On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions*

Karim O. Elish, Danfeng (Daphne) Yao, and Barbara G. Ryder
Department of Computer Science
Virginia Tech
Email: {kelish, danfeng, ryder}@cs.vt.edu

***Abstract*—Malware collusion is a new threat against Android application security. It refers to the scenario where two or more applications interact with each other to perform malicious tasks. Most existing solutions assume the attack model of a stand-alone malicious application, and thus cannot detect collusion. The objective of this position paper is to point out the need for practical solutions for detecting malware collusion. We show experimental evidence on the technical challenges associated with classifying benign Android inter-component communication (ICC) flows from colluding ones. We statically construct ICC Maps to capture pairwise communicating ICC channels of 2,644 real benign apps. We find that existing permission-based collusion-detection policies trigger a large number of false alerts in benign apps pairs.**

I. Introduction

Malicious software (malware) has been constantly evolving to evade detection. This evolution nature of malware was first shown in the seminal paper by Cohen [1] to cause endless arms-races between defenders and attackers. In the realm of smartphone application security, there exist a substantial amount of solutions for detecting malicious apps (e.g., Elish et al. [2], TaintDroid [3], RiskRanker [4], AppContext [5], Apposcopy [6], Zhang et al. [7], and Alharbi et al. [8]). Virtually all the existing solutions assume the attack model of a *single malicious application*. The detection techniques are focused on inspecting apps individually, through building behavioral models with features obtained from various static and dynamic program analysis.

The notion of malware collusion has been recently described in a few research papers [9], [10] as the next step that malware writers' may evolve into. *Collusion* refers to the scenario where two or more applications – written by the same malware writer – interact with each other to perform malicious tasks. The danger of malware collusion is that each colluding malware only needs to request a minimal set of privileges, which may make it appear benign under conventional screening mechanisms. Malware writers have strong incentives to write colluding malware.

In the Android system, Intent and inter-component communication (ICC) realize an encapsulated communication mechanism for passing messages between two applications, or between two internal components of the same application. Sending Intents through ICC channels is widely used in apps. For example, a restaurant search app may send an Intent to Google Maps app, so that Google Maps displays a map with the chosen restaurant's location. In this work, we investigate the technical challenges associated with distinguishing benign ICC flows from colluding ones. Inability to solve this problem may result in a high number of false alerts, i.e., misclassifying benign ICCs as collusion.

Most of the existing static ICC-based program analyses are for detecting vulnerable-yet-benign apps. For example, ComDroid [11] and Epicc [12] identify application communication-based vulnerabilities. CHEX [13] identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps, using data-flow-based reachability analysis. Amandroid [14] and DroidSafe [15] present a general information flow analysis framework for Android applications. All of the above solutions focus on individual app analysis, and none of them provides a complete solution against malware collusion attack.

XManDroid [9] provides a runtime monitoring of communication links between apps. It develops communication classification policies based on certain permissions combinations. XManDroid has limitations in distinguishing benign ICC flows from colluding ones, giving a high number of false alerts as demonstrated in our experiments. That solution is more appropriate for use to monitor a small number of apps already installed on a device, as opposed to screen a large number of apps (e.g., in Google Play Market) for possible collusion.

Because of the wide usage of ICC calls in benign apps pairs, accurate classification is quite challenging. We argue that practical solutions for detecting Android malware collusion needs to satisfy several requirements:

- To be able to characterize the context associated with communication channels with fine granularity,
- To define security policies for classification that minimize false alerts,
- To be scalable to a large number of apps (e.g., tens of thousands of apps).

The purpose of this position paper is to experimentally demonstrate the difficulties and technical challenges associated with applications collusion detection. We developed a static analysis tool to model the Intent-based ICC of Android applications. We constructed an *ICC Map* to capture pairwise communicating channels, and analyzed the ICC calls among 2,644 free popular applications from Google Play market.

*This work has been supported by DARPA APAC award FA8750-15-2-0076.

These apps have passed multiple screening tools and are considered benign. 84.4% of these benign apps have external ICC calls. Furthermore, we apply a set of classification policies in the existing XManDroid [9] collusion detection solution to these communicating apps. Our results show that these permission-based classification policies trigger a large number of false alerts in benign apps pairs.

