Android App Collusion

Ervin Oro
ervin.oro@aalto.fi

Tutor: Jorden Whitefield

set proper margins and document format

Abstract

abstract

KEYWORDS: *list, of, key, words*

1 Introduction

Android is an operating system (OS) that is primarily designed for mobile devices, e.g., smartphones and tablets. With more than two billion active devices [1], it is estimated to be the most widely used OS, surpassing even Microsoft Windows [2, 3]. Android is designed to be an open platform: developed and maintained by Google LLC, but largely released as the Android Open Source Project for everyone to study and evaluate [4]. The Android OS includes support for apps, which are easily installable application packages that can extend the functionality of devices. Apps can be developed and distributed by anyone with a very low barrier of entry.

While this popularity of Android is not reflected by the proportion of malware attacks, most of which still target Windows, both the number and complexity of attacks against Android are increasing [5]. This is especially troublesome, as many people increasingly rely on their smartphones – often to store their personal data, online account credentials, money, and more. McAfee estimates that revenues for mobile malware authors could be in the billion-dollar range by 2020 [6].

Given the increasing potential damage from Android malware, defending against it is an active area of research. Android uses a multi-layer security approach, combining machine learning, platform security and secure hardware [1]. Machine learning methods are utilised by Google Play Store to prevent uploading potentially harmful applications, and by Google Play Protect [7] to scan apps locally on users' devices. The platform security of Android has been enhanced over the years with the addition of multiple security features, for example, SELinux protections [8, "Security-Enhanced Linux in Android"], exploit mitigations [9], privilege reductions [10], and encryption. Recent versions of Android leverage hardware security features, including keystore and remote key attestation [11], and receive regular software updates. These security mechanisms have been partially successful, as exploit pricing and difficulty are growing by some estimates [1].

figure depicting multi-layer security could be added

However, malicious actors are continuously developing exploits to bypass existing protections, and a number of threats, e.g., app collusion, cannot yet be reliably detected nor defended against. App collusion is a secret collaboration between apps with malicious intentions (Section 2). This can be facilitated by any of the numerous ways for apps to communicate with each other that the Android system provides (Section 3). Methods for apps to collude also exist on the iOS platform [12]. Given a malicious app that would be detected and blocked bt state of the art security systems, its functionality can be split into several apps, so that each of them would be categorised as benign when analysed separately [13].

Android app collusion is not a new concept [14], and multiple attempts have been made to develop suitable detection systems (Section 5). Computationally simple but very coarse filters have been developed based on statically extracted features of apps [15, 13], while computationally expensive and more accurate filters have used formal modelling of Android apps [16] or modified versions of the Android OS [17] to track information flows. Heuristic approaches and manual analysis has also been proposed [18].

Despite this, there are currently no robust and usable ways to detect app collusions. Existing solutions have large number of false negatives, false positives, or they are infeasibly difficult to implement. The number of possible combinations of N apps is N^N, and Google Play Store alone is reported to host more than 2.6 million apps [19]. Most proposed solutions therefore apply very aggressive filtering, causing only some malicious combinations to be included into analysis, and others to be reported as false negatives. Furthermore, most proposed solutions have a large number of false positives due to their inability to differentiate collusion from legitimate collaboration. Some have experimented with checking apps manually to determine their intentions [18], but this demands even more aggressive filtering. Therefore, app collusion remains an open research challenge.

Furthermore, existing literature on the topic has significant limitations. Some influential work in the field is now outdated and has not been updated. Many authors have defined app collusion for themselves in ways that are simpler to detect but not applicable in the real world, often including large amounts of legitimate apps. Some published information also has credibility issues, either dismissing prior work by claiming that collusion is a new idea, or claiming that they have solved the problem without providing sufficient evidence.

This report provides an overview of the research domain of Android app collusion as follows: Section 2 discusses the nature and definitions of app collusion, Section 3 outlines the methods that can be used for colluding on Android, Section 4 describes known examples of colluding apps, and Section 5 reviews approaches that have been taken to collusion detection, and their limitations.

2 Description and definition of app collusion

Consider section preview

The Oxford English Dictionary defines collusion as a “Secret agreement or understanding for purposes of trickery or fraud; underhand scheming or working with another; deceit, fraud, trickery” [20]. Asăvoae et al. [21] define collusion for the case of Android apps as the situation where several apps are working together in performing a threat. According to these definitions, app collusion must have the following three properties:

1. Colluding apps must be working together secretly. Conversely, apps working together in collaboration is a common and encouraged practice when such collaboration is well documented [22, “Interacting with Other Apps”].

