Android Application Collusion Demystified

Fauzia Idrees Abro*, Muttukrishnan Rajarajan*, Thomas M. Chen*,
Yogachandran Rahulamathavan**

*City University London, UK, **Loughborough University London, UK

Abstract. Application collusion is an emerging threat to Android based devices. In app collusion, two or more apps collude in some manner to perform a malicious action that they are unable to do independently. Detection of colluding apps is a challenging task. Existing commercial malware detection systems analyse each app separately, hence fail to detect any joint malicious action performed by multiple apps through collusion. In this paper, we discuss the current state of research on app collusion and open challenges to the detection of colluding apps. We compare existing approaches and present an integrated approach to effectively detect app collusion.

1 Introduction

Android being the most popular platform for mobile devices is under proliferated malicious attacks. A recent threat is from app collusion; in which two or more apps collaborate to perform stealthy malicious operations by elevating their permission landscape using legitimate communication channels. Each app requests for a limited set of permissions which do not seem dangerous to users. However, when combined, these permissions have potentials to inflict a number of malicious attacks. Mobile users are generally unaware of this type of permission augmentation, they consider each app separately. Hence, their decision to install apps is thus limited in perspective due to unawareness of permission augmentation [1]. The main contributor of the app collusion is Android's Inter-Process Communication (IPC) mechanism. It supports the useful collaboration among apps for the purpose of resource sharing, however, it also introduces the risk of app collusion when the app collaboration is done with malicious intention.

Android implements sandbox and permission based access control to protect resources and sensitive data, however, being open source and developer-friendly architecture, it facilitates sharing of functionalities across multiple apps. It supports useful collaboration among apps for the purpose of resource sharing, however, cyber criminals exploit this to launch distributed malicious attack through app collusion [2].

Application collusion is possible with *Inter Process Communication (IPC)*, *covert channels* or system vulnerabilities. Malicious colluding apps are explicitly designed by cyber criminals by exploiting different methods such as developing app with same User ID. Such apps have more chances for a successful collusion attack. In some cases, mis-configured apps also participate in the collusion attack

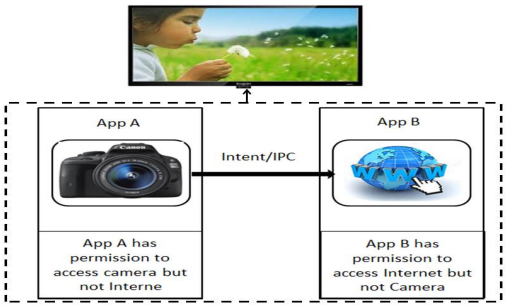

Fig. 1. Application Collusion Scenario

with a complete obliviousness of colluding app [3]. One of the collusion scenarios is illustrated in Figure 1: App 'A' has no permission to access the Internet, however it has permissions for camera. Similarly, App 'B' has no permission for the camera but can access the Internet. Assuming that the components of both apps are not protected by any access permission, they could collude to capture the pictures and upload on a remote server through the Internet.

Until recently, a small scale research is done on app collusion due to non availability of known samples of colluding apps for analysis [4]. Most of the existing works focus on identification of covert channels and development of experimental colluding apps. As a result of this innovative approach, the research on collusion gained momentum and there are now a few app collusion detection approaches available each with a limited scope. Despite the growing research interest, detection of malicious colluding apps has been a challenging task [5].

In this article, we give an overview of app collusion, potential risks and detection challenges. Aim of this article is to give an overview on the stealthy threat of app collusion and to repudiate the misconception about app isolation.

2 Android Primer

In Android, all applications are treated as potentially malicious. They are isolated from each other and do not have access to each others' private data. Each app runs in its own process and by default, can only access own files. This isolation is enforced with the sandbox, in which each app is assigned with a unique user identifier (UID) and own Virtual Machine (VM). App developers are required to sign the apps with a self-certified key. Apps signed with same key share User IDs and can use same sandbox [6].