To overcome the deficiencies in the existing work (namely, reducing the number of false alerts), we argue that there is a need for a more practical solution based on in-depth static flow analysis. We sketch a promising approach and give specific examples to show how to discover the *context* associated with benign ICC flows, and to formulate more fine-grained policies.

II. Malware Evolution and Attack Model

Malware collusion is a new malware generation attack that is very challenging to detect under the existing conventional screening techniques. A collusion attack occurs when malicious applications, likely written by the same adversary, collaborate to gain a set of permissions to perform malicious tasks. In malware collusion, each colluding malware only needs to perform a certain functionality, which may make it appear benign to evade the conventional detection tools. Hence, malware writers have strong motivation to write colluding malware to evade standard detection. Figure 1 depicts the application collusion threat model through Intent-based inter-component communication (ICC). Colluding applications may abuse system resources (e.g., sending spam SMS) or leak sensitive data. Colluding applications also may communicate indirectly, e.g., through shared files, or through covert channels as demonstrated in [10]. Detecting covert-channel based apps collusion remains an open problem.

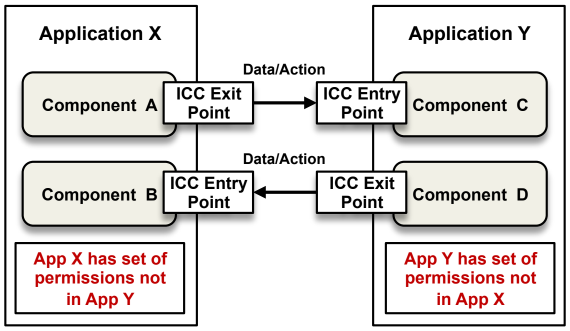

Fig. 1. Application collusion threat model via ICC.

In this paper, we focus our analysis on the Intent-based ICC channels which are standard communication channels in the Android. Colluding applications can use explicit or implicit Intents for inter-app communications. The explicit Intent specifies the target component/app, while the implicit Intent specifies the action string and will be delivered to any component that declares to handle this action.

In order for the attacker to make her/his apps collusion always succeed, she/he should use inter-app explicit Intents to guarantee that the Intent will be delivered to the correct component. Colluding apps based on inter-app implicit Intents may not always succeed since there is a chance that the implicit Intent will be wrongly delivered to a different component/app not the one intended by the attacker in her/his collusion scenario. This depends on the other apps installed on the mobile device and what are the actions they can handle. Therefore, using inter-app explicit Intents make the collusion more successful than using inter-app implicit Intents.

III. ICC Channel Characterization

The purpose of our analysis is to study the inter-app communication via ICC in Android and investigate whether it is commonly used or not. To achieve this, we define *ICC Map* to identify the group of communicating apps by capturing the ICC channels between them. In our ICC analysis, we focus on all Intent-based ICC APIs such as `startActivity(Intent i)`, `startService (Intent i)`, and `sendBroadcast(Intent i)`.

A. ICC Map Construction

The objective of the *ICC Map* is to capture and identify all the intra- and inter-app ICCs of an application.

Definition 1. ICC Map *is a directed graph $G(V, E)$ for app A. Each node $v \in V$ in G represents a component or action name, and each edge $e \in E$ in G represents an ICC API. There are two types of communication:*

- *Internal communication: component X is communicating with component Y, where both X and Y are internal components in app A.*
- *External communication: component X is communicating with component Y, where component X is in app A, and component Y is in app B.*

To determine if the app has an external communication, we get the list of components and their actions defined in the manifest of the app. Then, for each target component or action we find during our analysis, we match this target component with this list. If this target component is not defined as one of the internal components, we label it as an external component and infer that this application has external ICC.

For each app, we store the *ICC Map* information as a set S consisting of multiple four-item tuples $\{< ICCName_k, sourceComponent_k, targetComponent_k, typeOfCommunication_k >\}$, where

- $ICCName$ is the API name of ICC$_k$, such as `startActivity()` and `startService()`
- $sourceComponent$ is the name of the component which initiates the ICC$_k$ (exit point). It is a subclass of `Activity, Service, BroadcastReceiver`
- $targetComponent$ is the name of the component which receives the ICC$_k$ (entry point). It is a subclass of `Activity, Service, BroadcastReceiver`
- $typeOfCommunication$ is the type of ICC$_k$ communication, internal or external.