2. All colluding apps must be in agreement. A distinctly different but related concept is the “confused deputy” attack [23], where one app mistakenly exposes itself to other installed apps.

3. Colluding apps must have malicious intentions. The intentions of Android app collusion would then be to violate one of the security goals of Android, which are defined in [8] as:

   (a) to protect app data, user data, and system resources (including the network), and
   (b) to provide app isolation from the system, other apps, and the user.

It is important to note that the goal 3(b) is not to enforce isolation, but to provide isolation when required. As such, apps working together does not violate goal 3(b) in itself, but it would be a collusion if apps worked together to break isolation with some other non-content app, the system, or the user.

many things affect, including non technical

difficult to distinguish

alternative definition [24]

3 Methods for colluding

By default, all Android apps run in separate sandboxes [8, “Application security”], which are based on user separation by the Linux kernel, and enhanced by SELinux and `seccomp` [8, “Application Sandbox”]. By default, all communication between these sandboxes is blocked, but apps can open certain communication channels or prevent being separated into different sandboxes altogether. Some channels, so-called overt channels, are designed to be used by apps to communicate, while others, so-called covert channels, utilise functionalities which were originally intended for other purposes.

3.1 Overt channels

The Android OS has several channels designed for inter-app communication, which are described on pages “Application security” and “Application Sandbox” of [8].

Apps published by the same entity may share a sandbox. In this case, there are no restrictions for their communication, meaning that these apps can use any of the traditional UNIX-type mechanisms, including the filesystem, local sockets, or signals.

When apps are running in different sandboxes, the Linux kernel prevents these sandboxed apps from accessing any processes or files other than their own. In older Android versions, only Linux discretionary access control was used, allowing apps to make their files world-accessible, which was used internally by shared preferences – an inter-app communication method discussed by many works [25, 21]. Newer versions of Android forbid this using SELinux mandatory access control rules, meaning that shared preferences are longer relevant for this purpose. Apps can still use any file-based communication methods when they have permission to access the external storage, but this way users would be notified that such communication may take place when the Android system prompts them for permissions.

However, Android also provides a method for apps to communicate without any user-granted permissions or visibility. This is enabled by a remote procedure call mechanism called binder. Any app can send messages to the binder arbitrarily, but other apps must explicitly start listening and accept incoming communications. The Android platform provides three main ways to do this:

- Services [22, "Services overview"]: Apps may start services, which can provide interfaces that are directly accessible using binder.
- Intent filters [22, "Intents and Intent Filters"]: Intents are simple message objects that represents an intention to do something. Apps may ask some part of them to be executed when an intent with certain properties matching their filter is initiated.
- ContentProviders [22, "Content providers"]: Apps can define ContentProviders to expose some of their data.

The binder provides an easy way for apps to communicate with each other, promoting openness and allowing a separation of concerns. Examples include apps using an intent to ask the camera app to take a photo instead of asking camera control permission, and communication apps allowing other apps to share data through itself. Since binder has a well-defined interface, information flow through it could be monitored.

3.2 Covert channels

In addition to the overt channels, many covert channels have been discovered. Marforio et al. [26] propose a classification of communication channels based on whether application level APIs, OS native calls, or hardware functionalities are utilised. Al-Haiqi et al. [27] describe categorising covert channels as either timing or storage channels. However, neither of these categorising approaches provide clear boundaries in all cases nor cover all possible covert channels. This section provides some examples of covert channels in Android.

Schlegel et al. [14] show that any application can change the vibrate setting and use intent filters to be notified of changes to that setting without requiring specific permissions, therefore demonstrating that apps can create an information channel using the vibrate setting. Similarly, the volume setting can be used. While apps cannot subscribe to be notified when the volume is changed, and have to manually check this setting, it has the advantage of having 8 different states, as opposed to the vibrate setting, which is a boolean value. Both of these channels are invisible to users, as long as data transmission does not coincide with audio playback or receiving notifications.

Marforio et al. [26] describe how data could be exchanged between colluding apps by modifying and monitoring the number of threads, processor usage, and free space on the filesystem. However, the APIs they used have since been deprecated [28]. Reviewing the current Android documentation with their ideas in mind suggests that alternative APIs might allow similar attacks to be mounted on recent versions of Android. For example, free disk space can be queried through the `StatFs#getAvailableBlocksLong()` API on latest Android versions [22]. A proof of concept could be created to verify the viability of this channel.

The system load can also be measured indirectly to transmit information, as described by Marforio et al. [26]. In this scenario, the transmitting app modulates data payload by varying the load on the system. The receiving app then repeatedly runs a CPU-intensive computation and measures the time it takes to complete, in order to infer whether or not the transmitting app was loading the system. This approach was shown to work even when the receiver is a JavaScript code running in a browser, and not an installed Android app.

