Towards Automated Android App Collusion Detection

Irina Mariuca Asăvoae[1], Jorge Blasco[2], Thomas M. Chen[2], Harsha Kumara Kalutarage[3], Igor Muttik[4], Hoang Nga Nguyen[5], Markus Roggenbach *[1], and Siraj Ahmed Shaikh[5]

[1] *Swansea University, UK*
[2] *City University London, UK*
[3] *Queen's University of Belfast, UK*
[4] *Intel Security, UK*
[5] *Coventry University, UK*

Abstract

Android OS supports multiple communication methods between apps. This opens the possibility to carry out threats in a collaborative fashion, c.f. the Soundcomber example from 2011. In this paper we provide a concise definition of collusion and report on a number of automated detection approaches, developed in co-operation with Intel Security.

1 Introduction

The Android operating system (OS) is designed with a number of built-in security features such as app sandboxing and fairly granular access controls based on permissions. In real life, however, the isolation of apps is limited. In some respects even the opposite is true – there are many ways for apps to communicate across sandbox boundaries. The Android OS supports multiple communication methods which are fully documented (such as messaging via intents). The ability of apps with different security postures to communicate has a negative effect on security as an app (in a sandbox which has permissions to handle such data) is allowed to let sensitive data flow to another app (in another sandbox which has been denied permission to handle such data) and eventually leak out.

The Android ecosystem exacerbates the problem as the market pressure leads many developers to embed advertisement libraries into their apps. As a result, such code may be present in thousands of apps (all bearing different permissions). Advertisers have a known tendency to disregard user privacy in favour of monetisation. So there exists a risk that ad-libraries may communicate between sandboxes even without knowledge of apps authors to transmit sensitive data across, risking exposure and disregarding privacy.

Of course, any unscrupulous developer may also split functionality which they prefer to hide between multiple apps. Malicious behaviours similar to this are evident from known cases of apps exploiting insecure exposure of sensitive data by other apps [2].

Researchers have demonstrated that sets of apps may violate the permissions model causing data leaks or carrying malware [23]. Such apps are called *colluding sets of apps* and the phenomenon is called *app collusion*. Unfortunately, there are no effective tools to detect app collusion. The search space posed by possible combination of apps means that this is not straightforward. Effective methods are needed to narrow down the search to collusion candidates of interest.

In recent work [13], dating after submission of this paper, we have discovered that app collusion was actually being used in the field for quite a long time and without being detected in a large group of applications which use a popular Android SDK. This SDK is known to be included in more than a 1000 applications. This discovery was a result of applying techniques described in this paper to a large set of the most popular apps.

This paper contributes towards a practical automated system for collusion detection. We give a def-

* Corresponding author M.Roggenbach@swan.ac.uk

Copyright © by the paper's authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

In: D. Aspinall, L. Cavallaro, M. N. Seghir, M. Volkamer (eds.): Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS'16, London, UK, 06-April-2016, published at http://ceur-ws.org

inition of collusion in Section 2. This is followed by two potential approaches to filter down to potential candidates for collusion, using a rule based approach developed in Prolog in Section 3 and another statistical approach in Section 4. Section 5 presents a model-checking approach to detecting collusion in Android apps. Section 6 delves into the experimental outcomes and Section 7 discusses related work. Section 8 summarises our contributions and Section 9 concludes the paper with thoughts on future work.

2 Defining collusion

Our notion of collusion refers to the ability for a set of apps to carry out a threat in a collaborative fashion. This is in contrast to most existing work, where collusion is usually associated with inter-app communications and information leakage (see Section 7). We consider that colluding apps can carry out any threat such as the ones posed by single apps. The range of such threats includes [25]:

- Information theft: happens when one app accesses sensitive information and the other sends information outside the device boundaries.
- Money theft: happens when an app sends information to another app that is capable of using money sensitive API calls (e.g. SMS).
- Service misuse: happens when one app is able to control a system service and receives information (commands) from another app to control those services.

A threat can be described by a set of actions executed in a certain order. We model this by a partially ordered set $(T, \leq)$, where T is the set of actions and $\leq$ specifies the execution order. When $(T, \leq)$ is carried out, actions from T are sequentially executed, according to some total order $\leq^*$ such that $\leq \subseteq \leq^*$; in other words, $(T, \leq^*)$ is a total extension of $(T, \leq)$. Let $Ex((T, \leq))$ denote the set of all possible total extensions of $(T, \leq)$; i.e., all possible ways of carrying out threat $(T, \leq)$. Similarly, we also define inter-app communication as a partially ordered set. In this paper we discuss overt communications channels only.

