virus BULLETIN Covering the global threat landscape

DISTINGUISHING BETWEEN MALICIOUS APP COLLUSION AND BENIGN APP COLLABORATION: A MACHINE-LEARNING APPROACH

Irina Mariuca Asavoae[1], Jorge Blasco[2], Thomas M. Chen[3], Harsha Kumara Kalutarage[4], Igor Muttik[5], Hoang Nga Nguyen[6], Liam O'Reilly[1], Markus Roggenbach[1], Siraj Ahmed Shaikh[6]

1. INTRODUCTION

Mobile operating systems support multiple communication methods between apps running on mobile devices. Unfortunately, these convenient inter-app communication mechanisms also make it possible to carry out harmful actions in a collaborative fashion. Two or more mobile apps, viewed independently, may not appear to be malicious. However, in combination they could become harmful by exchanging information with one another and by performing malicious activities together. In 2014, a collaborative project known as ACiD was set up (http://acidproject.org.uk), one of the aims of which was to investigate this potential threat. ACiD stands for 'Application Collusion Detection'. Our main focus was *Android* OS, which we expected to be the primary target for attacks based on app collusion.

Multi-app threats have been considered theoretically for some years, but with the help of tools developed as part of the ACiD project, we were able to discover multiple colluding apps in the wild [1, 2].

In an attempt to evade detection both by mobile security tools and by malware and privacy filters employed by app markets, attackers may try to leverage multiple apps with different capabilities and permissions to achieve their goals – for example, using an app with access to sensitive data to communicate with another app that has Internet access. This technique of app collusion is difficult to detect, as each app will appear benign to most tools, enabling attackers to penetrate a large number of devices for a long period of time before they are caught.

This paper aims to:

- Provide a concise definition of mobile app collusion
- Summarize the state of the art
- Dive into how mobile app collusion attacks are manifested
- Describe what tools and methods (both automated and manual) malware researchers can employ in order to discover and prove the existence of such attacks on *Android* devices.

[1] Swansea University, UK
[2] Royal Holloway, University of London, UK
[3] City University London, UK
[4] Queen's University Belfast, UK
[5] Cyber Curio, UK
[6] Coventry University, UK

Definitions

During our initial analysis it became apparent that it is important to start with a set of good definitions. The reason is that inter-app communications are pretty common and take many forms – from benign to malicious. In most cases, communications are implemented by design and are documented and/or expected by the user. Occasionally, however, one app may use a vulnerability (a bug or a design flaw) in another app to perform actions other than those that are declared via its permissions. Meanwhile, the darkest end of the 'app spectrum' may be populated with app pairs (or triplets, etc.) which are deliberately designed to communicate with each other in order to violate security and privacy.

Identifying the properties associated with such bad behaviours would allow tools to be targeted to discover these malevolent forms of inter-app communication. Of course, the aim would be to exclude, as much as possible, benign apps where communications are implemented deliberately for the benefit of the user.

For a given a set of *Android* applications which are known to communicate with each other, we have defined the following three app categories:

- **Collaborating apps**: these apps are benign, their communications are implemented by design, they are useful, documented, and/or visible to the user.
- **Confused deputies**: in these cases one app is exploiting a vulnerability or a design flaw in another app in order to perform actions beyond its own declared capabilities.
- **Colluding apps**: these are sets of apps where inter-app communications are deliberately used for malicious purposes.

A scalable method for discovering and distinguishing these three app categories could be based on extracting features related to inter-app communications and employing machine-learning methods. To this end, we discuss features that may help solve this classification problem; we also investigate the nature and type of features that can be extracted automatically and discuss tools that can be used for feature extraction. For a number of apps we provide example values of the features.

For practical reasons, and because features separating malicious collusions and confused deputy scenarios are rather

limited, we focus on distinguishing collaborating apps from the two malicious categories.

The rest of the paper is organized as follows: in Section 2, we discuss two typical example sets of collaborating apps and colluding apps. In Section 3, we discuss a number of potential features that could help to distinguish between collaboration and collusion. For each of these features we discuss what it is, why it might be useful, how it can automatically be extracted, what values the feature yields for the example sets from Section 2, and finally, we give a brief evaluation of the feature (which, given the small size of the data set, can only be speculative). In Section 4, we describe various machine-learning approaches that could be applied. We conclude the paper with some remarks on the future of *Android*.

2. EXAMPLES

2.1. A collaborating app set

The *Amazon Shopping* app (com.amazon.mShop.android.shopping, version 14.2.0.100) collaborates with the *Facebook* app (com.facebook.katana, version 149.0.0.40.71) to allow users to share interesting items with others via *Facebook*. When a user finds an item that they wish to share with others, they can use the built-in *Android* share function. When activating this *Android* share facility, the menu that is presented changes depending on what apps are installed (see Figure 1). For instance, if the *Facebook* app is installed then one would be able to share the content with *Facebook* and create a new post. This is an example of collaboration that supports a user's workflow.

Note that the *Amazon Shopping* app documents this behaviour on *Google Play* and lists the following product feature: 'Send and share links to products via email, SMS, Facebook, Twitter, and more.' Note further that in this example we report on the existence of a collaboration but refrain from making any claim concerning the absence of a collusion between these two apps.

2.2. A colluding app set

When working on the ACiD project, we discovered a group of apps that used collusion to synchronize the execution of a potentially harmful payload. This payload was embedded into all the apps through a library called MoPlus SDK. MoPlus is included in more than 5,000 *Android* installation packages (APKs). This library has been known to be malicious since November 2015 [3]. However, the collusion behaviour of the SDK was previously unknown.