Our goal is to find all Intents used in the ICC APIs to identify the source and target components linked by the

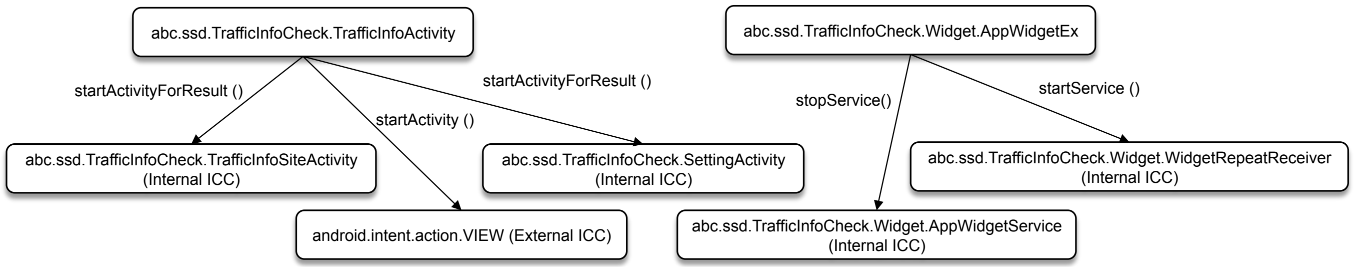
abc.ssd.TrafficInfoCheck.TrafficInfoActivity
startActivityForResult ()
startActivity ()
startActivityForResult ()
abc.ssd.TrafficInfoCheck.TrafficInfoSiteActivity (Internal ICC)
android.intent.action.VIEW (External ICC)
abc.ssd.TrafficInfoCheck.SettingActivity (Internal ICC)
abc.ssd.TrafficInfoCheck.Widget.AppWidgetEx
stopService()
startService ()
abc.ssd.TrafficInfoCheck.Widget.WidgetRepeatReceiver (Internal ICC)
abc.ssd.TrafficInfoCheck.Widget.AppWidgetService (Internal ICC)

Fig. 2. Partial ICC map for `abc.ssd.TrafficInfoCheck` application.

Intents. We utilize the data-flow analysis in Soot framework[1] to find the Intent object used in the ICC API. Specifically, we use data-flow analysis to construct the data dependence graph (DDG) of intra- and inter-procedural dependences to track the dependences between the definition and use of Intents in a given app. The DDG is a common program analysis structure which represents inter-procedural flows of data through a program [16]. The *ICC Map* graph is constructed based on the ICCMap information set S.

B. Example of ICC Map

Figure 2 shows an example of partial *ICC Map* for `abc.ssd.TrafficInfoCheck` app. It is an app to display information about the Japanese railway service. In this example, there are five ICC APIs. This map is constructed based on the ICCMap information set which has five tuples for this example. For instance, <startActivity(), abc.ssd.TrafficInfoCheck.TrafficInfoActivity, android.intent.action.VIEW, external> is one tuple in the *ICC Map* information set S. This tuple can be interpreted as there is an ICC call `startActivity()` from the source component (`abc.ssd.TrafficInfoCheck.TrafficInfo Activity`) to a target component which declares to handle this action `android.intent.action.VIEW`. This target component is an external component since none of the internal components in this app can handle this action. Only other components in other apps, which are declared to handle this action, will receive the request. Hence, we infer that this app is communicating with another app through this action.

IV. Experimental Evaluation

The purpose of this experiment is to show the difficulties and challenges in classifying benign Android ICCs from colluding ones. In particular, the objective of this experiment is to answer the following questions:

1) How often do benign apps perform inter-app communications with other apps?
2) How effective is the existing collusion detection solution (namely XManDroid) in terms of false positive rate?