We define collusion as follows:

A1: Actions are operations provided by Android API (such as record audio, access file, write file, send data, etc.). Let Act denote the set of all actions.

A2: Actions can be associated with a number of attributes (such as permissions, input parameters, etc.). Let B denote the set of all action attributes and $pms : Act \to \wp(B)$ specify the set of permissions required by Android to execute an action.

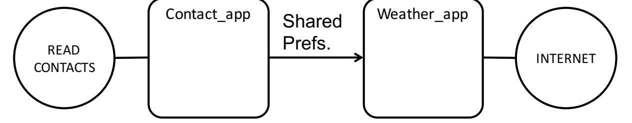

Figure 1: An example of colluding apps

A3: A threat $t = (T, \leq)$ is a partially ordered set. Let τ denote the set of all threats. In the scope of this paper, τ represents the set of all known threats caused by single applications.

A4: An inter-app communication $c = (C, \leq)$ is a partially ordered set. Let com denote the set of all known inter-app communications.

Definition 1 (Collusion)**.** *A set S consisting of at least two apps is* colluding *if together they execute a sequence $A \in Act^*$ such that:*

1. *there exists a subsequence A' of A such that $A' \in Ex(t)$ for some $t \in \tau$; furthermore, A' is collectively executed involving all apps in S, i.e., each app in S executes at least one action in A'; and*
2. *there exists a subsequence C' of A such that $C' \in Ex(c)$ for some $c \in com$.*

It is a general challenge to distinguish between benign and malicious application behaviours. Thus, our definition makes the realistic assumption that there are a number of known threats, e.g., Intel Security regularly identifies apps to be malicious, realising known threats. Now, collusion can be seen as a camouflage mechanism: the individual apps appear harmless, none of them alone poses a threat, e.g., they would not be able to execute a threat just in terms of their permissions; in combinatation, however, they realise a threat.

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

Example 1 (Stealing contact data)**.** *The two apps graphically represented in Figure 1 perform information theft: the* `Contact_app` *reads the contacts database to pass the data to* `Weather_app`*, which sends the data to a remote server controlled by the adversary. The information is sent through shared preferences.*

Using the collusion definition we can describe the actions performed by both apps as: $Act_{\texttt{Contact_app}} = \{a_{read_contacts}\}$, $Act_{\texttt{Weather_app}} = \{a_{send_file}\}$. *with* $pms(a_{read_contacts}) = \{Permission_contacts\}$ *and* $pms(a_{send_file}) = \{Permission_internet\}$. *The information threat t is given by* $T = \{a_{read_contacts}, a_{send_file}\}$ *and defining* $a_{read_contacts} \leq a_{send_file}$. *The inter-app communication is defined as* $com_{\texttt{Contact_app}} = \{send_{shared_prefs}\}$, $com_{\texttt{Weather_app}} = \{recv_{shared_prefs}\}$ *and* $send_{shared_prefs} \leq recv_{shared_prefs}$.

[1]Concrete examples are available on request.

3 Detecting collusion threat

Our first approximation to detect app collusion utilises Logic Programming in Prolog. Its goal 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; together with a further sifting mechanism (not discussed here) 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 [9] 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.

3.1 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, controlling 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}. Each permission is mapped to one or more of the four high level actions. For example, an app that declares the INTERNET permission will be capable of sending information outside the device:

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

3.2 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:

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

3.3 Threats

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. For example, the information theft threat is codified as the following Prolog rule:

$$\begin{array}{ll} & sensitive_information(App_a) \\ \wedge & information_outside(App_b) \\ \wedge & communicate(App_a, App_b, channel) \\ \rightarrow & collusion(App_a, App_b) \end{array}$$

Currently, we do not take into account the order of action execution.

4 Assessing the collusion possibility

In this section, we apply machine learning to classify app sets into colluding and non-colluding ones. To this end, we first define a probabilistic model. Then we train the model, i.e., estimate the model parameters on a training data set. As a third step we validate the model using a validation data set. Additionally, in Section 6.3, we check the model with testing data.