We found that apps that included the malicious version of the MoPlus SDK would talk to each other (when running on the same device) to check which of them had the highest

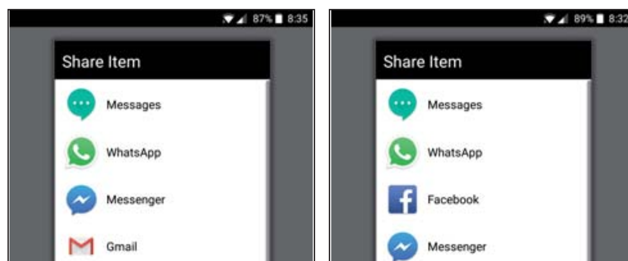

Figure 1: Left: Sharing an item from the Amazon Shopping app without the Facebook app installed. Right: Sharing an item from the Amazon Shopping app with the Facebook app installed yields a new menu entry, allowing the user to post the item to their Facebook account.

ID	Package	Version	MD5
C1	com.baidu.searchbox	6.0	062f91b3b1c900e2bc710166e6510654
C2	com.game.jewelsstar	1.6	00c7a61c7dababe41954879a8ec883dc
C3	com.baidu.browser.apps	5.6.4.0	0230e68490a88d2d4fc0184428ba2c07
C4	com.baidu.browser.apps	5.6.4.0	0658c01e2f28dff29bc40d57df6a0336
C5	com.appandetc.sexypuzzle	1.9.9.1	01a05de59d875077866dc3d81e889d9c
C6	com.baidu.appsearch	6.1.0	05260d6cc0a4d43e0346b368ddce8029
C7	com.baidu.appsearch	6.3.1	03f39e7de7ed90789b349d2a7a097d0b
C8	com.baidu.appsearch	6.4.0	0742c85c39c67c21c0b2fc9f33ab1232

Table 1: Set of colluding apps used for the experiments in this paper. C3 and C4 have the same package and version number, but differ in some of their content.

privileges. The app with the highest privileges would then be chosen to execute a local HTTP client to receive commands from an external C&C server, maximizing the effects of the malicious payload. MoPlus was using SharedPreferences and Intents for inter-app communication. (For a more detailed description of the colluding behaviour of these apps, please see [4].)

For this paper we performed experiments on colluding app sets which carried the MoPlus SDK. Table 1 provides a summary of those apps, along with their package names and MD5 hashes for ease of identification by other researchers.

3. A SPECULATIVE APPROACH TO DISTINGUISH BETWEEN COLLUSION AND COLLABORATION

We investigate the hypothesis that a number of features related to inter-app communications can automatically be extracted from APKs and that, with the help of machine learning, they can be used to distinguish between collaboration and collusion.

It should be noted that our approach is necessarily a speculative one due to a lack of labelled data (in other words, app sets classified as either benign or colluding maliciously) – there has been only one set of colluding apps identified in the field, and it is a challenge to provide many app sets exhibiting beneficial collaboration (where ideally one would also know that these collaborating apps don't collude).

To this end, we have identified a number of potential static features for which we will examine:

- Why they might be useful in order to distinguish between collusion and collaboration
- Whether they can be extracted automatically using current tools
- Their nature.

Apart from simple features related to communications (e.g. sending and receiving intents) there are several less obvious features to consider:

1. Code obfuscation
2. Properties extracted from app documentation
3. Ability of an app to detect if other apps are installed
4. Permissions used vs permissions requested
5. Code similarity.

For each of these features we will extract sample data from the collaborating and colluding app sets described in Section 2. Having examined the feature, we will discuss how suitable it appears for our purpose. Criteria include the possibility of automatic extraction and first indications based on our examples.

Additionally, we have considered the feature:

6. User interaction.

Though this might be quite a useful feature, due to a lack of corresponding tools, it is not possible to extract it automatically. We shall discuss in detail why this feature (currently) is not applicable.

Finally, we explore dynamic features.

3.1. Code obfuscation

What it is: Code obfuscation refers to code transformations that hide code functionality (fully) from human readers, e.g. for the purpose of intellectual property protection, or for disabling automatic functionality detectors, e.g. code analysers. A survey of code obfuscation techniques and obfuscation tools is presented in [5].

Why it might be a useful feature: [6] reports on methods to identify the presence of obfuscations via various obfuscators, such as *Bangcle* and *ProGuard*, while evaluating the likelihood of the presence of malware when these tools have been used. According to [6], *Bangcle* indicates an elevated risk of malware presence while *ProGuard* appears to be used in a more legitimate fashion, for protecting intellectual property.

We propose to use *automatic obfuscation evaluation* (AOE) to determine: (1) whether an app has been obfuscated, (2) which obfuscator(s) were used, and (3) which obfuscation techniques were applied. Further, to provide 'obfuscation collusion risk for a set of apps', denoted as OCR(S), we aggregate all the individual AOEs of the apps in the set S. Then we use OCR(S) to discriminate between collusion and collaboration, where sets below a certain threshold of OCR(S) would be considered 'low risk'. Next, we explain how AOE works at each step (1-3):

1. If an app is obfuscated, a dedicated tool can provide the degree of obfuscation, OD, which effectively obstructs malware analysis tools.
2. According to [6], identifying the obfuscator(s) used to hide the app code helps discriminate collusion from collaboration based on a credibility factor associated with an obfuscator. Consequently, we assume that the *malware production risk* (MPR) associated with an obfuscator is proportional to AOE.
3. Some obfuscation techniques are known to hide the code completely, e.g. (partial) server-side execution may load code from a remote resource. We use the code-hiding degree, HD, combined with OD to define an app's obfuscation risk evaluation (Oth-RE) as proportional to HD and OD.