We implemented a static analysis tool in Java to construct the *ICC Map* for a given Android app. We performed our ICC characterization study on 2,644 free popular real-world Android apps from Google Play market. These apps covering various application categories and different levels of popularity as determined by the user rating scale. We checked these apps using different tools such as VirusTotal[2] and Elish et al. [2]. All these tools indicate that these apps are benign.

A. ICC Analysis on Benign Apps

For each app, we construct the *ICC Map* to capture all types of communication that the app performs. In particular, we need to find whether the app is interacting with other apps or not. We found that **2,230 apps (84.4%) out of 2,644 apps perform external ICC** with other third party or built-in apps by either using implicit or explicit Intent. Table I summarizes our ICC analysis on 2,644 free popular apps.

For the apps with the external ICC (84.4%), **298 apps (11.3%) use explicit Intent ICC**. In particular, these apps send external ICC to other third party apps by specifying explicitly the name of the target component. On the other hand, **1,932 apps (73.1%) use implicit Intent ICC** (not specifying the target component) for communicating with other third party or built-in apps.

B. Collusion Detection Using XManDroid

XManDroid [9] is a runtime monitoring of communication links between apps, and it defines communication classification policies based on certain permissions combinations. XManDroid suffers from high false positives as it is acknowledged by its authors. We confirmed this property by evaluating a set of XManDroid's policies on randomly selected 20 benign apps pairs. We found these 20 benign apps pairs using our *ICC Map* analysis. Each pair uses direct explicit Intent ICC channels for the communication between the apps. We found that **11 out of 20 benign pairs of apps are misclassified as collusion according to XManDroid's policies**, a very high false positive rate (55%). Figure 3 shows the number of the 20 benign apps pairs that trigger alerts per XManDroid's policy.

The empirical results indicate that many Android apps are interacting with other third party or built-in apps through ICC channels. This ICC-based communication provides a potential opportunity for the attackers to develop malicious colluding apps by utilizing the ICC APIs in a similar way as in the communication between the benign apps. Therefore, it is challenging to develop a detection technique that can

[1]http://www.sable.mcgill.ca/soot

[2]https://www.virustotal.com

TABLE I
SUMMARY OF ICC ANALYSIS ON 2,644 BENIGN APPS. IMPLICIT AND EXPLICIT INTENT ICC ARE USED FOR COMMUNICATION BETWEEN APPS.

Action Used in External Implicit Intent ICC	**# of Apps (%)**
android.intent.action.VIEW	1870 (70.7%)
android.intent.action.SEND	943 (35.7%)
android.intent.action.DIAL	399 (15.1%)
android.intent.action.GET_CONTENT	275 (10.4%)
android.media.action.IMAGE_CAPTURE	231 (8.7%)
android.intent.action.CALL	158 (6.0%)
android.intent.action.PICK	139 (5.3%)
android.intent.action.SENDTO	122 (4.6%)
android.media.action.VIDEO_CAPTURE	62 (2.3%)
android.intent.action.DELETE	53 (2.0%)
android.intent.action.EDIT	48 (1.8%)
android.speech.action.RECOGNIZE_SPEECH	45 (1.7%)
android.intent.action.MEDIA_MOUNTED	42 (1.6%)
android.intent.action.INSERT	33 (1.2%)
android.intent.action.SEARCH	20 (0.8%)
android.intent.action.RINGTONE_PICKER	19 (0.7%)
android.intent.action.WEB_SEARCH	12 (0.5%)
android.intent.action.SYNC	3 (0.1%)
android.intent.action.ANSWER	2 (0.1%)
# of apps with **external implicit Intent ICC**	1932 (73.1%)
# of apps with **external explicit Intent ICC**	298 (11.3%)
Total # of apps with **external ICC**	2230 (84.4%)
Total # of apps with **Internal ICC**	414 (15.6%)

differentiate well between non-malicious ICC and malicious ICC. The reason is that most of the benign apps are interacting with other apps according to our empirical results, which makes it challenging to develop a detection mechanism.