4.1 Probabilistic Model

Estimating the collusion possibility within a set S of apps involves to estimate two different likelihood components L_τ and L_{com}. L_τ denotes the likelihood of carrying out a threat. L_{com} denotes the likelihood of performing some inter-app communication. Hence, the likelihood of colluding within S is given by $L_\tau \times L_{com}$.

In order to estimate L_τ, we employ a so-called Naive Bayesian informative [18] model. We consider a multivariate random variable $Y = (y_1, \ldots, y_k)$. Here, k is the total number of permissions in Android OS, and $y_j \in \{0,1\}$ are independent Bernoulli random variables. Variable y_j is 1 if permission j is found in S, 0 otherwise. $P(Y)$ stands for the probability of obtaining S with permissions as described by Y. Our probabilistic model is then given by the equation:

$$P(Y) = \prod_{j=1}^{k} \lambda_j^{y_j}(1-\lambda_j)^{1-y_j}$$

where $\lambda_j \in [0,1]$ is the Bernoulli parameter.

To compute L_τ we define $L_\tau = ln\{(P(Y))^{-1}\}$. This reflects on the likelihood of an app set to "being malicious" increasing as the number of permissions requested increases. Therefore, any monotonically decreasing function can be chosen [18]. For comparisons we average out L_τ and scale down to the range [0,1].

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 $P(Y)$ – we estimate λ_j using informative beta prior distributions [14] and define the maximum posterior estimation

$$\hat{\lambda_j} = \frac{\sum y_j + \alpha_j}{N + \alpha_j + \beta_j} \quad (1)$$

where N is the number of apps in the training set and α_j, β_j are penalty effects. In this work we set $\alpha_j = 1$. β_j depend on the critical level of permissions as given in [22, 18]. β_j can take either the value $2N$ (most critical), N (critical) or 1 (non-critical).

We consider L_{com} to be a binary function such that $L_{com} \in \{1,0\}$ which takes the value 1 if there is any inter app communication within S, 0 otherwise.

4.2 Model Training

Up to our recent discovery [13], there were no known colluding examples in the wild. However, there are sets of apps available, where individual apps have been classified by industry experts. Thus, we have utilised one such set provided by Intel Security, which consists of 9k+ malicious and 9k+ clean apps. As mentioned in Section 2, there is no evidence suggesting differences between threats caused by single apps and colluding apps. Thus, we can estimate L_τ based on these two sets[2]. As for L_{com}, there is no need to estimate any constants.

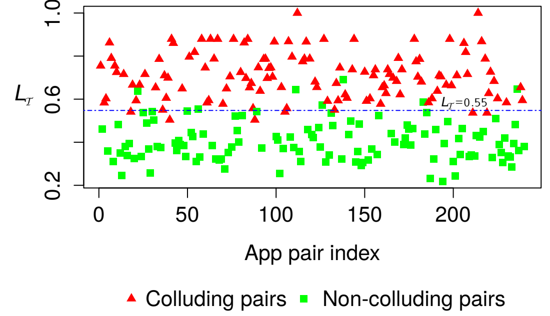

Figure 2: Distribution of L_τ over evaluation sample. Blue dotted line: best possible linear discriminant line.

4.3 Model Validation

We validate our model on a larger set of apps (a blind sample) that we created for the purpose of calibration of our collusion detection mechanisms. This validation set consists of 240 app pairs in which half are colluding pairs, the other half is non-colluding.

For this analysis, we implemented an automated process using R [19] and Bash scripts, which also includes calls to a third party research prototype [5] to find intent based communications. Additionally, a simple permission based rule set was defined to find communication using external storage. Overall, this process consists of the following steps: (1) extracting permissions of all single apps in a given app set S; (2) computing L_τ using extracted permissions; (3) if L_τ is greater than a certain threshold then estimating L_{com} as mentioned above; and finally (4) computing $L_\tau \times L_{com}$.

Figure 2 presents L_τ values for the validation dataset in which good visual separation can be seen between two classes with a lower (=0.50) and upper (=0.69) bounds for a discriminant line. Table 1 presents the confusion matrix obtained by fitting the best possible linear discriminant line at $L_\tau = 0.55$ in Figure 2.

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

Table 1: Confusion matrix for the evaluation sample.

Performance measures precision(=0.94) and F-measure(=0.95) were computed using the Table 1.

[2] Though our attack model is multiple apps, L_τ focuses only on operations required to execute a threat by a single app attack model. Additional communication operations required to execute the same threat in colluding model are covered by L_{com}.