Another approach is presented by Al-Haiqi et al. [27], where one app utilises the vibration motor to transmit data, and another app uses the accelerometer readings to receive that data. This is further developed by Qi et al. [29], who propose covert channels based on user behaviour. Instead of using the vibration motor, a transmitting app could prompt the user to move their phone in certain ways, for example, by posing as a rally game where the user needs to turn their phone at specific times based on a track generated by the malicious app.

Add a subsection conclusion paragraph

4 Examples of Android app collusion

Researchers have proposed several hypothetical and proof-of-concept collusion examples, and one example of app collusion has also been discovered in the real world. A common example of app collusion follows the pattern shown in Figure 1, and proceeds as follows:

(1) APP_A obtains some private information.

(2) APP_A transmits the information to APP_B using some overt or covert channel.

(3) APP_B exfiltrates the information over the internet.

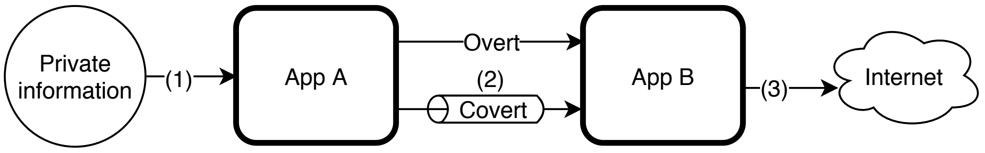

Figure 1. Structure of a basic app collusion example.

An early example of this kind of collusion was described by Schlegel et al. [14]. In their case, in place of the APP_A was an app called Soundcomber, which obtained private information using the microphone. To avoid detection, Soundcomber did not have permission to access the internet, instead they proposed to use a second app to exfiltrate the information. A similar hypothetical example is also described by Asăvoae et al. [21], where the APP_A would be a contacts app with READ_CONTACTS permission, and the APP_B would be a weather app with INTERNET permission. However, no examples following this pattern of collusion have been reported in the real world.

There is one known example of app collusion in the wild reported by Blasco et al. [30], and it has a different structure than described above.

It is a set of apps from different vendors, all of which include a library known as the MoPlus SDK (Figure 2). The collusion was noticed after a manual review of apps that had already been previously categorised as potentially harmful, and were capable of accessing shared preferences files from different apps. The MoPlus SDK was already known to contain remote-control capability, and is now confirmed to also collude between different instances of itself.

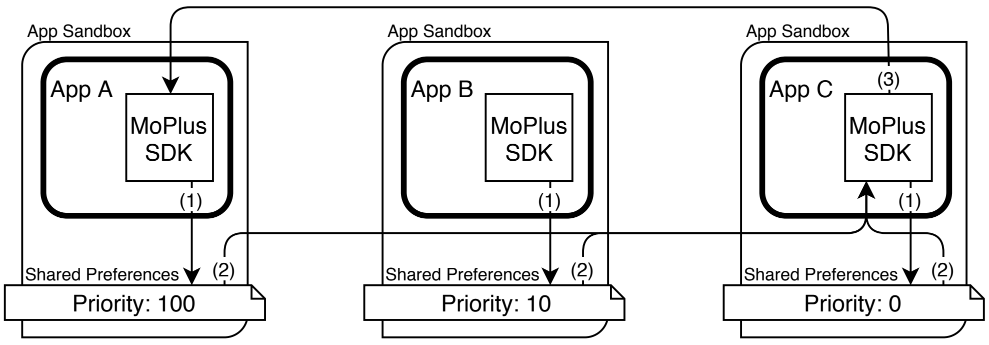

Figure 2. Collusion between instances of MoPlus SDK [30]

The MoPlus SDK may be embedded into many different apps, with 20 such being identified by [30]. It can then open a local HTTP server allowing the attacker to abuse any permissions given to its host app. With potentially many instances of the MoPlus SDK running on the same device with different permissions, they would work together to guarantee that only the instance with highest level of permissions is the one that starts accepting remote commands (Figure 2):

(1) Each instance stores a priority value based on the permissions it has
(2) Each instance queries all other instances for their priorities
(3) Instance with the highest priority is called

No sensitive information is exchanged between the apps, but still all three properties of app collusion are present.

5 Existing methods for detecting collusions

Attempts with various scopes and outcomes have been made to detect app collusion. Results range from computationally feasible but very coarse filters to more accurate but very expensive approaches.

5.1 Based on permissions/interfaces

The simplest approach is to base the analysis on statically extracted features of apps, such as the set of permissions declared or the set of Android APIs imported. Only considering such features makes it feasible to scan large amounts of apps. However, the obtained results only provide a coarse estimation of app collusion.

