. As our definition states, each of the threats is characterised by a sequence of actions. In fact, each of our collusion rules gathers the two elements required by the collusion definition explained in Section 3: (1) each app of the group must execute at least one action of the threat and (2) each app in S communicates at least with another app in S. The following rule vies an example of an information threat executed through two colluding apps:

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

Note that more apps could be involved in this same threat as simply forwarders of the information extracted by the first app until it arrives to the exfiltration app. This case is also covered by Definition 1, as the forwarding app need to execute their communication operations to succeed on their attack (fulfilling both of our definition conditions).

Finally, the Prolog rules defining collusion potential, the facts extracted from apps, and rules mapping permissions to high level actions and communications between apps are then put on a single file. This file is then fed into Prolog so collusion queries can be made. The overall process is depicted in Figure 5.

4.2 Machine learning collusion detection

Security solutions using machine learning employ algorithms designed to distinguish between malicious and benign apps. To this end, they analyse various features such as the APIs invoked, system calls, data-flows and resources used assuming a single malicious app attack model. In this section, we extend the same notion to assess the collusion threat which serves as an effective filtering mechanism for finding collusion candidates of interest. We employ probabilistic generative modelling for this task with the popular Naive Bayesian model.

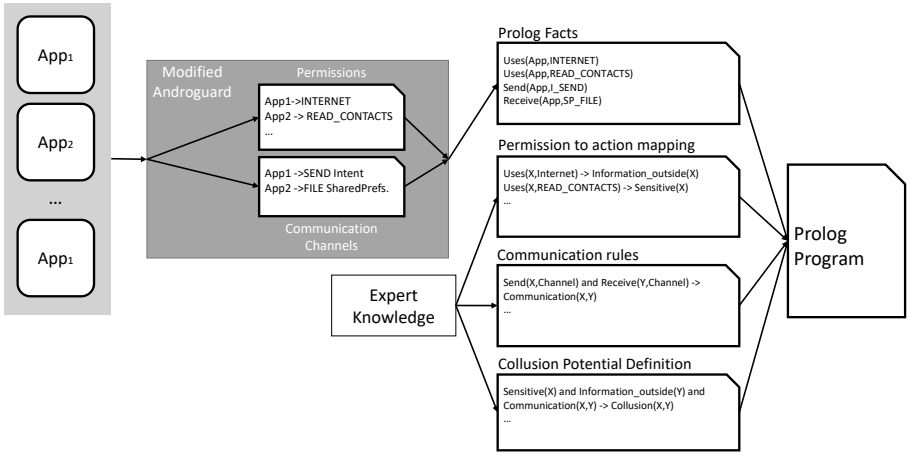

Fig. 5: General overview of the process followed in the rule based collusion detection approach.

4.2.1 Naive Bayes classifier

Let $X = [x_1, \ldots, x_k]$ be a k-dimensional space with binary inputs, where k is the total number of permissions in Android OS and $x_j \in \{0,1\}$ are independent Bernoulli random variables. A variable x_j takes the value 1 if permission j is found in the set S of apps under consideration, 0 otherwise. Let Y = {m-malicious, b-benign} be a one dimensional output space. The generative naive Bayes model specifies a joint probability, $p(x,y) = p(x \mid y).p(y)$, of the inputs x and the label y: the probability $p(x,y)$ of observing x and y at the same time is the same as the probability $p(x \mid y)$ of x happening when we know that y happens multiplied with the probability $p(y)$ that y happens. This explicitly models the actual distribution of each class (i.e. malicious and benign in our case) assuming that some parameters stochastic process generates the observations, and hence estimates the model parameters that best explains the training data. Once the model parameters are estimated (say $\hat{\theta}$), then we can compute $p(t_i \mid \hat{\theta})$ which gives the probability of the i^{th} test case is generated by the derived model. This can be applied in a classification problem as explained below.

Let p(x,y) be a joint distribution over $X \times Y$ from which a training set $R = \{x_i^k, y_i^1 \mid i = 1,2,3,...,n\}$ of n independent and identically distributed examples are drawn. The generative naive Bayes classifier uses R to estimate $p(x|y)$ and $p(y)$ in the joint distribution. If $c(.)$ stands for counting the number of occurrences of an event in the training set,

$$\hat{p}(x = 0 \mid y = m) = \frac{c(x = 0, y = m) + \alpha}{c(y = m) + 2\alpha} \tag{1}$$

where the pseudo count $\alpha > 0$ is the smoothing parameter. If $\alpha = 0$, i.e. taking the empirical estimates of the probabilities without smoothing, then

$$\hat{p}(x = 0 \mid y = m) = \frac{c(x = 0, y = m)}{c(y = m)} \tag{2}$$

Equation (2) estimates the likelihoods using the training set R. Uninformative priors, i.e. $\hat{p}(y = m)$, can also be estimated in the same way. Instead, we estimate prior distribution in an informative way in this work as it would help us in modelling the knowledge not available in data (e.g. permission's critical level). Informative prior estimation is described in section 4.2.3.

In order to classify the i^{th} test case, the model predicts $p(t_i \mid \hat{\theta})$ = m if and only if:

$$\frac{\hat{p}(x = t_i, y = m)}{\hat{p}(x = t_i, y = b)} > 1 \tag{3}$$

4.2.2 Threat likelihood

As per our collusion definition in Section 3, estimating the collusion threat likelihood $L_c(S)$ of a non-singleton set S of apps involves two likelihood components $L_\tau(S)$ and L_{com}(S). $L_\tau(S)$ expresses how likely the app set S can fulfil the sequence of actions required to execute a threat, and $L_{com}(S)$ is the ability to communicate between apps in S. Using the multiplication rule of well-known basic principles of counting:

$$L_c(S) = L_\tau(S) \times L_{com}(S) \tag{4}$$

As mentioned above, we employ so-called Naive Bayesian informative [50] model to demonstrate the evaluation of equation (4). First, we define a model, then train and validate the model, and finally test it using a testing data set.

4.2.3 Estimating L_τ

Let $X = [x_1, \ldots, x_k]$ be a k-dimensional random variable as defined in section 4.2.1. Then the probability mass function $P(X)$ gives the probability of obtaining S with permissions as described in X. Our probabilistic model $P(X)$ is then given by the equation (5):

$$P(X) = \prod_{j=1}^{k} \lambda_j^{x_j} (1 - \lambda_j)^{1 - x_j} \tag{5}$$

where $\lambda_j \in [0, 1]$ is a Bernoulli parameter. In order to compute L_τ for a given set S, we define a sample statistic as $Q_s = \frac{ln\{(P(X))^{-1}\}}{|S|}$, divide $ln\{(P(X))^{-1}\}$ by the number of distinct permissions in set S, and scale down it to the range [0,1] by dividing the max(Q_s) which estimated using empirical data. Hence, for a given set S, $L_\tau = \frac{Q_s}{max(Q_s)}$. The desired goal behind the above mathematical formulation is to make

requesting more critical permissions to increase the likelihood of "being malicious" than requesting less critical ones regardless of frequencies. Readers who require a detailed explanation of the mathematical notion behind the above formulation are invited to refer [50].

To complete our modelling, we need to estimate values $\hat{\lambda_j}$ that replace λ_j in the computation of L_τ. To this end – to avoid over fitting $P(X)$ – we estimate λ_j using informative beta prior distributions [41] and define the maximum posterior estimation

$$\hat{\lambda_j} = \frac{\sum x_j + \alpha_j}{N + \alpha_j + \beta_j} \tag{6}$$

where N is the number of apps in the training data set and α_j, β_j are the penalty effects. In this work we set $\alpha_j = 1$. The values for β_j depend on the critical level of permissions as given in [55, 50]. β_j can take either the value $2N$ (if permission j is most critical), N (if permission j is critical) or 1 (if permission j is non-critical).