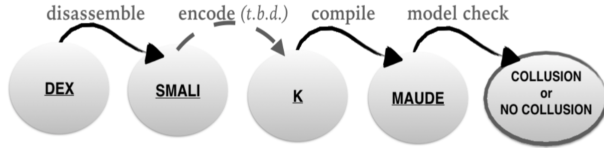

Figure 3: Work-flow

Precision quantifies the notion of specificity while F-measure provides a way to judge the overall performance of the model for each class. The higher these values are, the better the performance is. Such values could be due to a bias of the validation sample towards the methodology. Further investigation on the bias of the evaluation dataset is left for future work.

5 Model-Checking for collusion

We demonstrate the effective attempt to detect collusion via model-checking. Figure 3 shows the basic work-flow employed: it starts with a set of apps in the Dalvik Executable format DEX [1]; this code is disassembled and (currently manually) translated into a semantically faithful representation in the framework K [20] – this step also triggers a data flow analysis of the app code; compilation translates the K representation into a rewrite theory in Maude [7]; using the Maude model-checker provides an answer if the app set colludes or not: in case of collusion, the tool provides a readable counter example trace (see Section 6.4).

The rewriting logic semantics framework K allows the user to define *configurations*, *computations* and *rules* [20]. *Configurations* organise the program state in units called cells. In case of SMALI assembly programs, on the method level program states essentially consist of the current instruction, registers and parameters, and the class it belongs to. The operational semantics of, say, the assembly instruction

`const` *Register, Int*

for loading an integer constant *Int* into a register *Register* is captured by a *rule*

which reads: provided the current instruction is to load an integer I into a register R, then the cell "*regs*" capturing the state of the registers is updated by binding the value I to register R. The SMALI language also includes procedure calls. These allow to access functionality provided by the Android operating system, e.g., to read protected resources such as the GPS location or to send/receive a broadcast intent. Thus, besides encoding SMALI instructions in K, we also model the infrastructure that Android provides to apps.

The executions of the above semantics, instruction after instruction, defines the *computations.* However, the concrete semantics is far too detailed for effective model-checking. Therefore, we implement an abstraction in form of a data flow analysis. Here we represent each method individually as a graph, where registers, types, and constants are the nodes, and there is an edge between two nodes n_1 and n_2 iff n_2 is a parameter of the SMALI command that stores a value in n_1. For example, the SMALI command `const` *r1, 42* would lead to an edge from node "42" to node "*R1*". Analysing such graphs allows to detect which commands influence the values sent, say, in a broadcast intent, or publishing via the internet; these commands can be grouped into blocks and – rather than computing with concrete values – their effect can be captured symbolically.

For the detection of collusion, our K semantics carries a "trace" cell that records which operations provided by the Android API – see Definition 1 – have been executed in a specific run. Based on the "trace" cell we define collusion via a K rule: provided that the GPS location has been accessed, this value has been sent in a broadcast, this value has been received from a broadcast, and finally this value has been published, in that case we detect collusion "information theft".

Practical experiments with small apps demonstrate that this approach is feasible. Using the Maude model-checker, the state space of the (abstraction) of two apps is small, with only 8 states for the given example and the check takes less than a second. When modifying the apps in such a way that the information flow is broken, by, say using a different name for the broadcast and thus disabling communication between the apps, model-checking shows that no collusion is happening.

We use symbolic analysis over the byte code of the set S in order to obtain an in-depth inspection of the communication patterns. In the byte code of each app in S we detect the flow of communication with another app and a safe over-approximation of the data being communicated. We use static analysis over the byte code of apps to extract communication-flow between apps and data-flow inside each app. The data-flow information is filtered based on the private data being read and communicated.

It is future work, to complete the encoding to cover the full DEX language and to experiment with larger apps all well as with larger number of apps to address the fundamental question of scalability. We further intend to prove the abstraction to be sound, to show that if model-checking the abstracted code detects collusion then there is collusion in the original code, and to have a more general definition of collusion.

6 First experimental results

In this section we compile first experimental results obtained by applying the three methods discussed above to an artificial set.

6.1 An artificial colluding app set

In order to validate our different approaches against a known ground truth, we have created fourteen apps that cover all threats and use all communication channels described earlier in this paper. 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.