Finally, the value AOE for an app is defined as the product of MPR and Oth-RE.

Automated extraction of this feature: A recent tool, which we refer to here as TOD, described in [7], uses machine learning to identify the obfuscator of an *Android* app for a given set of obfuscation tools and for a number of their configuration options. This appears to be the first and only work to automatize the obfuscation identification problem. Technically, the authors of [7] identify a feature vector that represents a characteristic of the obfuscated code. TOD extracts this feature vector from the Dalvik bytecode and uses it to identify the obfuscator provenance information.

TOD focuses on obfuscations at the level of class, field and method names, as well as evaluating the package structure after removing unused code. Based on these parameters, the tool learns the different patterns of obfuscations available for obfuscators from a given training set. Currently, the training set focuses on learning obfuscation patterns for five obfuscators: *ProGuard* [8], *Allatori* [9], *DashO* [10], *Legu* [11] and *Bangcle* [12]. Experiments indicate that TOD identifies the obfuscator with 97% accuracy and recognizes the obfuscator's configuration with more than 90% accuracy. Based on [6], we associate *ProGuard* with low MPR while *Bangcle* gets a high MPR value.

Feature extraction from the running examples: Table 2 presents the evaluation results produced by TOD for the apps described in Section 2. We would like to thank the authors of [7] for providing us with these results. TOD produces a safe result in that, when a known obfuscator is detected as having been applied to an app, that obfuscator is reported as the result; in other cases, the reported result is 'Not known', which means that either an unknown obfuscator was used or no obfuscation was detected. Admittedly, the results in Table 2 do not help in distinguishing between collusion and collaboration solely via MPR for this example set. Note also that, for the moment, we do not provide an AOE aggregation formula for OCR(S).

Evaluation of the feature: Using a similar approach to the one employed in [6] for obfuscation tools and including risk evaluation for obfuscation techniques (e.g. techniques that make static analysis impractical) should assist in collusion/collaboration differentiation, via Oth-RE. However, currently, the TOD tool does not include detection of obfuscation techniques and the results for our example set do not show obvious differentiations.

3.2. Properties extracted from app documentation

What it is: It is clear that it should be possible to inspect app documentation and classify/cluster apps according to their described behaviour (e.g. game apps, weather apps, etc.). The *Google Play* app ranking algorithm is a good example of this approach. It utilizes app metadata such as title, description and reviews in order to rank user search results based on their relevance to user queries.

Why it might be a useful feature: We speculate that app descriptions can be utilized to distinguish between colluding and collaborating apps, subject to the assumption that benign collaborative behaviours (if any) will be documented in the app description and that behaviours that are invisible to the user are 'suspicious'. For example, for an app/app pair belonging to a certain class of apps, say gaming apps, there might be a certain expectation as to which resources (i.e. permissions) it might need. In other words, the described behaviour (as found in the app descriptions on, e.g. *Google Play*) should comply with the resources requested by an app/app pair, and any discrepancy between the described behaviour and requested resources warrants further investigation either in the context of collusion or as an individual malicious app. So, we need to analyse the app descriptions automatically to extract three features: (a) the category of the app, (b) resources that it wants to use, and (c) collaboration with other apps, as in the example shown in Table 3.

Automated extraction of this feature: Using the app description, features (a), (b) and (c) can be extracted as follows: we create a sample corpus of app descriptions for each app category in the *Google Play* app store (see Table 4). A corpus is a structure for storing text documents of app descriptions with their metadata. After that we do some basic

	Application	**Obfuscator : AOE**
C1	com.baidu.searchbox	ProGuard **:** LOW x Oth-RE
C2	com.game.jewelsstar	Not known **:** Oth-RE
C3	com.baidu.browser.apps	Not known **:** Oth-RE
C4	com.baidu.browser.apps	Not known **:** Oth-RE
C5	com.appandetc.sexypuzzle	ProGuard **:** Low x Oth-RE
C6	com.baidu.appsearch	ProGuard **:** Low x Oth-RE
C7	com.baidu.appsearch	Not known **:** Oth-RE
C8	com.baidu.appsearch	Not known **:** Oth-RE
A1,2,3	Amazon XYZ.apk	Not known **:** Oth-RE
F	Facebook_v149.0.0.40.7…om.apk	ProGuard **:** Low x Oth-RE

Table 2: Detection of obfuscation tool using TOD.

text preprocessing on our corpus – for example, removing extra white spaces and document words, ignoring extremely rare words and very common words, etc. Then we create the fundamental object for our text analysis called the Document Term Matrix (DTM). DTM is a matrix that describes the frequency of each term in each document in our corpus arranged in rows (app names) and columns (terms in the descriptions). If a term occurs in a particular app description *n* times, then the matrix entry corresponding to that row and column is *n*, otherwise 0. Finally, based on DTM, we can build our classifier to classify a given new app description in a known category, as DTM serves as a feature vector for this purpose. In order to extract (b) and (c) above, it's possible to employ a simple regular expression search on the app description text.

Feature extraction of the running examples: In order to compute discrepancies between described behaviour (i.e. app category) and requested resources shown in Table 3, we need a list of all permissions that are typically associated with each app category type in Table 4. For example, weather applications generally request the following resources (permissions): approximate location, precise location, view Wi-Fi connections, view network connections, full network access, control Near Field Communication, etc. Likewise, we can compute a 'norm' resources set for each category using a statistically significant sized sample set. Finally, for any given pair of apps, we can compute the distance between requested resources (Table 3) and the 'norm' resources set for that app category using a DTM. If this distance exceeds a certain threshold then the target app/app pair is suspicious and warrants further investigation.