4.2.4 Estimating L_{com}

In order to materialise a collusion, there should be an inter app communication *closely related* to the target threat. To establish this association we need to consider number of factors including the contextual parameters. At this stage of the research we do not focus on estimating the strength of connection (association) between the threat and the communication. Instead we investigate what percentage of communication channels can be detected through static code analysis, and simply assume[3] these channels can be used for malicious purpose by apps in set S. Hence we consider L_{com} to be a binary function such that $L_{com} \in \{1,0\}$ which takes the value 1 if there is inter app communication within S using either intents or external storage (we do not investigate other channels in this work).

4.2.5 Proposed probabilistic filter

Our probabilistic filter consists of two sub filters: an inner and an outer one. The inner filter applies on the top of the outer filter. The outer filter is based on the L_τ value which we can compute using permissions only. Permissions are very easy and cheap to extract from APKs - no decompilation, reverse engineering, complex code or data flow analysis is required. Hence the outer filter is computationally efficient. The majority of non-colluding app pairs in an average app set can be treated using this filter only (see Figure 6). This avoids the expensive static/dynamics analysis on these pairs. The inner filter is based on L_{com} value which we currently compute using static code analysis. A third party research prototype tool Didfail [15] was employed in finding intent based inter app communications. A set of permission

[3] This assumption might produce false positives, however, never false negatives. It is left as a future work to improve this.

based rules was defined to find communication using external storage. Algorithm 1 presents the proposed filter to find out colluding candidates of interest.

Algorithm 1: Probabilistic filter. The outer filter is based on L_τ and the inner filter is based on L_{com}.

```
Λ: Set of individual apps;
Ω: Set of pairs of colluding candidates of interest;
input  : Λ={app_1, app_2, app_3, ... , app_n}
output: Ω={pair_1, pair_2, pair_3, ... , pair_m}
if |Λ| ≥ 2 then
    Let Θ = set of all possible app pairs in Λ;
    foreach pair_j in Θ do
        Compute L_τ as described in section 4.2.3;
        /* outer filter                                   */
        if L_τ ≥ threshold then
            Compute L_com as described in section 4.2.4 ;
            /* inner filter                               */
            if L_com == 1 then
                Return (pair_j);
            end
        end
    end
end
```

4.2.6 Experimental setup and validation

Algorithm 1 was automated using R[4] and Bash scripts. As mentioned above, it also includes calls to a third party research prototype [15] to find intent based communications in computing L_{com}. Model parameters in Equation 5 was estimated using training datasets produced from a 29k size app sample provided by Intel security.

Our validation data set consists of 240 app pairs in which half (120) of them are known colluding pairs while the other half non-colluding pairs. In order to prevent over fitting, app pairs in the validation and testing sets were not included in the training set. As shown in Figure 6 proposed method assigns higher L_τ scores[5] for colluding pairs than clean pairs. Table 2 presents the confusion matrix obtained for the proposed method by fitting a linear discriminant line (LDL), i.e. the blue dotted line in Figure 6. Sensitivity=0.95, specificity=0.94, precision=0.94 and the F-score=0.95.

[4] http://www.r-project.org/

[5] We plot L_τ values in Figure 6 as outer filter in algorithm 1 depends on it, and to show that majority of non-colluding app pairs can be treated using L_τ only. However, it should be noted that $L_c = L_\tau$ for colluding pairs as $L_{com} = 1$.

However as shown in Figure 6, colluding and non-colluding are not easily separable two classes by a LDL. There are some overlaps between class elements. As a result it produces false classifications in Table 2. It is possible to reduce false alarms by changing the threshold. For example either setting the best possible discriminant line or its upper bound (or even higher, see figure 6) as the threshold will produce zero false positives or vice versa in Table 2. But as a result it will increase false negative rate that will affect on the F-score - the performance measure of the classifier. Hence it would be a trade-off between a class accuracy and overall performance. However since the base rate of colluding apps in the wild is close to zero as far as anyone knows, the false positive rate of this method would have to be vanishingly small to be useful in practice. Instead of LDL, using a non-linear discriminator would also be another possibility to reduce false alarms. This is left as a future work to investigate.

The average processing time per app pair was 80s which consists of $\leq 1s$ for the outer filter and rest of the time for the inner filter. Average time was calculated on a mobile workstation with an Intel Core i7-4810MQ 2.8GHz CPU and 32GB of RAM.

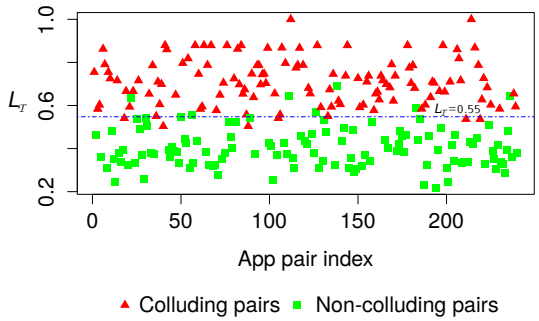

Fig. 6: Validation: L_τ score obtained by each pair in the validation data set.

Table 2: Confusion matrix for the naive Bayesian method.

n=240	**Actual Colluding**	**Actual Non-colluding**
Predicted Colluding	114	7
Predicted Non-colluding	6	113

4.3 Evaluation of filtering

We validate both our filtering methods against a known ground truth by applying them to a set of artificially created apps. Furthermore, we report on managing complexity by scaling up our rule based detection method to deal with 50000+ real world applications.

4.3.1 Testing the Prolog filter

The validation of the Prolog filter has been carried out with fourteen artificially crafted apps that cover information theft, money theft and service misuse. Created apps use Intents, Shared Preferences and External storage as communication channels. They are organised in four colluding sets:

- The **Document Extractor** set consists of one app (*id 1*) that looks for sensitive documents on the external storage; the other app (*id 2*) sends the information received (via SharedPreferences) to a remote server.
- The **Botnet** set consists of four apps. One app (*id 3*) acts as a relay that receives orders from the command and control center. The other colluding apps execute commands (delivered via BroadcastIntents) depending on their permissions: sending SMS messages (*id 4*), stealing the user's contacts (*id 5*) and starting and stopping tasks (*id 6*).
- The **Contact Extractor** set consists of three apps. The first (*id 7*) reads contacts from the address book, the second (*id 8*) forwards them via the external storage to the third one (*id 9*), which sends them to the Internet. The first and second app communicate via BroadcastIntents .
- The **Location Stealing** set consists of one app (*id 12*) that reads the user location and shares it with the second app (*id 13*), which sends the information to the Internet.

The three non-colluding apps are a document viewer (*id 10*), an information sharing app (*id 11*) and a location viewer (*id 14*). The first app is able to display different file types in the device screen and use other apps (via broadcast intents) to share their uniform resource identifier (URI). The second app receives text fragments from other apps and sends them to a remote server. The third app receives a location from another app (with the same intent used by apps 12 and 13) and shows it to the user on the screen.

Table 3 shows the results obtained with our rule based approached. The entry "dark red club" in row 1 and column 2 means: the program detects that app *id 1* sends information to app *id 2*, and these two apps collude on an "information theft". As we take communication direction into consideration, the resulting matrix is non-symmetric, e.g., there is no entry in row 2 and column 1. The entry "light red club" in row 1 and column 10 means: the program flags collusion of type "information theft" though the set $\{id\ 1, id\ 10\}$ is clean.This provides further information about the collusion attack. For instance, one can see the information leak in information theft

Table 3: Collusion Matrix of the Prolog program. ♣ = Information theft. $ = Money theft. ♠ = Service misuse. ♣, $, ♠ = False positives.

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1		♣								♣	♣			
2														
3				$ ♣	♠	♠								
4														
5			♣							♣	♣			
6			♣	♣										
7		♣						♣	♣	♣	♣			
8									♣					
9														
10											♣			
11														
12													♣	♣
13														
14														

attacks. Additionally, the way we defined the communication rules makes it possible to identify transitive collusion attacks (i.e. app 7 colluding with app 9 through app 8). The approach identifies all colluding app sets. It also flags eight false positives due to over-approximation. Note, that there are no false negatives due to the nature of our test set: it utilises only those communication methods that our Prolog approach is able to identify.