6.2 Detecting collusion threat with Prolog

Table 2 shows the results obtained with our rule based approached from Section 3 on the apps from Section 6.1. 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.The approach identifies all colluding app sets. It also flags

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

Table 2: Collusion Matrix of the Prolog program. ♣ = Information theft. $ = Money theft. ♠ = Service misuse. ♣, $, ♠ (light) = False positives.

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.

6.3 Computing collusion possibility

Table 3 shows results from our alternative approach from Section 4 on the same set of apps. Each cell denotes the L_τ value for the corresponding pair. To minimise false negatives, we use the lower bound (=0.50) gained from the validation data set 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. This yields symmetric data – for readability we leave the lower half of the matrix empty. Dark red shows true positives, light red shows false positives, dark green shows true negatives, and light green shows false negatives.

This approach finds two of the four colluding app sets and flags five false positives. It relies on a 3rd party tool to detect inter-app communication which ignores communication using shared preferences, thus the app set $\{id\ 1,\ id\ 2\}$ is not detected. As we restrict ourselves to pairwise analysis only, the app set $\{id\ 7,\ id\ 9\}$ can't be detected, as it communicates via *id 8*. Finally, app set $\{id\ 12,\ id\ 13\}$ was not reported since its L_τ value is less than the chosen threshold. Choosing a lower threshold could avoid this false negative, but at the cost of a lower class accuracy and performance.

A precise estimation of L_{com} would be useful in order to reduce the number of false positives in our analysis. However, communication is only a necessary, not a sufficient condition for collusion. A recent study [10] shows that 84.4% of market available apps 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

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														

Table 3: Matrix for collusion possibility.

our model. Figure 2 supports this claim as it present L_τ for both classes.

Both approaches to detect the potential of collusion are constrained in terms of the type of inter-app communication channels they account for due to reasons explained previously. This makes it difficult to provide for a straightforward comparison. The rule-based approach is not limited to pairs of set for collusion (which may involve more than two apps). It also allows us assess for the direction of the colluding behaviour (for cases of information flow). Defining rules requires expert knowledge and for some cases explicit rules may not exist; this is overcome by providing for rules to over-approximate for potentially colluding behaviour.

A statistical approach, on the other hand, has the advantage that it can reflect on varying degrees of possible collusion for a given set of apps. In this paper we have largely focused on static attributes, such as permissions, a number of other similar static attributes (such as developer ID, download metrics, and so on) may also be placed on a scale of suspicion for collusion; albeit the current implementation of the statistical approach in this paper is limited due to tool availability.

6.4 Software model-checking

We inspect two sets of applications, one colluding {*id 12*, *id 13*} and another non-clouding pair {*id 12*, *id 14*} with software model checking as described in Section 5. The sender *id 12* and receiver (*id 13* or *id 14*) communicate via broadcast messages containing the details of the GPS location. In the colluding case the broadcast is successful, i.e., the sender reads the GPS location then broadcasts it, and the receiver gets the broadcast then publishes the GPS location on the internet. In the non-colluding case, the sender still broadcasts the private data which reaches the receiver but this data is never published. Instead, the receiver publishes something else. The data-flow analysis shows, for the non-colluding case, that the private data is received but is never published. This aspect is not obvious for our Prolog filter. Similarly, the data-flow analysis detects collusion in the first case and reports its path witnesses, such as:

```
<trace>  call(readSecret, p1)
-> r1 := callRet(readSecret)
-> call(getBroadcast,r1,r1,"locsteal",p1)
-> call(sendBroadcast,"locsteal",r1)
-> r2 := callRet(getBroadcast)
-> call(publish,r2)
</trace>
```

7 Related work

App collusion can be traced back to *confused deputy* attacks [12]. They happen in form of *permission re-delegation attacks* [11, 8, 26] when a permission is carelessly exposed through a public component. Soundcomber [23] is an example where extracted information is transmitted using both overt and covert channels.

Marforio et al. [17] define colluding applications as those applications that cooperate in a violation of some security property of the system. Another definition is given by [16] where multiple apps can come together to perform a certain task, which is out of their permission capabilities. The malicious component of collusion is also acknowledged by Bagheri, Sadeghi et al. [21]. A more detailed definition is given by Elish [10], which defines it as the collaboration between malicious apps, likely written by the same adversary, to obtain a set of permissions to perform attacks.