We implemented the above analysis using R – a language designed for statistical computing. It is possible to automate the entire analysis with the help of web-crawling tools (e.g. *SEO Spider*, *Selenium*) and machine-learning tools (e.g. R, Python).

Evaluation of the feature: As shown in Table 3, MoPlus apps have not documented their association with other apps in their app descriptions. However, *Amazon* has documented its cooperation with the *Facebook* and *Twitter* apps. From the machine-learning perspective, it's true that a few counter examples are insufficient to draw a generalized conclusion, but this observation provides us with reasonable grounds to speculate that this kind of analysis could become a useful feature and that it would be worthwhile investigating further.

3.3. Ability of an app to detect if other apps are installed

What it is: The *Android* operating system offers a wide range of communication options that enable apps to cooperate and share information. Additionally, *Android* allows apps, when the necessary permissions are requested, to check if other apps are installed. This feature could be used by collaborating apps to adapt their interface and functionality to the apps available on the execution device, but it could also be used to coordinate a collusion attack.

Why it might be a useful feature: Colluding apps can execute their actions without checking if their collusion counterparts are installed and currently running. However, for attackers this approach has several drawbacks. Firstly, the attack may not be successful if the second app is required but is inactive. Secondly, the malicious payload may create visible indicators (system calls, logs, files, etc.), which would simplify discovery of the collusion even when a single app is analysed. So we expect most colluding apps to include a step to detect if their colluding 'buddies' are installed and active on the victim's device.

App name	Category	Requested resources	Documented collaboration
A1	Shopping	Find accounts on the device, add or remove accounts, read your contacts, approximate location, ...	Facebook, Twitter
F	Social	Retrieve running apps, find accounts on the device, add or remove accounts, read your own contact card, ...	None
C3	Communication	Approximate location, precise location, ...	None
C6	Tools	Find accounts on the device, read the contents of your USB storage, ...	None

Table 3: Feature extraction (due to space constraints only part of the table is presented).

Category	Examples
Art & design	Sketchbooks, painter tools, art & design tools, colouring books
Books & reference	Book readers, reference books, textbooks, dictionaries, thesaurus, wikis
Shopping	Online shopping, auctions, coupons, price comparison, shopping lists, product reviews

Table 4: Possible app categories in the Google Play app store [13] (due to space constraints only part of the table is presented).

Automated extraction of this feature: App developers have access to several API calls and commands to detect the presence of other apps in the system. These can be detected by static analysis tools such as *Androguard* [14].The following list summarizes the indicators which could be used:

- **M1:** The *Android* ActivityManager and PackageManager classes offer information about opened and installed apps. Apps using these classes to get information about the running processes on the system must request the GET_TASKS permission.
- **M2:** The list of running processes can be accessed through a shell command. *Android* allows apps to execute shell commands through the Runtime class. For instance, the list of running processes could be obtained by executing the *Linux* 'ps' command via Runtime.getRuntime().exec('ps aux'), which returns a Process object.
- **M3:** A colluding app could create a lock file or variable inside any of the *Android* shared resources such as Shared Preferences, the external storage or the Internet. Access to the external storage requires the READ_ EXTERNAL_STORAGE and WRITE_ EXTERNAL_ STORAGE permissions. Access to the Internet requires the INTERNET permission.
- **M4:** Colluding apps can register an IntentFilter to answer specific broadcast intents from other colluding apps. Other apps can check if an IntentFilter is registered by other Broadcast receivers through the PackageManager API. The queryBroadcastReceivers method returns those BroadcastReceiver objects that have registered to answer a specific Intent. In a similar way, this class also offers the queryIntentActivities method, which returns the Activity objects that would answer a specific Intent.

These APIs allow apps to verify if another app is installed but do not necessarily indicate the existence of colluding apps. However, since we expect colluding apps to check if their 'buddies' are installed on the same device, we consider this feature to be indicative of collusion.

Feature extraction of the running examples: In fact, one of these indicators was present in a documented field collusion case [4]. In this case, colluding apps checked if a specific BroadcastReceiver was installed to see if the other colluding apps were installed and running on the victim's device (M4).

	M1	M2	M3	M4
MoPlus (colluding)	√			√
Amazon/Facebook (collaborating)	√		√	√

Table 5: Comparison of the MoPlus SDK with Amazon and Facebook apps.

Table 5 compares the MoPlus SDK (all apps which include malicious versions of this SDK are known to collude [3]) with the *Amazon* and *Facebook* apps. For each detection possibility, we show (with a check mark, '√') which sets of apps have the indicators in at least one of the apps. In the case of MoPlus apps, we have analysed classes belonging to the MoPlus SDK, not the whole binary file.

Evaluation of the feature: While extraction of this feature is not very hard, the results show no significant difference between benign and malicious app sets. Collaborating apps may also check if other apps are present in order to improve user experience.

3.4. Permissions used vs permissions requested

What it is: In this section, we show that permissions analysis based on manifest files can provide a useful insight when differentiating between collusion and collaboration.

Why it might be a useful feature: Intuitively, honest programmers will always be transparent about the permissions that are used in their applications. Conversely, malicious programmers are likely to request many permissions in order to obtain more privileges, or to hide the use of critical permissions in the manifest files of their applications. In this section, we will examine whether analysing permissions is useful to differentiate between the two. In particular, we will look at ratios between used and declared permissions and at the number of permissions used without being declared.