Our false positives happen mainly because two reasons. First, we do not consider in our initial classification some of the communication channels that are already widely use by apps in Android. For example, the Intent with action VIEW or SEND are very common in Android applications. It is unlikely that apps would use them for collusion as other apps could have registered to receive the same information. Second, in this approach, we identify apps that are communicating by sharing access to sensitive resources, but we do not look at how that access is shared. It must be noted, that the main aim of this approach is to reduce the amount of app combinations that are being passed through the data-flow analysis.

4.3.2 Testing the Probabilistic filter

We tested the Probabilistic filter with a different sample consisting of 91 app pairs. Figure 7 presents the outcome for this set. Each cell in the table denotes a L_τ value for the corresponding app pair. Note that though there are 196 possible pairs (i.e. 14x14 cells in the table), for readability, we leave the lower half empty since the table is symmetric. Pairs on the diagonal are also not interesting to our discussion. To minimise false negatives, we use the lower bound (=0.50) gained from the validation dataset for the discriminant line as threshold for L_τ. We report possible collusion if $L_\tau \geq 0.5$ and $L_{com} = 1$, otherwise we report non-collusion. Dark red shows true

positives, light red shows false positives, dark green shows true negatives, and light green shows false negatives.

With regards to false alarms, app pair (1,2) was not detected by our analysis due to the third party tool does not detect communication using SharedPreferences. Since we do only pairwise analysis, app pair (7,9) was not reported. That pair depends on transitive communication. Pair (12,13) was not reported since L_τ is less than the chosen threshold. As mentioned in section 4.2.6, it would be possible to reduce false alarms by changing the LDL threshold, but subject to degrading the overall performance measure of the classifier.

Precise estimation of L_{com} would be useful to reduce false alarms in our analysis. But it should be noted that existence of a communication is only a necessary condition to happen a collusion, but not a sufficient condition to detect it. In this context it is worth to mention that a recent study [23] shows that 84.4% of non-colluding apps in the market place can communicate with other apps either using explicit (11.3%) or implicit (73.1%) intent calls. Therefore the threat element (i.e. L_τ) is far more informative in collusion estimation than the communication element (L_{com}) in our model.

Both validation and testing samples are blind samples and we have not properly investigated them for the biasedness or realisticity.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1		0.51	0.61	0.97	1	0.8	1	0.81	0.77	0.77	0.77	0.44	0.44	0.95
2			0.48	0.62	0.55	0.49	0.55	0.58	0.51	0.51	0.58	0.31	0.31	0.49
3				0.69	0.64	0.56	0.64	0.48	0.61	0.61	0.72	0.41	0.41	0.58
4					1	0.84	1	0.85	0.71	0.71	0.82	0.56	0.56	0.95
5						0.84	1	0.86	0.67	0.67	0.82	0.47	0.47	1
6							0.84	0.68	0.58	0.58	0.65	0.43	0.43	0.78
7								0.86	0.67	0.67	0.82	0.47	0.47	1
8									0.51	0.51	0.58	0.31	0.31	0.77
9										0.77	0.77	0.44	0.44	0.61
10											0.77	0.44	0.44	0.61
11												0.47	0.47	0.73
12													0.47	0.41
13														0.41
14														

Fig. 7: Testing the proposed filter. For readability – we leave the upper half empty since the table is symmetric. Pairs on the diagonal are also not interesting to our discussion. Dark red shows true positives, light red shows false positives, dark green shows true negatives, and light green shows false negatives.

5 Model-checking for collusion

Filtering is an effective method to isolate app sets. Using software model checking, we provide a sound method for proving app sets to be clean that also returns example traces for potential collusion based on the $\mathbb{K}$ framework [54] – c.f. Figure 8. We start with a set of apps in the form of an Application Package File (APK). The DEX code

in each APK file is disassembled into the Smali format with open source tools. The Smali code of the apps is parsed by the $\mathbb{K}$ tool. Compilation in the $\mathbb{K}$ tool translates the $\mathbb{K}$ representation of the Android apps into a rewrite theory in Maude [18]. Finally, the Maude model checker searches the transition system compiled by the $\mathbb{K}$ tool to provide an answer if the input set of Android apps colludes or not. In the case when collusion is detected, the tool provides a readable counter example trace. In this section we focus on information theft only.

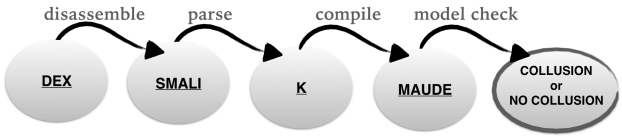

Fig. 8: Work-flow for the Android formal semantics in the $\mathbb{K}$ framework.

5.1 Software model checking

Software model checking is a methodology widely used for the verification of properties about programs w.r.t. their executions. A profane view on model checking would be to see it as instance of the travelling salesman problem: every state of the system shall be visited and checked. This means that, upfront, model checking is nothing but a specialised search in a certain type of graph or, as it is known in the field, a *transition system*.

Initially, the application of model checking focused on simple transition systems, especially coming from hardware. *Simplicity* was necessary to contain a notorious efficiency problem known as the "state space explosion. Namely, the methodology would fail to produce timely efficient results due to the exponential nature of the complexity of the model checking procedures w.r.t. the number of system states.

Modern model checking tools attempt to meet the challenge posed by (higher level) programs, i.e., software, that are known to quickly produce a large (potentially unbounded) number of states, e.g., due to dynamic data structures, parallelism, etc. Hence, software model checking uses, in addition to basic model checking, other techniques (e.g. theorem proving or abstract interpretation) in order to coherently simplify the transition system given to the model checker.

A standard example is given by imperative programming languages. Here, a program p is viewed as a sequence of program locations $pc_i, i \geq 0$, that identify instructions. The effect of an instruction I at pc_i is a relation r_i which associates the states before with the states after the execution of I. Software model checking computes the transitive closure R of the relations r_i to obtain the set of *reachable* states of the program p.

Note, however, that for infinite state programs the computation of R may not terminate or may require an unreasonable amount of time or memory to terminate. Hence software model checking transforms the state space of the program into a "simpler" one by, essentially, eliminating unnecessary details in the relation r_i thus obtaining an *abstract* program a defined by the relations $a(r_i)$. The model checking of a, usually named "abstract" model checking, trades off precision for efficiency. A rigorous choice of the abstract set of states (i.e. abstract domain) and the abstract relations $a(r_i)$ (i.e. abstract semantics) ensures that the abstract model checking is sound (i.e. proving the property in the abstract system implies the property is proved in the original, concrete, system).

5.1.1 Challenges

In the following we will explain how we define a transition system using $\mathbb{K}$ and what abstractions we define in order to allow for an effective check for collusion.

Formalising Dalvik Byte-code in $\mathbb{K}$ poses a number of challenges: there are about 220 instructions to be formalised, the code is object oriented, it is register based (in contrast to stack based, as Java Byte-code), it utilises callbacks and intent based communication, see [3]. We provide two different semantics for DEX code, namely a concrete and an abstract one. While the concrete semantics has the benefit to be intuitive and thus easy to be validated, it is the abstract semantics that we employ for app model checking. We see the step from the descriptive level provided by [3] to the concrete, formal semantics as a 'controllable' one, where human intuition is able to bridge the gap. In future work, we intend to justify the step from the concrete semantics to the abstract one by a formal proof. Our implementation of both Android semantics in $\mathbb{K}$ is freely available[6]. The code of the colluding apps discussed in this paper is accessible via an encrypted web-page. The password is available on request[7].

5.2 The $\mathbb{K}$ framework

The $\mathbb{K}$ framework [54] proposes a methodology for the design and analysis of programming languages; the framework comes with a rewriting-based specification language and tool support for parsing, interpreting, model-checking and deductive formal verification. The ideal work-flow in the $\mathbb{K}$ framework starts with a formal and executable language syntax and semantics, given as a $\mathbb{K}$ specification, which then is tested on program examples in order to gain confidence in the language definition. Here, the $\mathbb{K}$ framework offers model checking via compilation into Maude programs (i.e., using the existing reachability tool and LTL Maude model checker).