Android app comes as .apk file, which contains the byte code, data, resources, libraries and a manifest file. Manifest file declares the permissions, intents, features and components of an app. The components that can be handled by an app are declared with intent filters.

System resources and user data are protected through permissions. Figure 2 illustrates the communication between apps in a sandbox environment. App 1

can use only those system resources and user data for which it has permissions. Similarly, app 2 is also limited to use certain resources. Although both apps have limited permissions to access the resources but through IPC, they are able to augment their permissions and get over-privileged access to system resources and user data.

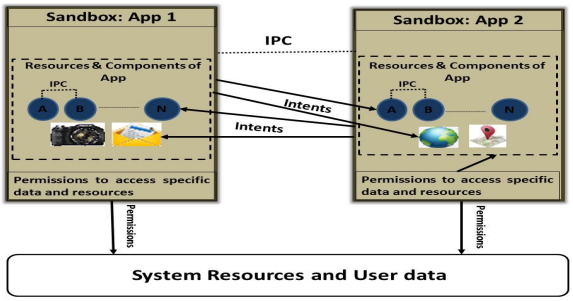

Fig. 2. Inter Process Communication

In this section, we provide an overview of IPC, which is a main facilitator of app collusion.

2.1 Android Security Architecture

Permission Mechanism. Permissions are used to restrict the access of system resources and user data on the device. Permissions are organized into permission groups so that they can be identified clearly with their capabilities and what resources or data they can use on the device [2]. Prior to installation, user are presented with a list of permissions. It is mandatory for the users to approve all the requested permissions as there is no option for the selection. Once granted, permissions remain valid unless the app is un-installed or updated [6].

There are four protection levels assigned to the permissions depending on the capabilities and possible security risks. These groups are: *Normal, Dangerous, Signature* and *Signature or system*. Android has a control system to certify if the app should be granted the permission governed by certain protection level [5].

Shared User ID. Android assigns a unique user ID to each app to ensure that it runs in its own process and can only access the allocated system resources. Android enforces app isolation by assigned user IDs, however, it also permits apps to share user IDs if they are developed with the same signature or certificate [3]. Apps with shared User IDs (shared Userid) can access each other's data and can run in same process, thereby limiting the effectiveness of isolation provided with user ID.

Components. Components are the basic modules that are run by apps or the system. There are four types of components: *Activities, Services, Content Providers* and *Broadcast Receivers*. *Activities* provide the user interface, each screen shown to a user is represented by a single activity. *Services* implement functionality of background processes which do not need user interface [4]. *Content Providers* provide database for sharing data among applications. Broadcast Receivers receive the notifications from system and other apps. It also sends messages to other components (activities or services).

Intents. Intents are messages used to communicate between the components of apps. These messages are used to request actions or services from other application components. Intents declare the intention to perform an operation [3]. It could be launching of an Activity, broadcasting Intent to any interested Broadcast Receivers or starting a background Service like music etc.

An intent contains mainly two parts: action and data. Action is the operation to be performed such as `BOOT_COMPLETED`, `ACTION_CALL`, `SMS_RECEIVE`, `ACTION_BATTERY_LOW` and `NEW_OUTGOING_CALL` etc. The data is a piece of information to operate on, such as a phone number, email address, web link etc.

Intents are of two types: *Explicit* and *Implicit*. *Explicit intent* specifies the component exclusively by class name. Explicit intents are mostly used by apps to start their components. *Implicit intent* does not specify a particular component by name. Apps with implicit intent only specify the required action without specifying particular apps or component [7]. System itself selects the app from device which can perform the requisite task. Implicit intents are vulnerable to exploits as they can combine operations of various applications, if they are not handled properly.

Sandboxing. Sandboxing isolates an app from other apps and system resources. Each app has a unique identifier and has access to the allocated System files and resources against the unique identifier. An app can also access files of othe apps that are declared as readable/wri-teable/executable for others.