One example by Asavoae et al. [15] is a filtering method based on the API calls and declared permissions of apps. They define collusion as a pair of apps that satisfy these three conditions:

C1 First app declares a permission giving access to sensitive information.

C2 Second app declares a permission giving app the ability to send information to the outside world.

C3 Finally, on some channel the first app must be able to send and second app able to receive. As channels they consider Intents (each action separately), External storage, and shared preferences (each file separately; but now deprecated as discussed in Section 3.1).

It is not clear from their description, but can be assumed, that they further ignore apps that match both criteria C1 and C2, since otherwise their filter would detect any pair of apps that can communicate using one of these channels. They claim that while not all apps flagged by this approach are necessarily colluding, all colluding apps are flagged by this filter. However, it must be noted that in reality they only consider a narrow subset of app collusions, and, for example, the MoPlus SDK would be labelled as benign by this filter.

Chen et al. [13] propose a similar approach. First, they notice that exists a machine learning algorithm that, given the set of all API calls in an app, can predict whether the app is malicious or not. Only a single overt channel is captured for communication between apps, as their work only considers intents. For each app they detect which intents it can send and receive, and then they group together apps that could communicate with each other. To classify, whether a group of apps is benign or malicious, they pass the union of all API calls used by the apps in the group to the aforementioned machine learning algorithm.

Neither of these approaches can detect whether apps communicate with each other, nor what kind of information is exchanged between them, since they use no information about the order of operations within apps. Additionally, they both consider only a limited set of overt channels.

5.2 Based on control flow analysis

A more accurate detection of collusion can be achieved when the code of apps is analysed to detect whether or not they exchange data with other apps. Two approaches have ben taken to that: model-based and runtime.

Asăvoae et al. [16] describe a model-based method for checking whether information theft through collusion exists within the set of possible control flows for an app. They propose disassembling apps and analysing Dalvic virtual machine (VM) instructions with their input and output objects. In all possible control flows, when one of predefined Android APIs returns an object, it is annotated with the initiating app ID and a boolean "sensitive". Whenever an object is used as an argument to an instruction, its sensitivity and app ID values are propagated to all resulting objects. Input set of apps is considered to be colluding if there exists a control flow in which an object marked sensitive is passed to one of predefined exporting Android APIs, so that the current app ID is different to the initial app ID from the object. They explore modelling apps with different levels of abstraction, including more direct modelling in which case the analysis takes longer, or even forever when the app code contains any loops, and a higher level of abstraction, which is less accurate, but guaranteed to provide an answer. They only consider intent based inter-app communication channel, and do not analyse native code components of apps.

Another approach is to modify the Android OS to track information flow at runtime. Enck et al. provide an example of this called TaintDroid [17], which can track information flows from sensitive sources to sensitive sinks system wide. Within the Java code of apps it adds annotations for each variable separately (by augmenting the Java VM), but for communication through Binder these are at message accuracy, in native system libraries with method call accuracy, and for disc access with file accuracy. TaintDroid cannot track information flow in native app components, which is solved by authors here by banning them entirely, breaking 5% of apps by some estimates. Use of annotations is similar to [16], but the amount of false positives is smaller, because the corresponding exit state must be reached for it to be reported. However, due to the limited granularity of annotations, their propagation can be overly conservative, which still results in false positives. False negatives are also possible, because annotations are not propagated through covert channels. Additionally, TaintDroid incurs a 14% CPU overhead.

5.3 Other

None of the aforementioned research has adressed detection of collusion that uses covert channels. Some work has looked into closing specific covert channels [28, 29], but it is not reasonable to assume that all covert channels can be found, especially since there exist known covert channels that have not been closed (Section 3.2).

Muttik [18] proposes augmenting approaches mentioned above with additional heuristic rules. He argues that colluding apps can also be detected using indirect signals, such as whether apps come from the same source, explicitly encourage co-installation, are frequently installed together, use similar libraries, etc. Furthermore, he notes that signals like publication date, app market and installation method can be used.

According to Muttik, McAfee has a product for defending against app collusion by combining methods described in Sections 5.1 and 5.2, abovementioned heuristics, and manual review and reverse engineering of apps. In 2016 McAfee claimed that their product is able to detect colluding mobile apps and stop any malicious combination of apps from running [31]. However, based on known information about the state of the art, including the limitations outlined above, this is unlikely to be entirely true.

6 Conclusion

Bibliography

[1] Android Open Source Project *et al.*, "Android security 2017 year in review," Mar. 2018.