[6] http://www.cs.swan.ac.uk/~csmarkus/ProcessesAndData/androidsmali-semantics-k

[7] http://www.cs.swansea.ac.uk/~csmarkus/ProcessesAndData/sites/default/files/uploads/resources/code.zip.

A $\mathbb{K}$ specification consists of *configurations*, *computations*, and *rules*, using a specialised notation to write semantic entities, i.e., $\mathbb{K}$-cells. For example, the $\mathbb{K}$-cell representing the set of program variables as a mapping from identifiers *Id* to values *Val* is given by $\langle Id \mapsto Val \rangle_{\mathsf{vars}}$. Configurations in $\mathbb{K}$ are labelled and nested $\mathbb{K}$-cells, used to represent the structure of the program state. Rules in $\mathbb{K}$ are of two types: computational and structural. Computational rules represent transitions in a program execution and are specified as configuration updates. Structural rules provide internal changes of the program state such that the configuration form can enable the application of computational rules.

5.3 A concrete semantics for Dalvik code

The concrete semantics specifies system configurations and transition rules for all Smali instructions and a number of Android API calls in $\mathbb{K}$. Here, we strictly follow their explanation [2].

5.3.1 System configurations

Configurations are defined in $\mathbb{K}$ style as cells which might contains sub-cells. Top of a configuration is a “sandboxes” cell, containing a “broadcasts” sub-cell abstracting the Android intent broadcast service and possibly multiple “sandbox” cells capturing states of installed apps.

In $\mathbb{K}$, the asterisk symbol next to the name “sandbox” specifies that the number of “sandbox” cells within a “sandboxes” cell is 0 or more. Each sandbox cell simulates

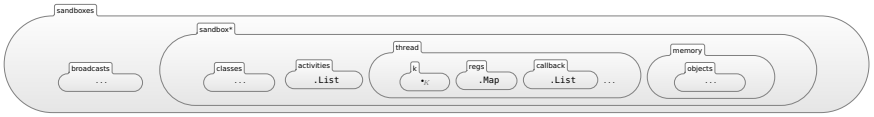

Fig. 9: Android configuration.

the environment in which an application is isolated. It contains the classes of the application, the currently executed thread, and the memory storing the objects that have been instantiated so far. For the current thread, we store the instructions left to be run in a “k” cell, while the content of the current registers are kept in a “regs” cell . Classes and Method cells can be defined similarly. In turn, each “method” cell consists of the name of the method, the return type of the method and the statements of the method within a “methodbody” cell. Finally, “object” cells are used to store the objects that have been instantiated so far. They are stored within the “memory” cell of a “sandbox”. As depicted in Figure 10, an object cell contains a reference

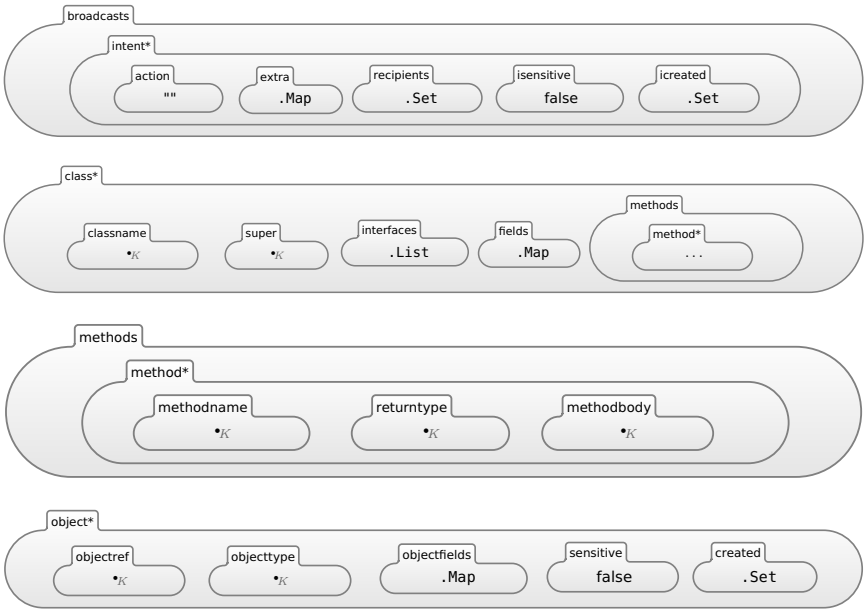

Fig. 10: Sub-cells of a configuration: broadcasts and object.

(an integer), its type, values of its corresponding fields, a Boolean value to indicate whether the object is sensitive and the set of applications that have created this object. The last two cells have been added for the sake of program analysis.

5.3.2 Smali instructions

As a concrete example of how to formalise an instruction, let us consider the `iget` $R1, R2, CN \rightarrow FN : FT$ instruction. `iget` retrieves the field value of an object. Here, *CN* is the name of a class, *FN* and *FT* are the name of the field to be read and its type, register *R*2 holds the reference to the object to be read from, and – after execution of the instruction – register *R*1 shall hold the value of the field *FN*. The $\mathbb{K}$ rule for its semantics is illustrated in Figure 11. This $\mathbb{K}$ rule is enabled when (i) the `k` cell of a thread starts with an `iget` instruction, (ii) *R*2 is resolved to a reference *I*2 of some object where (iii) *FN* maps to a value of *TV1*. When the rule is applied, *TV1* is copied into *R*1.

The semantics for Smali instructions in $\mathbb{K}$ is organised in a number of separate modules as shown in Figure 12, where arrows specify import. The "semantic-core" contains the semantics rules for basic instructions and directives such as "nop" (no operation), ".registers n" and ".locals n" where *n* is an integer. Additionally, it also defines several auxiliary functions which are used later in other modules for semantic rules. For example, the function "isKImplementedAPI" is defined to determine

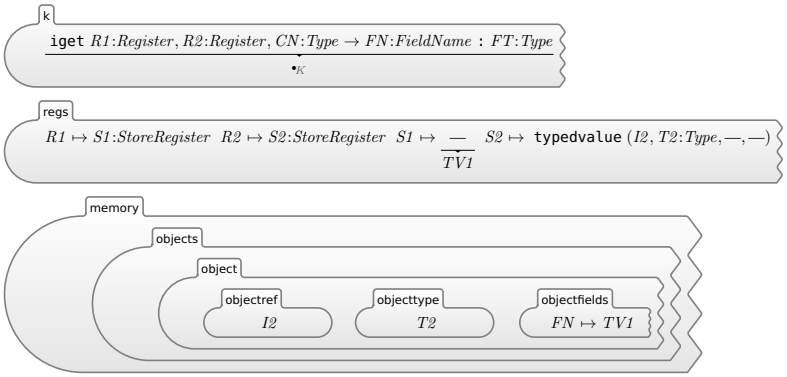

Fig. 11: $\mathbb{K}$ rule for the semantics of `iget` instruction.

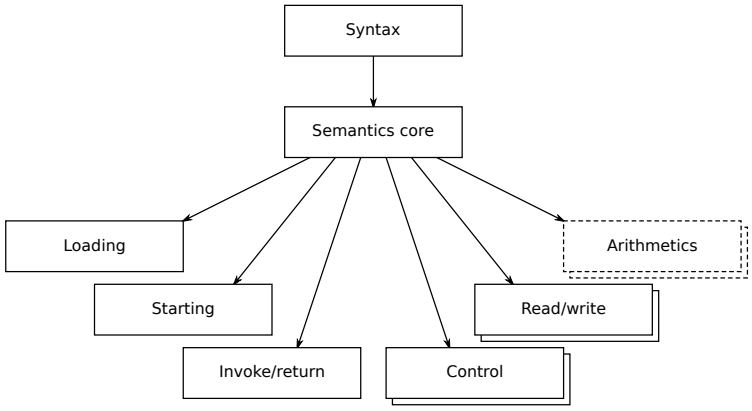

Fig. 12: Semantic module structure.

whether an API method has been implemented within the $\mathbb{K}$ framework; if not, the interpreter will look for it within the classes of the invoking application.