Access Control Mechanism. In Android, the access control mechanism of Linux prevails. It controls access to files by process ownership. Each running process is assigned a UserID and for each file, access rules are specified. File access rules are defined for a user, group and everyone, thus granting permissions to read, write and execute on file.

Application Signing. Cryptographic signatures are used for verification of app source and for establishing trust among apps. Developers are required to sign the app to enable signature based permissions, and to allow apps from the same developer to share the UserID. A self-signed certificate of the signing key is enclosed into the app installation package for validation at installation time.

2.2 Covert Communication Channels

A covert channel is a stealthy mechanism to exchange information between apps in a manner that it cannot be detected [8]. There are two types of covert channels: *Timing* and *Storage*. Timing channels modulate the time spent on execution of some task or using some resource. Storage channels relate to modifying the data item such as configuration changes etc. Example of covert channel is sending user data to a remote server by encoding it as network delays over the normal network traffic [9]. Figure 3 depicts a covert channel, where a file of 20 bytes containing some data is sent through a normal communication channel. The file size is a covert information. This information might not be of any importance to the receiver but significantly valuable for the malicious party.

Covert channel typically exploit the shared resources to read, store and modify data as a medium for communication between two malicious entities. This type of information exchange is different from IPC based resource sharing. App collusion through covert channels is investigated by implementing high throughput covert channels in [2].

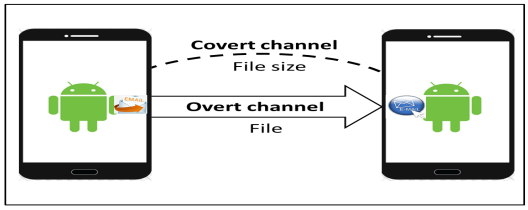

Fig. 3. Overt and covert channel

3 IPC related Attacks

Android security builds upon sandbox, application signing and permission mechanism. However, these protections fail if the resource and task sharing procedures provided through IPC are used with malicious intentions. In this section, we discuss the IPC related attacks on Android devices.

3.1 Application Collusion Attack

In application collusion attack, two or more apps collude to perform a malicious operation which is broken into small actions [2]. Each of the participating apps communicate using legitimate communication channels to perform the part assigned to them. Apps do not need to break any security framework or exploit the system vulnerabilities for carrying out a collaborative operation [5]. App

collusion helps in malware evasion as the current anti-malware solutions are not capable of simultaneously analyzing multiple apps.

3.2 Privilege Escalation Attack

In privilege escalation attack, an application with few permissions accesses components of more privileged application [10].This attack is prevalent in misconfigured apps mainly from the third party market. The default device applications of phone, clock and settings were also vulnerable to this attack [11]. Confused deputy attack is a type of privilege escalation attack. A compromised deputy may potentially transmit the sensitive data to the destination specified in the spoofed intent. Consider an app which is processing some sensitive information like bank details at the time of receipt of spoofed intent. It is likely that such an information may be passed on to the url or phone number defined in the malicious intent.

3.3 Intents related Attacks

Explicit and implicit intents may potentially assist in colluding attacks. Although, explicit intents guarantee the success of collusion between apps, implicit intents can also be intercepted by the malicious apps with matching intent filters. We discuss some of the known intents related attacks.

Broadcast Theft. A *public broadcast* sent by application is vulnerable to interception. As shown in Figure 4, a malicious app 'M' can passively listen to the public broadcasts while the actual recipient is also listening. If a malicious receiver registers itself as a high priority receiver in *ordered broadcasts* and receives the broadcast first, it could stop further broadcasting to other legitimate recipients. The *ordered broadcasts* are serially delivered messages to the recipients that follow an order according to the priority of receivers. Public and ordered broadcasts may cause eavesdropping and Denial of Service (DoS) attacks [4].