Automated extraction of this feature: We used Permission Checker[7] [15], a permissions analysis tool for APK files, to extract the requested permissions from the manifest file within an APK and the permissions used by method invocations in the APK's bytecode. To this end, *Android* permissions (AP) are categorized into four different groups:

- Declared permissions (DAP): APs declared in the *Android* manifest file.
- Used permissions (EAP): APs declared and used in the bytecode.
- Ghost permissions (GAP): APs used but not declared.
- Useless permissions (UAP): APs declared but not used.

Given an application, A, DAP_A denotes the set of declared permissions for A, and EAP_A the set of used permissions. For each pair of applications (A,B), we are interested in the ratio of used vs declared APs, defined as:

$$R_{A,B} = |EAP_A \cup EAP_B| \,/\, |DAP_A \cup DAP_B| \in [0,1]$$

and the number of undeclared permissions:

$$U_{A,B} = |GAP_A \cup GAP_B| \in \mathbb{N}$$

[7] The results of this tool should be interpreted with a little caution: on applying it to apps we had written ourselves, the results were not completely accurate.

Feature extraction of the running examples: We apply Permission Checker and compute *R* and *U* first for the collaboration set containing *Facebook* and *Amazon* applications. They are: (A1) *Amazon Shopping* (version 14.2.0.100), (A2) *Amazon Prime Video* (version 3.0.213.147041), (A3) *Amazon Underground* (version 8.0.0.200), and (F) *Facebook* (version 149.0.0.40.71). The results are shown in Table 6.

$R_{A,B}$	A1	A2	A3
F	0.10	0.18	0.14

$U_{A,B}$	A1	A2	A3
F	0	0	0

Table 6: Pair analysis of Amazon and Facebook applications.

The same analysis is applied to the colluding set of MoPlus applications. The results are shown in Table 7.

For benign apps, Table 6 shows that the $R_{A,B}$ ratio varies from 0.14 to 0.18, while the number of undeclared permissions, $U_{A,B}$, is 0. Table 7 shows that, for colluding pairs of MoPlus applications, the $R_{A,B}$ ratio ranges between 0.17 and 0.29, while undeclared $U_{A,B}$ can go up to 2.

Evaluation of the feature: We demonstrated that this feature can be extracted automatically. Overall, our evaluation supports the conjecture that $R_{A,B}$ values are smaller for collaboration than for collusion. Therefore, they can be useful features to differentiate between collaboration and collusion.

3.5. Code similarity

What it is: Identifying code similarities serves many purposes which include studying code evolution, detecting source code plagiarism, enabling refactoring, and performing defect tracking and repair.

Why it might be a useful feature: All maliciously colluding apps ought to operate together, so they are likely to be coded by the same person (or team) and during the same time frame. This makes it more than likely that they would have identical development environments, share significant portions of source code, use the same libraries, and perhaps even share a coding style and include the same programmer's errors. Therefore code similarity metrics should be strong and useful features.

Automated extraction of this feature: Many tools are freely available to detect and measure code similarity. They are based on different techniques, ranging from lightweight line- and token-based syntactic approaches to heavyweight semantics-based approaches. For example, a lightweight tool like *Simian* (*Similarity Analyser*) identifies code similarities based on syntactic techniques, but does not quantify the degree of similarity that exists between potential clones – but a user can define a similarity score based on its output, as shown later in this section. Moreover, a tool like *Moss* emphasizes the semantic similarities of programs using the 'winnowing algorithm' [16] to selected fragments of source code to be fingerprinted, and then calculates a similarity percentage based on the set of common fingerprints.

Feature extraction of the running examples: In order to demonstrate the feasibility of proposed feature extraction, we downloaded an APK from the *Google Play Store* along with its source code [17]. Using *JADX* [18] – a tool for creating Java source code from *Android* DEX and APK files – Java source code was produced for the APK. Then *Simian* [19] was employed to calculate similarities between the original source code and the re-engineered source code. *Simian* reported 12 similar lines out of 176 line comparisons in this analysis, which resulted in a 12/176 = 0.07 similarity score. Though we compared the original source code of the app against its re-engineered code, the similarity score obtained here was very low. This is probably due to the fact that *Simian* performs only syntactic pattern matching rather than functional/semantic matchings. One can never get back to the exact same source since there is no metadata with the compiled code. Therefore, re-engineered code may be syntactically different from its original source code and a low similarity score via syntactic pattern matching should be expected.

Evaluation of the feature: Table 8 presents the code similarity scores automatically computed for the set of

$R_{A,B}/U_{A,B}$	C1	C2	C3	C4	C5	C6	C7	C8
C1	N/A	0.28/1	0.19/1	0.19/1	0.21/1	0.27/0	0.19/0	0.19/0
C2	0.28/1	N/A	0.20/0	0.20/0	0.28/2	0.29/0	0.20/0	0.19/0
C3	0.19/1	0.20/0	N/A	0.19/0	0.18/1	0.23/0	0.17/0	0.17/0
C4	0.19/1	0.20/0	0.19/0	N/A	0.18/1	0.23/0	0.17/0	0.17/0
C5	0.21/1	0.28/2	0.18/1	0.18/1	N/A	0.24/0	0.18/0	0.17/0
C6	0.27/0	0.29/0	0.23/0	0.23/0	0.24/0	N/A	0.19/0	0.18/0
C7	0.19/0	0.20/0	0.17/0	0.17/0	0.18/0	0.19/0	N/A	0.18/0
C8	0.19/0	0.19/0	0.17/0	0.17/0	0.17/0	0.18/0	0.18/0	N/A

Table 7: Pair analysis of MoPlus applications on $R_{A,B}$ and $U_{A,B}$.