The "loading" module is responsible for constructing the initial configuration. When running a Smali file in the $\mathbb{K}$ framework, it will parse the file according to the defined syntax and placed the entire resulting abstract syntax tree (AST) in a cell. The rules defined in the loading module are used to take this AST and distribute its elements to the right cells of the initial configuration. In particular, each application is placed in a sandbox cell, its classes are placed in the classes cell, etc. The "invoke/return" module defines the semantic rules for invoking methods and return instructions. The "control" module specifies the semantics of instructions such as "if-then" and "goto", which may change the program counter in a non-trivially way. The "read/write" module implements the semantics of instructions for manipulating objects in the memory such as instantiating new objects or array, initialising elements of an array, retrieving value of an object field and changing the value of an

object field. Finally, the "arithmetics" module specifies the semantics of arithmetic instructions such as addition, subtraction, multiply, division and bit-wise operations.

In some situations, our semantics has to deal with unknown values such as the device's location returned by Android OS. In $\mathbb{K}$, unknown values can be represented by the built-in constant $\cdot K$. To this end, we provide for each of the "control", "read-/write", "arithmetics" modules a counter-part that is responsible for unknown values. For example, when the value to be compared with 0 in an `ifz` Smali instruction is unknown, we assume that the result is either true or false, thereby leading to a non-deterministic execution of the Smali program. Similarly, arithmetical operations propagate unknown values.

5.3.3 Semantics for the Android APIs

Regarding the semantics of the Android APIs which encompasses a rich set of predefined classes and methods, API classes and methods usually come together with Android OS on an Android device and hence are not included in the DEX code of an app. Obviously, one may obtain the Smali code of those API classes and methods. However, this will significantly increase the size of the Smali code to be analysed in $\mathbb{K}$ and consequently the state space of the obtained models. To this end, we directly implement the semantics of some of these classes and methods in $\mathbb{K}$ rules, based on their description[2]. While the first approach appears to be more faithful, it would significantly increase the size of the Smali code to be analysed in $\mathbb{K}$ and consequently the state space of the obtained models. This is avoided by the second approach where one can choose the abstraction level required for the analysis in question.

In Figure 13, we show the structure for $\mathbb{K}$ modules which implements the semantics of some API methods.

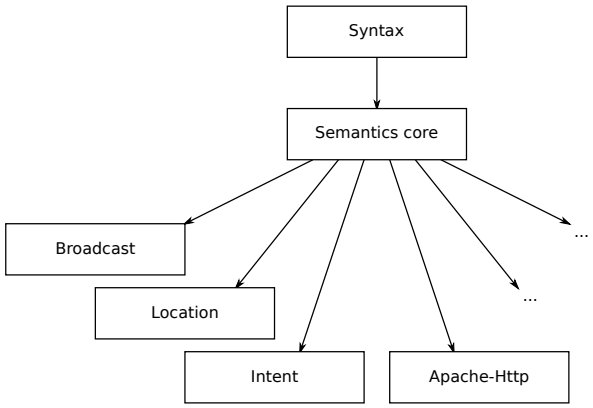

Fig. 13: Semantic module structure for Android API.

In particular, we have implemented a number of APIs, including modules Location, Intent, Broadcast, and Apache-Http. Other API classes and methods can be implemented similarly. For those modules that are not (yet) implemented in $\mathbb{K}$, we provide a mechanism that a call to any of them returns an unknown result, i.e., the "$\cdot K$" value.

A typical example is the Location module which is responsible for implementing the semantics of API methods relating to the Location Manager such as registering a callback function when the device's location changes, i.e., the `requestLocationUpdates` method from `LocationManager` class. When a registered callback method is called, it is provided with an input parameter referring to a `Location` object. The creator of this object is designated to the application in the current sandbox by the object's created cell. Furthermore, it is marked as sensitive in its sensitive cell (see Figure 10).

The module Intent is responsible for implementing the semantics of API methods for creating and manipulating intents such as the constructor of `Intent` class, adding extra string data into an intent, i.e., `putExtra` from `Intent` class and retrieving extra string data from an intent, i.e., `getStringExtra` method also from `Intent` class. The module Broadcast is responsible for implementing the semantics of API methods relating to broadcasting intents, for example: broadcasting an intent, i.e., `sendBroadcast` method from `Context` class; and registering an callback function when receiving an broadcasted intent, i.e., `registerReceiver` method from `Context` class. In particular, when `sendBroadcast` with an intent, this intent will be place in `broadcasts` cell (see Figure 9) in the configuration. Then, callback methods previously registered by a call to `registerReceiver` will be called according to the newly placed intent in `broadcasts` cell. Finally, the module `Apache-Http` implements the semantics of methods relating to sending http request, i.e. `execute` method from `HttpUriRequest` class.

5.3.4 Detecting collusion on the concrete semantics level:

Finally, we detect information theft via collusion by annotating any "object" cell with two additional values: "sensitive" and "created". *Sensitive* is a Boolean value indicating if the object is sensitive (e.g., device locations, contacts, private data, etc.). *Created* is a set of app ids that initialise the object. Information theft collusion is conducted when one app collects sensitive data from an Android device and forwards to another app who will export it outside the device boundaries. In detail, this process includes, within a device: (i) a sensitive object O_1 is initialised by an app A_1, i.e., *Sensitive* of O_1 is *true* and *Created* of O_1 contains id A_1; (ii) O_1 (or its content) is forwarded another app A_2 via communication (possibly through a series of actions of creating new objects or editing existing objects using O_1 where their *Sensitive* is updated to that of O_1 and their *Created* is updated to include A_1); (iii) A_2 calls an API to export O_1 (or any of these objects whose *Sensitive* is true and *Created* contains id $A_1 \neq A_2$). Information theft collusion is detected in our framework

when A_2 calls the API to export an object with *Sensitive* equal true and *Created* containing any id $A_1 \neq A_2$.

This characterisation of collusion as an information flow property implies the conditions of Definition 1:

- A_1 contributes in retrieving the sensitive data while A_2 is exporting.
- A_1 and A_2 communicate with each other to transfer the sensitive data from A_1 to A_2.

5.4 An abstract semantics for Dalvik

The abstract semantics lightens the configuration and the transitions in order to gain efficiency for model checking while maintaining enough information to verify collusion. The abstract configuration has a cell structure similar to the concrete configuration except for the memory cell: instead of creating objects, in abstract semantics we record the information flow by propagating the object types and the constants (either strings or numerical). Structurally, the $\mathbb{K}$ specification for the abstract semantics is organised in the same way as the concrete one, c.f. Figure 12. In the followings we describe the differences that render the abstraction.

In the "read/write" module the abstract semantics neglects the memory-related details as described next: The abstract semantics for the *instructions that create new object instances* (e.g., "`new-instance` *Register*, *Type*") sets the register to the type of the newly created object. The *arithmetic instructions* only define data dependency between the source registers and the destination register. The *move instruction*, that copies one register into another, sets the contents of the source register into the destination. Finally, the *load/store instructions*, that copy from or into the memory, are similarly abstracted into data-dependence. We exemplify this latest class of instructions with the abstract semantics of the `iget` instruction in Figure 14.

Fig. 14: $\mathbb{K}$ rule for the abstract semantics of `iget` instruction.

The abstract semantics is field insensitive, e.g., the `iget` instruction only maintains the information collected in the object register, R_2. In order to add field sensitivity to the abstraction, we only need to en-queue in R_1 the field F such that after the substitution we have $R_1 \mapsto F \curvearrowright L2$.