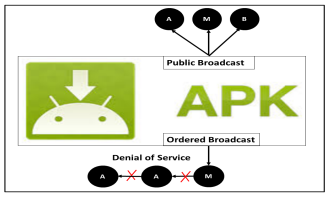

Fig. 4. Broadcast Theft Attack

Activity Hijacking. If a malicious app registers to receive the implicit intent, it may launch activity hijacking attack on successful interception of intent. With activity hijacking, a malicious activity can illegally read the data of the intent before relaying it to the recipient [2]. It can also launch some malicious

activity instead of the actual one. Consider a scenario, in which an activity is required to notify the user for the completion of certain action. The malicious user can falsely notify the user for the completion of uncompleted activity like *un-installation of app* or *transaction completed*.

Service Hijacking. If an exported service is not protected with permissions, it can be intercepted by an illegitimate service, which may connect the requesting app with a malicious service instead of the actual one [5]. In this attack, the malicious user hijacks the implicit intent which contains the details of service and start the malicious service in place of the expected one.

Implicit intents are not guaranteed to be received by the desired recipient because it does not exclusively specify the recipient. A malicious app can intercept an un-protected intent and access its data by declaring a matching intent filter [6]. This type of attack may be used for Phishing, Denial of Service (DoS) and component hijacking attacks are possible with unauthorized intent receipt.

Intent Spoofing. In Intent spoofing attack, the malicious app controls the unprotected public component of a vulnerable app. It starts performing as the deputy of the controlling app and carries out the malicious activity on behalf of the controlling app [3]. This type of attack is also known as *Confused deputy attack* as the deputies (victim apps) are unaware of their participation in the malicious activities. Figure 5 illustrates the confused deputy attack.

A malicious broadcast injection is also possible with spoofed intent when a broadcast receiver that is registered to receive the system broadcasts trusts an incoming malicious broadcast as a legitimate one and performs those actions which need system triggers.

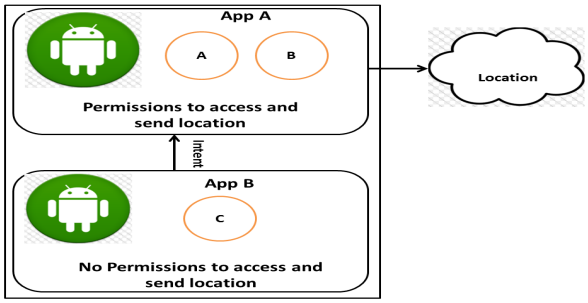

Fig. 5. Attack Scenario: Confused Deputy Attack

4 DETECTION OF COLLUDING APPS: OPEN CHALLENGES AND POTENTIAL MEASURES

Detection of app collusion is a very complex proposition. There are a number of challenges in designing a solution to detect the malicious colluding apps and there remain big question marks over efficacy of such solutions. This is the prime reason that we don't have a lot of reliable choices available for such detections.

4.1 Challenges

First challenge in detection is classification of IPC into benign and malicious groups. Android is an open source platform, which encourages resource sharing among apps by re-using the components. IPC is mainly used by apps to interact with different inter and intra components. The main problem is to distinguish between the benign collaboration and malicious collusion. Such a distinction is likely to come up with a cost of very high false positives. Keeping the false positive rate to lowest, is another problem.

Secondly, considering the substantial number of apps available in the Android market (more than 2 Million apps by Feb 2016), there is a difficulty of analyzing pairs of apps. It is computationally challenging and cost exorbitant to analyze all possible pairs of apps to detect the malicious collusion between sets of apps given the search space. Analysis of all possible app pairs of total of N apps would require N^2 pairs. Similarly, to analyze sets of three colluding apps, it would require to analyze N^3 apps. An eective collusion detection tool must be capable of isolating potential sets of apps and carrying out further investigations.

Another glaring challenge is the presence of a number of covert channels in the system. Detection of covert channels is an NP-hard problem as it would require monitoring of all the possible communication channels [12]. Covert channels are difficult to detect because they use overt channels for conveying stealthy information.