[2] Awio Web Services LLC. (2018, Dec.) Browser & platform market share. [Online]. Available: https://www.w3counter.com/globalstats.php?year=2018&month=12

[3] StatCounter. (2018, Dec.) Operating system market share worldwide. [Online]. Available: http://gs.statcounter.com/os-market-share

[4] Android Open Source Project. (2019) The Android source code. [Online]. Available: https://source.android.com/setup

[5] AV-TEST GmbH, "Security report 2017/18," Jul. 2018.

[6] McAfee, "Mobile threat report," Apr. 2018.

[7] Android Open Source Project. (2019) Google Play Protect: securing 2 billion users daily. [Online]. Available: https://www.android.com/play-protect/

[8] ——. (2019) Security. [Online]. Available: https://source.android.com/security

[9] J. Edge. (2016, Aug.) Hardened usercopy. [Online]. Available: https://lwn.net/Articles/695991/

[10] P. Lawrence. (2017, Jul.) Seccomp filter in Android O. [Online]. Available: https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html

[11] S. Willden. (2017, Sep.) Keystore key attestation. [Online]. Available: https://android-developers.googleblog.com/2017/09/keystore-key-attestation.html

[12] L. Deshotels *et al.*, "SandScout," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, 2016. doi:10.1145/2976749.2978336

[13] H. Chen *et al.*, "Malware collusion attack against machine learning based methods: issues and countermeasures," in *Cloud Computing and Security*, X. Sun, Z. Pan, and E. Bertino, Eds. Cham: Springer International Publishing, 2018, pp. 465–477. doi:10.1007/978-3-030-00018-9_41

[14] R. Schlegel *et al.*, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proceedings of the Network and Distributed System Security Symposium, NDSS'2011*, Jan. 2011.

[15] I. M. Asavoae *et al.*, "Towards automated Android app collusion detection," *arXiv preprint arXiv:1603.02308*, 2016.

[16] I. M. Asăvoae, H. N. Nguyen, and M. Roggenbach, "Software model checking for mobile security – collusion detection in $\mathbb{K}$," in *Model Checking Software*, M. d. M. Gallardo and P. Merino, Eds. Cham: Springer International Publishing, 2018, pp. 3–25. doi:10.1007/978-3-319-94111-0_1

[17] W. Enck *et al.*, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 5:1–5:29, Jun. 2014. doi:10.1145/2619091

[18] I. Muttik, “Partners in crime: investigating mobile app collusion,” in *McAfee Labs threats report*. McAfee, Jun. 2016, pp. 8–15.

[19] Statista. (2018, Dec.) Number of available applications in the Google Play Store from December 2009 to December 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[20] “collusion, n.” in *OED Online*. Oxford University Press, Dec. 2018.

[21] I. M. Asăvoae *et al.*, “Detecting malicious collusion between mobile software applications: the Android^{TM} case,” in *Data Analytics and Decision Support for Cybersecurity*. Springer International Publishing, Aug. 2017, pp. 55–97. doi:10.1007/978-3-319-59439-2_3

[22] Android Open Source Project. (2019) Android developers. [Online]. Available: https://developer.android.com/

[23] N. Hardy, “The confused deputy: (or why capabilities might have been invented),” *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988. doi:10.1145/54289.871709

[24] M. Xu *et al.*, “AppHolmes: detecting and characterizing app collusion among third-party Android markets,” in *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. ACM Press, 2017. doi:10.1145/3038912.3052645

[25] S. Bhandari *et al.*, “Android inter-app communication threats and detection techniques,” *Computers & Security*, vol. 70, pp. 392 – 421, 2017. doi:10.1016/j.cose.2017.07.002

[26] C. Marforio *et al.*, “Analysis of the communication between colluding applications on modern smartphones,” in *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*. ACM Press, 2012. doi:10.1145/2420950.2420958

[27] A. Al-Haiqi, M. Ismail, and R. Nordin, “A new sensors-based covert channel on Android,” *The Scientific World Journal*, vol. 2014, pp. 1–14, 2014. doi:10.1155/2014/969628

[28] nn...@google.com. (2017, Mar.) Android O prevents access to /proc/stat. [Online]. Available: https://issuetracker.google.com/issues/37140047

[29] W. Qi *et al.*, “Construction and mitigation of user-behavior-based covert channels on smartphones,” *IEEE Transactions on Mobile Computing*, vol. 17, no. 1, pp. 44–57, jan 2018. doi:10.1109/tmc.2017.2696945

[30] J. Blasco *et al.*, “Wild Android collusions,” in *VB2016*, Oct. 2016.

[31] McAfee, “Safeguarding against colluding mobile apps,” May 2016.

Todo list

remove list of todos