The module "invoke/return" contains the most significant differences of the abstract semantics w.r.t. the concrete semantics. The *invoke instructions* differentiate the API calls from the app's methods. The methods defined in the app are executed

upon invocation (using a call stack) while the API calls are further discriminated into app-communication (i.e., "send" or "receive"), APIs which trigger callbacks, APIs which access sensitive data, APIs which publish data, and ordinary APIs. We currently consider only Intent based inter-app communication. All invoke instructions add information to the data-flow as follows: the object for which the method is invoked depends on the parameters of the invoked method. Similarly, the *move-result instruction* defines data-dependence between the parameters of the latest invoked method and the register where the result is written. The data-flow abstraction allows us to see an API call just as an instruction producing additional data dependency. Hence, we do not need to treat separately these APIs as in the concrete semantics (by either executing their code or giving them special semantics). This gives a lightweight feature to the abstract semantics meanwhile enlarging the class of apps that can be verified. Obviously, the price paid is the over approximation of the app behaviours which induces false positive colluding results.

The rules producing transitions in the transition system are defined in the "control" module. The rules for *branching instructions*, i.e., `if-then` instructions, are always considered non-deterministic in the abstract semantics. The rules for `goto` instruction check if the goto destination was already traversed in the current execution and, if this is the case, the jump to the destination is replaced by a fall through. As such, the loops are traversed at most once since the data-flow collection only requires one loop traversal.

5.4.1 Detecting collusion on the abstract semantics level:

Detecting collusion on the abstract semantics level works as follows: When an API accessing sensitive data is invoked in an app A_1, the data-flow is augmented with special a label "$secret(A_1)$". If, via the data-flow abstraction, the "secret" arrives into the parameters of a publish invocation of a different app A_2 ($A_1 \neq A_2$) then we discover a collusion pattern for information theft. Note that the "secret" could be passed from A_1 to A_2 directly or via other apps A's.

The property detected in the abstract semantics is a safe over-approximation of Definition 1. Namely, (i) the set of colluding apps S includes two different apps A_1 and A_2, hence S is not a singleton set; (ii) the apps A_1 and A_2 execute the beginning and the end of the threat (i.e. the extraction and the publication of the secret, respectively) while the apps A's act as messengers; (iii) all the discovered apps contribute in communicating the secret.

Note, we say that the abstract collusion result is an over-approximation due to the fact that only "non-colluding" results could be a guarantee for the non-existence of a set S with the characteristics given by Definition 1. If a colluding set S is reported in the abstract model checking then this is either a true collusion, as argued in (i–iii), or a false witness to collusion. A false witness (also named "spurious counterexample" in abstract model checking) may appear due to the overprotective nature of the data-flow abstraction. This abstraction assumes that any data "touching" the secret may take it and pass it (e.g. when the secret is given as parameter to an API call f then any

other parameter and the result of f are assumed to know the secret). Consequently, any collusion set S reported by the abstract model checking has to be verified (e.g. by exercising the concrete semantics over S).

5.5 Experimental Results

We demonstrate how collusion is detected using our concrete and our abstract semantics on two Android applications, called `LocSender` and `LocReceiver`. Together, these two apps jointly carry out an "information theft".

They consist of about 100 lines of Java code / 3000 lines of Smali code each. Originally written to explore if collusion was actually possible (there is no APK of the Soundcomber example), here they serve as a test for our model checking approach.

`LocSender` obtains the location of the Android device and communicates it using a broadcast intent. `LocReceiver` constantly waits for such a broadcast. On receiving such message, it extracts the location information and finally sends it to the Internet as an HTTP request. We have two variants of `LocReceiver`: one contains a `while` loop pre-processing the HTTP request while the other does not. Additionally, we create two further versions of each `LocReceiver` variant where collusion is broken by (1) not sending the HTTP request at the end, (2) altering the name of the intent that it waits for – named `LocReceiver1` and `LocReceiver2`, respectively. Furthermore, we (3) create a `LocSender1` which sends a non-sensitive piece of information rather than the location. In total, we will have eight experiments where the two firsts have a collusion while the six lasts do not[8]. Figure 15 summarises the experimental results.

App1	App2	Loop	Collusion	Concrete		Abstract	
				Runtime	Detected	Runtime	Detected
`LocSender`	`LocReceiver`		✓	55s	✓	30s	✓
`LocSender`	`LocReceiver`	✓	✓	time-out		33s	✓
`LocSender`	`LocReceiver1`			1m13s		31.984s	
`LocSender`	`LocReceiver1`	✓		time-out		34s	
`LocSender`	`LocReceiver2`			53s		32s	
`LocSender`	`LocReceiver2`	✓		time-out		33s	
`LocSender1`	`LocReceiver`			1m11s		32s	
`LocSender1`	`LocReceiver`	✓		time-out		34s	

Fig. 15: Experimental result.

[8] All experiments are carried out on a Macbook Pro with an Intel i7 2.2GHz quad-core processor and 16GB of memory.

5.5.1 Evaluation

Our experiments indicate that our approach works correctly: if there is collusion it is either detected or has a timeout, if there is no collusion then none is detected. In case of detection, we obtain a trace providing evidence of a run leading to information theft. The experiments further demonstrate the need for an abstract semantics, beyond the obvious argument of speed: e.g. in case of a loop where the number of iterations depends on an environmental parameter that can't be determined, the concrete semantics yields a time out, while the abstract semantics still is able to produce a result. Model checking with the abstract semantics is about twice as fast as with the concrete semantics. At least for such small examples, our approach appears to be feasible.

6 Related work

In this Section we review the different previous works that have addressed the identification and prevention of Android malicious software. We first review previous approaches to detect and identify Android malware (single apps) in general. Then, we address previous work on detection and identification of colluding apps. Finally, we review works that focus on collusion prevention.

6.1 Detecting malicious applications

In general, techniques for detecting Android malware are categorised into two groups: static and dynamic. In static analysis, certain features of an app are extracted and analysed using different approaches such as machine learning techniques. For example, Kirin [26] proposes a set of policies which allows matching permissions requested by an app as an indication for potentially malicious behaviour. DREBIN [4] trained Support Vector Machines for classifying malwares using number of features: used hardware components, requested permissions, critical and suspicious API calls and network addresses. Similar static techniques can be found in [16, 19, 38, 44, 63]. Conversely, dynamic analysis detects malware at the run-time. It deploys suitable monitors on Android systems and constantly looks for malicious behaviours imposed by software within the system. For example, [33] keeps track of the network traffic (DNS and HTTP requests in particular) in an Android system as input and then utilises Naive Bayes Classifier in order to detect malicious behaviours. Similarly, [39] collects information about the usage of network (data sent and received), memory and CPU and then uses multivariate time-series techniques to decide if an app admitted malicious behaviours. A different approach to translate Android apps into formal specifications and then employing existing model checking techniques to explore all possible runs of the apps in order to search

for a matching malicious activity represented by formulas of some temporal logic can be found in [60, 10].

In contrast to malware detection, detecting colluding apps involves not only identifying whether a security threat can be carried out by these apps but also revealing whether communication between them occurs during the attack. In other words, existing malware detection techniques are not directly applicable for detecting collusion.

6.2 Detecting malicious inter-app communication

Current research mostly focuses on detecting inter-app communication and information leakage. DidFail [15] is a analysis tool for Android apps that detects possible information flows between multiple apps. Each APK is fed into the APK transformer, a tool that annotates intent-related function calls with information that uniquely identifies individual cases where intents are used in the app, and then transformed APK is passed to two other tools: FlowDroid [5, 28] and Epicc [49]. The FlowDroid tool performs static taint tracking in Android apps. That analysis is field, flow and context sensitive with some object sensitivity. Epicc performs static analysis to map out inter-component communication within an Android app. Epicc [49] provides flow and context sensitive analysis for app communication, but it does not tackle each and every possible communication channels between apps' components. The most similar work to DidFail is IccTA [42] which statically analyses app sets to detect flows of sensitive data. IccTA uses a single-phase approach that runs the full analysis monolithically, as opposed to DidFail's composition two-phase analysis. DidFail authors acknowledge the fact that IccTA is more precise than the current version of DidFail because of its greater context sensitivity. This supports our claim in section 4.2 - "context would be the key" for improving the precision. FUSE [52], a static information flow analysis tool for multi-apps, provides similar functions as DidFail and IccTA in addition to visualising inter-component communication (ICC) maps. DroidSafe [31] is a static information flow analysis tool to report potential leaks of sensitive information in Android applications.