Lastly, known malicious colluding apps are not available for analysis. The non-availability of known samples of colluding apps, makes it difficult to validate the experiment results. Analysis and validation of collusion detection is a quandary, we need known samples of colluding apps to validate the detection method, but to find the samples, a reliable detection method is mandatory, which itself is not available in an authenticated form.

An effective collusion detection system must overcome the aforementioned challenges and encompasses an integrated solution. The detection of IPC based collusion have been recently proposed in a few research papers [12], [13], [14], and [15]. The proposed approaches have a number of limitations and the accuracy and efficiency of these methods is questionable due to non-availability of universally accepted dataset of malware colluding apps.

The solution proposed in [12] is to re-design the security model of Android system to mitigate the risk of collusion. However, this would involve a big cost

and complexity in re-writing the OS components and ensuring their compatibility and smooth functioning in conjunction with already available millions of apps in the Android market.

Another approach [13] is limited to the detection of collusion based on intents only. It analyzes the interaction of components through intent filters only and analyzes only two apps at a time. Currently, this approach suffers with a high false positive rate. It is a memory consuming approach which may not be feasible for mobile phones keeping in view the limited memory of phones. The extensive memory consumption may deteriorate the performance of device.

Similarly, [14] is also mainly based on intent messages. This approach faces the challenges of conventional rule based methods that are prone to evasion with obfuscation and reflection. Scalability is a major drawback of their approach.

Malware collusion detection tool [15] supports the latest API versions only, hence analysis of apps developed under earlier versions is not possible. Technical details of the tool are not available for performance verifications and evaluations. It generates a high number of false alarms mainly due to its reliance over information flows.

The detection of covert channels is still an under explored research area. [11] and [16] investigate the identification of covert channels. [11] has a limited scope of detecting covert channels related to shared resources only such as *reading of the voice volume*, *change of the screen state* and *change of vibration settings* etc. Similarly, [16] handles data flows only. However, it is possible to exploit these approaches for identification of other unknown covert channels.

4.2 Potential Measures

The complexity and challenges of collusion detection merit a hybrid framework. As a result of our analysis, we recommend an integrated approach for detection of app collusion. We also suggest that a covert channel may not be detected in isolation, but its existence may be realized whilst analyzing the IPC related security breaches. We argue that any mobile user downloads a limited number of apps as opposed to available millions of apps. A user cannot install millions of apps on a single device, hence, there is no need to analyze the millions of app pairs or triplets for possible collusion. On the average, a mobile user installs 20 to 30 apps. A system capable of analyzing 50^2 or 50^3 apps is sufficient for a common mobile user. This solution may also be augmented with a cloud based analysis engine if the number of concurrently analyzed apps is increased to 4, 5 or more. Cloud based analysis is an efficient and cost effective approach for high computational operations. We believe that adopting such an approach is essentially required to facilitate the identification of sets of colluding apps from a dataset of millions of apps.

Since permissions and intents facilitate inter and intra-app communication, analysis of these features has potentials to detect app collusion. Adding shared user IDs and publicly declared intents is also recommended as the collaborating apps may use same User IDs to make sure that the attack is successful.

The proposed system is shown in Figure 6. In first stage, apps are analyzed to identify those which share user IDs as they have more potentials to collude successfully. In second stage, permissions and intents are extracted and analysed for *source permission*, *source intent*, *sink permission* and *sink intent*. A pairwise communication mapping of apps is generated from the source and sink permissions and intents. The identified communicating pairs of apps are further analysed to check if their communication is limited to each other or more apps. The classifier stage is used to classify the app into colluding or non-colluding ones and users are notified for possible collusion. In the proposed approach, permissions and intents are grouped into four categories: *source permissions*, *source intents*, *sink permissions* and *sink intents*. *Source* permissions or intents are those that initiate some operation, whereas the *sink* permissions and intents are those which act upon to complete the required operation [1].