ComDroid [6] is a static analysis tool that looks for confused deputies through *Intents*. XManDroid [3] and TrustDroid [4] extend the Android OS. Both allow for fine-grained policies that control inter-app information exchange; none of them address covert channels. In [24] authors analyze, using different risk metrics, several compartmentalisation strategies to minimise the risk of app collusion, showing two or three app compartments drastically reduce the risk of collusion for a set of 20 to 50 apps.

8 Summary

We have presented a new and concise definition of collusion in the context of Android OS. Colluding apps may carry out information theft, money theft, or service misuse; to this end, malware is distributed over multiple apps. As demonstrated by Soundcomber (and our own app sets), collusion is a possibility – which we recently demonstrated to exists in the wild [13]. Together with our industrial partner Intel Security, we have developed a number of approaches towards effectively detecting collusion. Early experimental results on small app sets (of a size nearly the scale of the number of apps one has on a phone) look promising: a combination of the rule-based approach and the machine learning approach could serve as a filter, after which we employ model checking as a decision procedure.

9 Future work

Like in [13], we will continue using our tools for analysing app sets in the wild. However, it should be noted that a frontal attack on detecting collusions to analyse analysing pairs, triplets and 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.

We have to emphasise that such tools are essential for many collusion-detection methods. Besides our model checking approach, one could, for example, think of the approach to merge pair of apps into a single app to inspect aggregated data flows [15]. Merging is slow and therefore app-combining approach is predicated on effective filtering of app pairs of interest.

Alternatively (or additionally), to the two filters described in our paper, 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^{TM}) 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.

Our goal 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. 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.

Acknowledgement

This work has been funded by EPSRC and we are excited to work on this challenging piece of research[3]. A special thanks goes to Erwin R. Catesbeiana (Jr) for excellent guidance through the Android ecosystem.

References

[1] Android Open Source Project. Dalvik Bytecode. `https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`, 2016.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Notices - PLDI'14*, volume 49, pages 259–269. ACM, 2014.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. *TU Darmstadt, TR-2011-04*, 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *SPSM'11*, pages 51–62. ACM, 2011.

[5] J. Burket, L. Flynn, W. Klieber, J. Lim, W. Shen, and W. Snavely. Making DidFail succeed: Enhancing the CERT static taint analyzer for Android app sets. CMU/SEI-2015-TR-001. 2015.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys'11*, pages 239–252, 2011.

[7] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and C. Talcott. All about Maude. *LNCS*, 4350, 2007.

[3] All code related to the project can be found in our GitHub page: `https://www.github.com/acidrepo`

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Information Security*, pages 346–360. 2010.

[9] A. Desnos. Androguard. `https://github.com/androguard/androguard`, 2016.

[10] K. O. Elish, D. Yao, and B. G. Ryder. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *MoST*, 2015.

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

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

[13] M. R. Jorge Blasco, Igor Muttik. Wild android collusions, 2016. submitted.

[14] K. Krishnamoorthy. *Handbook of statistical distributions with applications*. CRC Press, 2015.

[15] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon. ApkCombiner: Combining multiple Android apps to support inter-app analysis. In *SEC'15*, pages 513–527. Springer, 2015.

[16] T. Mall and S. Gupta. Critical evaluation of security framework in Android applications: Android-level security and application-level security. *Int. Res. J. of Comp. and Elec. Engg.*, 2, 2014.

[17] C. Marforio, A. Francillon, and S. Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. technical report, 2011.

[18] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of Android apps. In *CCS'12*, pages 241–252.

[19] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[20] G. Rosu. From rewriting logic, to programming language semantics, to program verification. In *Logic, Rewriting, and Concurrency*, pages 598–616. 2015.

[21] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of Android inter-app security vulnerabilities using COVERT. In *ICSE'15*, pages 725–728, 2015.

[22] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *SACMAT'12*, pages 13–22. ACM, 2012.

[23] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS'11*, pages 17–33, 2011.

[24] G. Suarez-Tangil, J. E. Tapiador, and P. Peris-Lopez. Compartmentation policies for Android apps: A combinatorial optimization approach. In *Network and System Security*, pages 63–77. 2015.

[25] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. *Comm. Surveys & Tutorials, IEEE*, 16(2):961–987, 2014.

[26] L. Wu, X. Du, and H. Zhang. An effective access control scheme for preventing permission leak in Android. In *Comp., Networking and Comm.*, pages 57–61. IEEE, 2015.