ComDroid [17] detects app communication vulnerabilities. Automatic detection of inter-app permission leakage is provided [56]. Authors address three kinds of such attacks: confused deputy, permission collusion and intent spoofing and use taint analysis to detect them. An empirical evaluation of the robustness of ICC through fuzz testing can be found in [45]. A study of network covert channels on Android is [29, 30]. Authors show that covert channels can be successfully implemented in Android for data leakage. A security framework for Android to protect against confused deputy and collusion attacks is proposed [14]. The master thesis [53] provides an analysis of covert channels on mobile devices. COVERT [9] is a tool for compositional analysing inter-app vulnerabilities. TaintDroid [25], an information-flow tracking system, provides a real time analysis by leveraging Android's virtualized execution environment. DroidForce [51], build upon on FlowDroid, attempts to ad-

dresses app collusion problem with a dynamic enforcement mechanism backed by a flexible policy language. However static analysis encourages in collusion detection due the scalability and completeness issues [24]. Desired properties for a practical solution include, but not limited to: characterising the context associated with communication channels with fine granularity, minimising false alarms and ability to scalable for a large number of apps.

6.3 Other Approaches

Application collusion can also be mitigated by implementing compartmentalisation techniques, like Samsung Knox [1]. These techniques isolate app groups by forbidding any communication between apps in different groups. In [61] authors analyse several compartmentalisation strategies to minimise the risk of app collusion. Their results show that it is enough to have two or three app compartments to greatly reduce the risk posed by a set of 20 to 50 apps. In order to reduce the risk further, the amount of app compartments must be increased exponentially.

Finally, Bartel et al. [43] propose the tool *APKCombiner* which joins two apps into a single APK file. In this way, a security analyst can use inter-component, instead of inter-app, communication analysers to analyse the inter app communication mechanisms that exist between apps. From an evaluation set of 3000 apps they were able of joining 88% of them. The average time required to join two apps with APKCombiner is three minutes. This makes it hard to use for practical app analysis.

7 Conclusion and Future work

We summarise the state of the art w.r.t. collusion prevention and point the reader to the current open research questions in the field.

A frontal approach to detecting collusions to analyse pairs, triplets and larger sets is not practical given the search space. Thus, we consider the step of pre-filtering apps essential for a collusion detection system if it were to be used in practice. Even if we could find all collusions in all existing apps, new ones appear every day and they could create new collusions with previously analysed apps. Continuously re-analysing the growing space of all Android apps is infeasible so an effective collusion-discovery tool must include an effective set of methods to isolate potential sets which require further examination.

The best long-term solution would be to enforce more isolation in the Android OS itself. For example, apps may be required to explicitly declare all communications (this includes not only inter-app channels but also declaring all Internet domains, ports and services which they intend to use) via their manifests and then the OS will be able to block all other undeclared communications. However, this will not work for already existing apps (as well as many apps which could be created

before such OS hardening were implemented) so in the meantime the best practical approach is to employ, enhance and expand the array of filtering mechanisms we developed to discover potentially colluding sets of apps.

A filter based on Android app permissions is the simplest one. Permissions are very easy and cheap to extract from APKs – no de-compilation, reverse engineering, complex code or data flow analysis is required. Alternatively (or additionally), to the two filters described in our chapter, imprecise heuristic methods to find "interesting" app sets may include: statistical code analysis of apps (e.g. to locate APIs potentially responsible to communication, accessing sensitive information, etc.); and taking into account apps' publication time and distribution channel (app market, direct installation, etc.).

Attackers are more likely to release colluding apps in a relatively short time frame and that they are likely to engineer the distribution in such a way that sufficient number of users would install the whole set (likely from the same app market). To discover such scenarios one can employ: analysis of security telemetry focused on users devices to examine installation/removal of apps, list of processes simultaneously executing, device-specific APK download/installation logs from app markets (like Google Play™) and meta-data about APKs in app markets (upload time by developers, developer ID, source IP, etc.). Such data would allow constructing a full view of existing app sets on user devices. Only naturally occurring sets (either installed on same device or actually executing simultaneously) may be analysed for collusion which should drastically reduce the number of sets that require deeper analysis.

Naturally, finding "interesting" app sets is not enough: in the end, some analysis is required to figure out if a given set of apps colludes. Manual analysis is costly, merging apps into a single one often fails, however software model checking of suitable abstractions of an app set might be a way forward. We demonstrated that both semantic approaches are – in principle – able to successfully model check for app collusion realising the threat of information theft. Here, naturally the abstract semantics outperforms the concrete one. Though it is still early days, we dare to express the following expectation: we believe that our approach will scale thanks to its powerful built-in abstraction mechanisms.

The aspiration set out in this chapter is to build a fully automated and effective collusion detection system, and tool performance will be central to address scale. It is not clear yet where the bottleneck will be when we apply our approach to real-life apps in a fully operational deployment. Further work will focus on identifying these bottlenecks to optimise the slowest elements of our tool-chain. Detecting covert channels would be a challenge as modelling such will not be trivial; this is the natural next step.

In the long run, collusions are a part of a general problem of effective isolation of software. This problem exists in all environments which implement sandboxing of software –from other mobile operating systems (like iOS and Tizen) to virtual machines in server farms (like Amazon EC2, Microsoft Azure and similar). We can see how covert communications between sandboxes may be used to breach security and create data leaks. The tendency to have more and better isolation is, of course,

a positive one but we should fully expect the attackers to employ collusion methods more often to circumvent security. We endeavour to see if our methods developed for Android would be applicable to a wider range of operating environments.

References

1. (2016). URL https://www.samsungknox.com/
2. Android Package Index. http://developer.android.com/reference/packages.html (2016)
3. Android Open Source Project: Dalvik Bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html (2016)
4. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: 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). URL http://www.internetsociety.org/doc/drebin-effective-and-explainable-detection-android-malware-your-pocket
5. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: ACM SIGPLAN Notices - PLDI'14, vol. 49, pp. 259–269. ACM (2014)
6. Asavoae, I.M., Blasco, J., Chen, T.M., Kalutarage, H.K., Muttik, I., Nguyen, H.N., Roggenbach, M., Shaikh, S.A.: Towards Automated Android App Collusion Detection. CoRR **abs/1603.02308** (2016). URL http://arxiv.org/abs/1603.02308
7. Asavoae, I.M., Muttik, I., Roggenbach, M.: Android malware: They divide, we conquer. Bucharest, Romania (2016)
8. Asavoae, I.M., Nguyen, H.N., Roggenbach, M., Shaikh, S.A.: Utilising $\mathbb{K}$ Semantics for Collusion Detection in Android Applications. In: Critical Systems: Formal Methods and Automated Verification - Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings, pp. 142–149 (2016). DOI 10.1007/978-3-319-45943-$1_1$0
9. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: Covert: Compositional analysis of android inter-app vulnerabilities. Tech. rep., Tech. Rep. GMU-CS-TR-2015-1, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA (2015)
10. Beaucamps, P., Gnaedig, I., Marion, J.: Abstraction-based malware analysis using rewriting and model checking. In: S. Foresti, M. Yung, F. Martinelli (eds.) Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7459, pp. 806–823. Springer (2012). DOI 10.1007/978-3-642-33167-1_46. URL http://dx.doi.org/10.1007/978-3-642-33167-1_46
11. Blasco, J., Chen, T., Muttik, I., Roggenbach, M.: Wild android collusions. Virus Bulletin 2016 (2016)
12. Blasco, J., Chen, T.M., Muttik, I., Roggenbach, M.: Efficient Detection of App Collusion Potential Using Logic Programming. Submitted to IEEE Transactions on Mobile Computing (2016)
13. Blasco, J., Muttik, I.: Android collusion conspiracy (2015)
14. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: NDSS (2012)