MoPlus apps. The score for the *Facebook-Amazon* app pair is 0.26 and there is no clear distinction from the MoPlus app values. However, the sample size is not statistically significant so we cannot draw any conclusion here. As mentioned above, getting back to the exact same source code is difficult, and re-engineered code is syntactically different from its original source code. This might have a negative effect on the usefulness of this feature in practice.

	C1	C2	C3	C4	C5	C6	C7	C8
C1	N/A	0.081	0.081	0.174	0.088	0.118	0.109	0.131
C2	0.081	N/A	0.01	0.01	0.028	0.012	0.016	0.013
C3	0.081	0.01	N/A	1	0.008	0.062	0.061	0.073
C4	0.174	0.01	1	N/A	0.008	0.062	0.061	0.073
C5	0.088	0.028	0.008	0.008	N/A	0.009	0.014	0.01
C6	0.118	0.012	0.062	0.062	0.009	N/A	0.883	0.764
C7	0.109	0.016	0.061	0.061	0.014	0.883	N/A	0.647
C8	0.131	0.013	0.073	0.073	0.01	0.764	0.647	N/A

Table 8: Similarity scores (pairwise) for MoPlus applications.

3.6. User interaction

What it is: This feature is defined based on the apps' interaction with the user and it aims to predict malicious behaviour based on measuring the honesty of an app's communication with the user. We propose to evaluate the quality and quantity of the messages/images used by an app for user interaction.

Why it might be a useful feature: If an app shows a standard *Android* 'Share' icon, then it is likely that corresponding communications with other apps would be benign. Even better would be if there was a message that explained what is shared and why. However, producing good user interaction (UI) evaluators is a challenging task. In our view, the evaluation of the interaction needs to focus on the content of the UI. For example, a long, detailed message (which is easier to evaluate) explaining a button requesting access to a particular resource would work well for more mature users. If, however, the same long text is presented to a child, then it might not be quite as effective: due to different perception capabilities, there are different classes of users. Optimally, we would want to use a UI analysis tool capable of combining analysis of the text and the complexity of the UI elements to produce a collusion risk evaluation. Moreover, a combination of such UI analysis with API mapping could be used to decide whether a certain UI element, such as a button, should allow a specific API call. We foresee this analysis as being rather useful when looking for collusion, but there is a challenge in finding suitable tools.

Automated extraction of this feature: At the moment, there are several tools capable of testing user interaction in *Android* apps, for example *Monkey Runner* [20] and *Espresso Testing Framework* [21]. *Monkey Runner* allows apps to be installed and tested by generating stimuli so that an app does things as if a user were interacting with it. The tool basically acts as a software robot, which produces a sequence of touch events that, when sent to an emulator or device, interacts as a human would. Coupling this with a dynamic analysis tool would allow us to inspect which UI elements trigger communication APIs.

The *Espresso Testing Framework* is included within *Android* and allows the execution of UI tests. Instead of generating touch events, this framework accesses the interface elements of the screen directly. This approach would prove useful for extracting the interface elements of apps for evaluating their quality, eventually by means of a wrapper.

Feature extraction of the running examples: We were not able to find a tool that suits our aims for UI collusion risk evaluation. As mentioned, the available UI tools target slightly different usage scenarios. This makes automated feature extraction cumbersome.

Evaluation of the feature: The UI collusion risk evaluation could be executed via a combination of techniques involving data mining, text evaluation and image processing. We still believe that UI interaction has potential in distinguishing malicious behaviour.

3.7. Dynamic features

In the previous sections, we discussed many features which would assist in discovering colluding apps and which can be extracted statically. The feature set may be improved further by adding dynamic features, extracted at runtime. In many cases dynamic tools would also be helpful to improve extraction of static features – for example when static analysis tools fail to detect and extract the features due to code obfuscation or due to the code using reflection.

One approach to facilitate dynamic analysis is to put wrappers around apps in order to log their actions – for example, a wrapper might log accesses to restricted resources and external communications. One such tool is *APIMonitor* from the team that developed *Droidbox* [22]. To extract the runtime features of an app one would need to apply the wrapper to that app, execute it, and then analyse the logs generated by *APIMonitor*.

Another approach is to create a special instrumented version of *Android*, populate it with multiple apps, and record the actions. There are existing research tools based on this approach, for example *CopperDroid* [23], which is focused on dynamic analysis of apps. *CopperDroid*, however, is currently only capable of monitoring a single app, although it may be extended to execute multiple apps and extract features related to access to sensitive resources and inter-app

communications. Unfortunately, using dynamic feature extractors was well beyond the scope of our project.

4. MACHINE LEARNING

Having explored a number of features, we will now discuss how one could potentially use them to train a model with a machine-learning approach in order to distinguish automatically between app collaboration and collusion.

In general, it is the quality of the input data that determines the output quality of machine-learning algorithms. Hence, after exploring a potential feature set, it is necessary to perform data exploration (including feature engineering), cleaning and preparation before initiating modelling and evaluation. This will help us to systematically identify important features among the feature set that actually inform our modelling. Unfortunately, the sizes of the app sample sets (both colluding and collaborative) employed in this paper were not sufficient for this purpose.

Learning algorithms are chosen based on the given input data and the learning task at hand. For example, if labelled data is provided and the learning task is 'classification', then we can apply a **supervised learning** algorithm, e.g. support vector machine (SVM) [24], to the problem. In the collusion context, classification refers, for example, to automatically labelling a previously unseen app pair as either colluding or collaborating. However, applying a supervised algorithm to our problem would be difficult due to the lack of known collaborating and known colluding app samples that are available in the wild. As in many other security problems, this represents a major constraint in applying supervised learning techniques. However, it is still possible to apply unsupervised, semi-supervised or novelty (anomaly) detection techniques [25, 26] to this problem.