With additional policy refinements, the identified colluding apps can be classified into benign and malicious apps. This approach may be integrated with the methodologies proposed in [16] and [11] to monitor the data flow sources and sinks of IPC and tracking of shared resources. Information flow system proposed in [16] to monitor the data flow sources and sinks in IPC is a good trade-off in detecting the covert channels however, it lacks the tracking of shared resources. Mapping structure of [11] helps in tracking the shared resources used by two interacting apps.

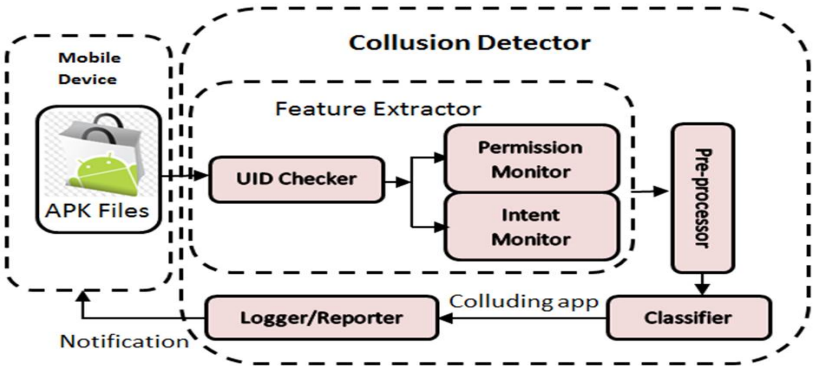

Fig. 6. Collusion Detection Model

Effective detection of app collusion requires monitoring of IPC and all possible covert communication channels: *shared resources* and *data flow sources and sinks*. The proposed framework integrated with Taintdroid [16] would be a good starter towards a comprehensive detection system.

5 RELATED WORK

IPC and intents have not been explored the way permissions have been investigated. Most of the existing IPC based studies focus on finding the IPC related vulnerabilities. [17] investigated the IPC framework and interaction of system components. [3] detects the IPC related vulnerabilities. [18] suggested improvement in ComDroid by segregating the communication messages into inter and intra-applications groups so that the risk of inter-application attacks may be reduced. [19] characterized Android components and their interaction. They investigated risks associated with misconfigured intents. [20] examined vulnerable public component interfaces of apps. [21] generated test scenarios to demonstrate the ICC vulnerabilities. [22] performs information flow analysis to investigate the communication exploits. [23] investigated intents related vulnerabilities and demonstrated how they may be exploited to insert the malicious data. Their experiments found 29 out of a total of 64 investigated apps as vulnerable to intent related attacks. All of these works focus on finding communication vulnerabilities, and none of them used IPC and intents for malware detection.

6 CONCLUSIONS

The concept of colluding apps has emerged recently. App collusion can cause irrevocable damage to mobile users. Detection of colluding apps is quite a challenging task. Some of the challenges are: distinction between the benign and malicious collaboration, false positive rate, presence of covert channels and concurrent analysis of millions of apps. Existing malware detection system are designed to analyse each app in isolation. There is no commercially available detection system which can analyse multiple apps concurrently to detect the collusion.

In this paper, we discussed the current state and open challenges to detection of colluding apps. To address the problem, we have proposed an integrated approach to detect app collusion. However, the complexity of problem merits a collaborative large scale investigations to mitigate a very large number of known and unknown communication channels between apps besides known IPC and covert channels. Our future work aims to validate the proposed framework on real colluding apps.