15. Burket, J., Flynn, L., Klieber, W., Lim, J., Snavely, W.: Making didfail succeed: Enhancing the cert static taint analyzer for android app sets. Tech. Rep. MSU-CSE-00-2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh,USA (2015)
16. Canfora, G., Lorenzo, A.D., Medvet, E., Mercaldo, F., Visaggio, C.A.: Effectiveness of opcode ngrams for detection of multi family android malware. In: 10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015, pp. 333–340 (2015). DOI 10.1109/ARES.2015.57. URL http://dx.doi.org/10.1109/ARES.2015.57
17. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys'11, pp. 239–252 (2011)
18. Clavel, M., Duran, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Talcott, C.: All about Maude. LNCS **4350** (2007)
19. Dai, G., Ge, J., Cai, M., Xu, D., Li, W.: SVM-based malware detection for android applications. In: Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015, pp. 33:1–33:2 (2015). DOI 10.1145/2766498.2774991. URL http://doi.acm.org/10.1145/2766498.2774991
20. Desnos, A.: Androguard. https://github.com/androguard/androguard (2016)
21. Dubey, A., Misra, A.: Android Security: Attacks and Defenses. CRC Press (2013)
22. Elenkov, K.: Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press (2014)
23. Elish, K.O., Yao, D., Ryder, B.G.: On the need of precise inter-app ICC classification for detecting Android malware collusions. In: MoST (2015)
24. Elish, K.O., Yao, D.D., Ryder, B.G.: On the need of precise inter-app icc classification for detecting android malware collusions. In: Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy (2015)
25. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) **32**(2), 5 (2014)
26. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on Computer and communications security, pp. 235–245. ACM (2009)
27. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE security & privacy (1), 50–57 (2009)
28. Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., le Traon, Y., Octeau, D., McDaniel, P.: Highly precise taint analysis for android applications. EC SPRIDE, TU Darmstadt, Tech. Rep (2013)
29. Gasior, W., Yang, L.: Network covert channels on the android platform. In: Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research, p. 61. ACM (2011)
30. Gasior, W., Yang, L.: Exploring covert channel in android platform. In: Cyber Security (CyberSecurity), 2012 International Conference on, pp. 173–177 (2012). DOI 10.1109/CyberSecurity.2012.29
31. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS (2015)
32. Gunasekera, S.: Android Apps Security. Apress (2012)
33. Han, H., Chen, Z., Yan, Q., Peng, L., Zhang, L.: A real-time android malware detection system based on network traffic analysis. In: Algorithms and Architectures for Parallel Processing - 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015. Proceedings, Part III, pp. 504–516 (2015). DOI 10.1007/978-3-319-27137-8_37. URL http://dx.doi.org/10.1007/978-3-319-27137-8_37
34. Hardy, N.: The confused deputy:(or why capabilities might have been invented). ACM SIGOPS Operating Systems Review **22**(4), 36–38 (1988)
35. Harley, D., Lee, A.: Antimalware evaluation and testing. In: AVIEN Malware Defense Guide. Elsevier (2007)

36. Huskamp, J.C.: Covert communication channels in timesharing systems. Ph.D. thesis, California Univ., Berkeley (1978)
37. Kalutarage, H.K., Nguyen, H.N., Shaikh, S.A.: Towards a threat assessment framework for apps collusion. Telecommunication Systems pp. 1–14 (2017). DOI 10.1007/s11235-017-0296-1. URL http://dx.doi.org/10.1007/s11235-017-0296-1
38. Kate, P.M., Dhavale, S.V.: Two phase static analysis technique for android malware detection. In: Proceedings of the Third International Symposium on Women in Computing and Informatics, WCI 2015, co-located with ICACCI 2015, Kochi, India, August 10-13, 2015, pp. 650–655 (2015). DOI 10.1145/2791405.2791558. URL http://doi.acm.org/10.1145/2791405.2791558
39. Kim, K., Choi, M.: Android malware detection using multivariate time-series technique. In: 17th Asia-Pacific Network Operations and Management Symposium, APNOMS 2015, Busan, South Korea, August 19-21, 2015, pp. 198–202 (2015). DOI 10.1109/APNOMS.2015.7275426. URL http://dx.doi.org/10.1109/APNOMS.2015.7275426
40. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, pp. 1–6. ACM (2014)
41. Krishnamoorthy, K.: Handbook of statistical distributions with applications. CRC Press (2015)
42. Li, L., Bartel, A., Bissyand, T., Klein, J., Le Traon, Y., Arzt, S., Siegfried, R., Bodden, E., Octeau, D., Mcdaniel, P.: IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In: Proceedings of the 37th International Conference on Software Engineering (ICSE 2015) (2015)
43. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y.: ApkCombiner: Combining multiple Android apps to support inter-app analysis. In: SEC'15, pp. 513–527. Springer (2015)
44. Li, Q., Li, X.: Android malware detection based on static analysis of characteristic tree. In: 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2015, Xi'an, China, September 17-19, 2015, pp. 84–91 (2015). DOI 10.1109/CyberC.2015.88. URL http://dx.doi.org/10.1109/CyberC.2015.88
45. Maji, A.K., Arshad, F., Bagchi, S., Rellermeyer, J.S., et al.: An empirical study of the robustness of inter-component communication in android. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on, pp. 1–12. IEEE (2012)
46. Marforio, C., Francillon, A., Capkun, S.: Application collusion attack on the permission-based security model and its implications for modern smartphone systems. technical report (2011)
47. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51–60. ACM (2012)
48. Muttik, I.: Partners in crime: Investigating mobile app collusion. In: McAfee Threat Report (2016)
49. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In: USENIX Security 2013 (2013)
50. Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Using probabilistic generative models for ranking risks of android apps. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 241–252. ACM (2012)
51. Rasthofer, S., Arzt, S., Lovat, E., Bodden, E.: Droidforce: enforcing complex, data-centric, system-wide policies in android. In: Availability, Reliability and Security (ARES), 2014 Ninth International Conference on, pp. 40–49. IEEE (2014)
52. Ravitch, T., Creswick, E.R., Tomb, A., Foltzer, A., Elliott, T., Casburn, L.: Multi-app security analysis with fuse: Statically detecting android app collusion. In: Proceedings of the 4th Program Protection and Reverse Engineering Workshop, p. 4. ACM (2014)
53. Ritzdorf, H.: Analyzing covert channels on mobile devices. Ph.D. thesis, ETH Zürich, Department of Computer Science (2012)
54. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010)

55. Sarma, B.P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Android permissions: a perspective combining risks and benefits. In: Proceedings of the 17th ACM symposium on Access Control Models and Technologies, pp. 13–22. ACM (2012)
56. Sbirlea, D., Burke, M., Guarnieri, S., Pistoia, M., Sarkar, V.: Automatic detection of inter-application permission leaks in android applications. IBM Journal of Research and Development **57**(6), 10:1–10:12 (2013). DOI 10.1147/JRD.2013.2284403
57. Schlegel, R., Zhang, K., Zhou, X.y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: NDSS'11, pp. 17–33 (2011)
58. Shen, S.: 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
59. Six, J.: Application Security for the Android Platform: Processes, Permissions, and Other Safeguards. O'Reilly (2011)
60. Song, F., Touili, T.: Model-checking for android malware detection. In: J. Garrigue (ed.) Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings, *Lecture Notes in Computer Science*, vol. 8858, pp. 216–235. Springer (2014). DOI 10.1007/978-3-319-12736-1_12. URL http://dx.doi.org/10.1007/978-3-319-12736-1_12
61. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P.: Compartmentation policies for Android apps: A combinatorial optimization approach. In: Network and System Security, pp. 63–77 (2015)
62. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Ribagorda, A.: Evolution, detection and analysis of malware for smart devices. Comm. Surveys & Tutorials, IEEE **16**(2), 961–987 (2014)
63. Wang, Z., Li, C., Guan, Y., Xue, Y.: Droidchain: A novel malware detection method for android based on behavior chain. In: 2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015, pp. 727–728 (2015). DOI 10.1109/CNS.2015.7346906. URL http://dx.doi.org/10.1109/CNS.2015.7346906