In **unsupervised learning**, we only need the input data of the features, no corresponding output labels are required. The goal here is to model the underlying structure or distribution of the data in order to learn more about the dataset. Algorithms are left on their own to discover and present interesting structures rather than training them using labelled data. Our problem can be modelled as a clustering (e.g. k-means) or association (e.g. AprioriDP) rule problem [25]. Clustering discovers the inherent groupings in our dataset while association discovers interesting relationships between elements of the input data. The underlying assumption here would be that colluding and collaborating apps have different distributions in terms of the above features and therefore should form disjoint groups that correspond to colluding and collaborating app sets.

As shown in the literature [27], app collusion is a real threat. But as far as anyone knows, the base rate of colluding apps in the wild is close to zero. In this situation, approaching the issue as a **semi-supervised learning** problem would be sensible. Many real-world security problems (e.g. credit card/toll fraud detection) fall into this category, in which only some events are labelled as benign/malicious and the majority are unlabelled [28]. This is because it can be expensive to label data, as domain experts are required. Unlabelled data is typically cheap and easy to collect and store. In this case unsupervised learning algorithms can be employed to discover and learn structures within the input variables, and then supervised learning (e.g. SVM, KNN) [25] can be applied to make best-guess predictions for the unlabelled data. After that, the data can feed into a supervised learning algorithm for training purposes and the model can be used to make predictions on new, unseen app pairs as they are colluding and collaborating.

Alternatively, it might be possible to obtain a considerable amount of data representing the benign class (i.e. non-colluding), whilst not having sufficient and reliable data representing the malicious class (i.e. colluding). This would hinder training and, in particular, the testing procedures of whatever algorithms are chosen to solve this problem. In order to minimize this difficulty, we propose a **one-class modelling approach** (including novelty/anomaly detection) [29] to move forward in such a context. The idea here would be first to train a model using only benign samples. This trained model could then be used for the identification of new/unknown data (that the model had not been trained with) with the help of either statistical or machine-learning based approaches. To achieve this, spot-checking the one-class support vector machine would be the first step.

Note that none of the above learning algorithms needs to be built from scratch. Many useful libraries of extensible algorithms are freely available in R, Python, and also ML libraries in other programming languages. These can be extended and adapted for this purpose.

5. FUTURE OF ANDROID

Malicious collusion became possible in *Android* due to a series of unfortunate design decisions. Future versions of *Android* (or any OS that succeeds it) must have a better defined and more regulated communication framework in general. We strongly advocate the following changes:

Firstly, we believe that the OS should require an explicit declaration of all app communication methods in the app manifest. At the moment this is done with the exported property for Services and BroadcastReceivers, but the operating system does not even make them visible to the user. If app developers were required to make explicit declarations then all inter-app communications would be visible to users and analysis tools would be able to inspect them. This approach would also underpin more granular policies (in *Google Play* as well as in the OS at runtime) around

communications. Similarly, it would be very desirable to have a declaration in the manifest of all the Internet domains and URLs that an app is allowed to contact. That would provide a much better and more granular control than the very broad INTERNET permission. It would also allow analysis of communications which occur outside of the device, where an external website is operating as an intermediary between two apps on the same device. This step would significantly reduce the risk of privacy intrusions from advertisement libraries embedded in apps and would allow better methods for checking the privacy status of apps.

Secondly, we propose allowing apps to interact while enabling them also to evaluate the interaction with other apps. The evaluation protocol would be pre-set and intended to give each app a certain authority level in evaluating the quality of the interaction with its peers. The protocol could be provided via a wrapper that each application is either allowed or required to use in certain *Android* environments. This approach would enable post-mortem analysis methods for evaluating apps based on their behaviour in interaction with their peers. The main question that our future research would, in this case, need to answer is: which basic set of rules should the peer evaluation system provide? This basic set of rules would be crucial for allowing discrimination between collusion and collaboration. An approach that implements this to some extent is Android Work [30], which introduces a further sandboxing that separates personal apps from professional ones: the mobile device management administrator decides which apps go into the professional sandbox, and the user is able to install apps on the personal sandbox. Apps from the two different sandboxes are not allowed to interact and they can be switched on and off by category. This would mean implementing OS support for app group isolation – similar, for example, to *Samsung Knox*.

Both of our proposals would have to include a regulatory phase and an evaluatory/analysis mechanism. In the regulatory phase a set of rules should be provided either by *Android* OS, e.g. the finer-grained communication intents declaration suggested in the first proposal, or by *Android* wrappers, e.g. the set of standards in app evaluation of peer communication. The evaluatory phase uses the set of rules introduced in the regulatory phase to analyse apps and discover properties like collusion. For example, the first proposal aims for traditional analysers to benefit from additional information in order to produce more accurate results. Meanwhile, the second proposal relies on the post-mortem evaluatory phase that focuses on apps' feedback based on communications to evaluate collusion risk.

Coupling of the regulatory and evaluatory phases aligns the research desideratum of our current work with today's trend of regulating and verifying AI. Namely, with recent AI developments, it has become obvious that technology may go out of control and may produce unexpected problems. Take, for example, the case of the *Facebook* AI business agent-to-agent negotiation project [31] that used natural language and produced unexpectedly efficient results. The conclusion of the project was that AI may be able to surpass intuitive strategies when trained well enough. Consequently, the scientific community is currently organizing itself to evaluate the status and progress of AI, and to propose an (initial) set of measures for controlling the direction of its development [32].