References

1. Karim O Elish, Danfeng Yao and Barbara G Ryder, (2015) On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions. Proc. of IEEE Mobile Security Technologies
2. Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, Srdjan Capkun (2012) Analysis of the communication between colluding applications on modern smartphones. Proc. of the 28th Annual Computer Security Applications Conference. pp. 51–60

3. Erika Chin, Adrienne Porter Felt, Kate Greenwood, David Wagner (2011) Analyzing inter-application communication in Android. Proc. of the 9th ACM conf. on Mobile systems, applications and services. pp.239–252
4. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, Bhargava Shastry (2012) Towards Taming Privilege-Escalation Attacks on Android. NDSS
5. Fauzia Idrees, Muttukrishnan Rajarajan (2014) Investigating the android intents and permissions for malware detection. Proc. of IEEE Wireless and Mobile Computing, Networking and Communications. pp. 354–358
6. Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steven Hanna, Erika Chin (2011) Permission Re-Delegation: Attacks and Defenses. USENIX Security Symposium
7. Fauzia Idrees, Muttukrishnan Rajarajantitle, Mauro Conti, Thomas M. Chen and Rahulamathavan Yogachandran (2017) PIndroid: A novel Android malware detection system using ensemble learning methods. Computers & Security. vol. 68, Elsevier, pp.36–46
8. Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, Sam Malek (2015) Covert: Compositional analysis of android inter-app permission leakage. IEEE Transactions on Software Engineering no. 9, pp. 866–886
9. Wade Gasior, Li Yang (2011) Network covert channels on the Android platform. Proc.of the Seventh Annual ACM Workshop on Cyber Security and Information Intelligence Research, pp. 61–67
10. Davi, L., Dmitrienko, A., Sadeghi, A. R., Winandy, M. (2010) Privilege escalation attacks on android. In Int. Conf. on Information Security. pp. 346-360
11. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi (2011) Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report:Technische Universität Darmstadt
12. Atif M Memon, Ali Anwar (2015) Colluding Apps: Tomorrow's Mobile Malware Threat. IEEE Security & Privacy, no. 6, pp. 77–81
13. Shweta Bhandari, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur (2016) Intersection automata based model for Android application collusion. Advanced Information Networking and Applications. pp. 901–908
14. Irina Asavoaeca, Blasco Jorge, Thomas Chen, Harsha Kumara , Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, Siraj Shaikh (2016) Towards Automated Android App Collusion Detection. arXiv preprint arXiv:1603.02308
15. Ravitch Tristan, Creswick E Rogan, Tomb Aaron, Foltzer Adam, Elliott Trevor, Casburn Ledah (2014) Multi-app security analysis with fuse: Statically detecting android app collusion. Proc. of the 4th Program Protection and Reverse Engineering Workshop. pp.4
16. W. , P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth (2010) TaintDroid: an information flow tracking system for realtime privacy monitoring on smartphones. Proc. of the 9th USENIX Conf on Operating Systems Design and Implementation. (OSDI'10), pp. 1–6
17. William Enck, Machigar Ongtang, Patrick McDaniel (2009) Understanding Android Security. IEEE Security and Privacy. 7:50–57
18. David Kantola, Erika Chin, Warren He, David Wagner (2012) Reducing attack surfaces for intra-application communication in Android. Proc. of second ACM workshop on Security and privacy in smartphones and mobile devices. pp. 69–80
19. Amiya Maji, Fahad Arshad, Saurabh Bagchi, Jan Rellermeyer (2012) An empirical study of the robustness of inter-component communication in Android. Int. Conf. on Dependable Systems and Networks. pp. 1–12

20. Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, Guofei Jiang (2012) Chex: statically vetting Android apps for component hijacking vulnerabilities. Proc. of conf. on Computer and communications security. pp. 229–240
21. Andrea Avancini, Mariano Ceccato (2013) Security testing of the communication among Android applications. Proc. of 8th IEEE International Workshop on Automation of Software Test. pp. 57–63
22. Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, Martin C Rinard (2015) Information Flow Analysis of Android Applications in DroidSafe. NDSS. pp. 1–16
23. Daniele Gallingani, Rigel Gjomemo, VN Venkatakrishnan, Stefano Zanero (2015) Practical Exploit Generation for Intent Message Vulnerabilities in Android. Proc. of the 5th ACM Conf. on Data and application security. pp. 155–157