Moreover, we envisage seeing the usage of AI/machine-learning methods in app code. Hence, early security measures must be provided in the form of more (self) regulation and safer activity of apps in *Android* and other mobile/IoT operating systems.

6. CONCLUSION

There cannot be collusion in a single app, there has to be a set (a pair/triplet/quadruplet, etc.). Having in mind a high number of *Android* apps and an almost infinite number of app sets, it soon becomes clear that only only automated methods are appropriate for discovering collusions and for distinguishing between benign cooperation and malicious activities.

We have described and evaluated a set of carefully selected features (based on common sense as well as on expert opinions). We have shown that machine learning is a promising approach to distinguish between colluding and collaborating apps. Many of the proposed features could form a solid basis for detecting malicious app sets in the *Android* universe. In some cases we have identified technological gaps where tools are missing or require more work before they can be applied to the problem at hand.

The ultimate solution, however, is to implement changes in the OS – that would discourage abuse and allow much easier, automated discovery when it happens.

REFERENCES

[1] Asavoae, I.M.; Blasco, J.; Chen, T.M.; Kalutarage, H.K.; Muttik, I; Nguyen, H.N.; Roggenbach, M.; Shaikh, S.A. Detecting Malicious Collusion Between Mobile Software Applications: The Android TM Case. In Palomares Carrascosa, I.; Kalutarage, H.K.; Huang, Y. (Eds.): Data Analytics and Decision Support for Cybersecurity – Trends, Methodologies and Applications, Springer 2017.

[2] Blasco, J.; Chen, T.M.; Muttik, I.; Roggenbach, M. Detection of App Collusion Potential Using Logic Programming. Journal of Network and Computer Applications, Volume 105, 1 March 2018, pp.88-104.

[3] 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/.

[4] Blasco, J.; Muttik, I.; Roggenbach, M.; Chen, T.M. Wild Android Collusions. In Proceedings of the 26th Virus Bulletin International Conference, 2016. https://www.virusbulletin.com/virusbulletin/2018/03/vb2016-paper-wild-android-collusions/.

[5] Collberg, C.; Thomborson, C.; Low, D. A taxonomy of Obfuscating Transformations. Technical Report 148, The University of Auckland.

[6] Apvrille, A.; Nigam, R. Obfuscation in Android Malware and how to Fight Back. https://www.virusbulletin.com/virusbulletin/2014/07/obfuscation-android-malware-and-how-fight-back.

[7] Wang, Y.; Rountev, A. Who Changed You? Obfuscator Identification for Android. In MOBILESoft@ICSE 2017: 154-164.

[8] ProGuard. https://www.guardsquare.com/en/proguard; https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html#manual/introduction.html.

[9] Allatori. http://www.allatori.com/.

[10] DashO. https://www.preemptive.com/products/dasho/overview.

[11] Legu. http://wiki.open.qq.com/wiki/%E5%BA%94%E7%94%A8%E5%8A%A0%E5%9B%BA.

[12] Bangcle. https://www.bangcle.com/.

[13] https://support.google.com/googleplay/android-developer/answer/113475?hl=en-GB.

[14] Androguard. https://github.com/androguard/androguard.

[15] Merlo, A.; Georgiu, G.C. RiskInDroid: Machine Learning-Based Risk Analysis on Android. SEC 2017: 538-552. https://link.springer.com/chapter/10.1007%2F978-3-319-58469-0_36.

[16] Schleimer, S.; Wilkerson, D.; Aiken, A. Local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003 June 9 (pp.76-85). ACM.

[17] https://github.com/gabrielecirulli/2048.

[18] JADX. https://github.com/skylot/jadx.

[19] Simian. http://www.harukizaemon.com/simian/index.html.

[20] Monkey Runner. https://developer.android.com/studio/test/monkeyrunner/index.html.

[21] Espresso Testing Framework. https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[22] Droidbox. https://github.com/pjlantz/droidbox.

[23] Tam, K.; Salahuddin, J.K.; Aristide, F.; Cavallaro, L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In NDSS. 2015.

[24] Corinna Cortes, C.; Vapnik, V. Support-vector networks. Machine learning 20, no. 3 (1995): 273-297.

[25] Roiger, R.J. Data mining: a tutorial-based primer. CRC Press, 2017.

[26] Buczak, A.; Guven, E. A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Communications Surveys & Tutorials 18, no. 2 (2016): 1153-1176.

[27] 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 2012 Dec 3 (pp. 51-60). ACM.

[28] Jaiswal, A.; Manjunatha, A.S.; Madhu, B.R.; Chidananda Murthy, P. Predicting unlabeled traffic for intrusion detection using semi-supervised machine learning. In Electrical, Electronics, Communication, Computer and Optimization Techniques (ICEECCOT), 016 International Conference on 2016 Dec 9 (pp.218-222). IEEE.

[29] Pimentel, M.; Clifton, D.A.; Clifton, L.; Tarassenko, L. A review of novelty detection. Signal Processing 99 (2014): 215-249.

[30] Android Work. https://www.android.com/enterprise/employees/.

[31] Lewis, M.; Yarats, D.; Dauphin, Y.; Parikh, D.; Batra, D. Deal or No Deal? End-to-End Learning of Negotiation Dialogues. In EMNLP 2017: 2443-2453.

[32] Boddington, P.; Millican, P.; Wooldridge, M. Minds and Machines Special Issue: Ethics and Artificial Intelligence. Minds and Machines 27(4): 569-574 (2017).

Editor: Martijn Grooten

Head of Testing: Peter Karsai

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionuţ Răileanu, Chris Stock

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

© 2018 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Email: editor@virusbulletin.com

Web: https://www.virusbulletin.com/