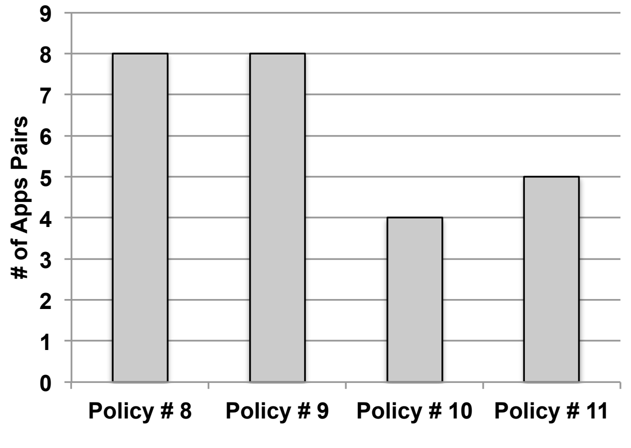

Fig. 3. Histogram showing the number of the 20 benign apps pairs that trigger alerts per XManDroid's policy [9]. Some apps pairs trigger alerts for more than one policy. XManDroid's policy can be found in [9].

V. COLLUSION DETECTION WITH CROSS-APP DATA-FLOW ANALYSIS

The above experiments show the challenges in distinguishing benign inter-app ICCs from colluding inter-app ICCs in practice, since most of the apps perform external communications. In this paper, we proposed *ICC Map* to statically characterize the inter-app ICC channels among the Android apps. This *ICC Map* does not provide a solution for apps collusion detection but it helps to identify pair or group of communicating apps. Thus, one can utilize this *ICC Map* to first identify the communicating apps. Then, a set of security policies can be defined and applied to differentiate between benign communicating apps and colluding ones.

There may be multiple ICC calls between two apps. The sensitivity of each ICC call may differ substantially and needs to be analyzed case-by-case. Permission-based policies cannot achieve this granularity. Inferring the ICC sensitivity requires in-depth data-flow analysis in both apps, detailing how the data is created, modified, and consumed. Policies based on such an analysis will be more fine-grained than permission-based policies, reducing false alerts on benign ICCs.

We argue the need for in-depth static flow analysis (e.g., definition-use relations) in both source and destination apps for collusion detection. The key is the discovery of the *context* associated with benign ICC flows. The discovery requires new static program analysis algorithms and data structures. Static analysis-based solution provides complete analysis coverage, and scalable to analyze large number of apps. Admittedly, static analysis-based solution in general is affected by obfuscated code and can not handle dynamic code loading. However, we argue that any successful collusion detection technique should be comprehensive, scalable, and not limited by the device's resources constraints. Hence, we advocate the use of static analysis-based solution for collusion detection because of the scalability and completeness provided by the static analysis.

Existing work on single-app classification showed that statically extracted features on user-trigger dependence, i.e., the degree of sensitive API calls having def-use dependence relations on user inputs[3], is very effective [2], [17]. Benign single apps have a high degree of user-trigger dependence, whereas malware – often performing activities under stealth mode – does not. Hence, it is conceivable to write similar user-trigger dependence policies for collusion detection, i.e., how much the app actively involves the user in implicitly authorizing inter-app ICC calls. Such a policy is under the hypothesis that benign ICC flows are intended and initiated by the users (at the source app), whereas colluding ones are not. As a future work, we plan to design and implement such policies to differentiate between benign inter-app ICC and malicious inter-app ICC.

Admittedly, all policies have their limitations and may be bypassed by sophisticated malware writers. Yet, the advantages of fine-grained policies are twofold: making the tool more

[3]User inputs may be onClick(), onItemClick(), etc.

usable by reducing false alerts, and creating bigger obstacles for malware to bypass.

VI. Conclusions and Future Work

In this position paper, we argue that it is very challenging to detect malware collusion. We demonstrate the challenges through experimental evidence. The experimental results indicate that many Android apps communicate with other apps through ICC channels, which makes it challenging to develop a detection mechanism without generating many false alerts as in XManDroid [9]. None of the existing solutions provides a complete solution against app collusion attack. An effective solution should be scalable to a large number of apps, and define policies for classification that minimize false alerts.

For future work, we plan to utilize our *ICC Map* for application collusion detection and define security policies to differentiate between benign inter-app ICC and colluding inter-app ICC. Our goal is to defend against the malware collusion attack in the Android system.

Acknowledgement

We thank our shepherd, William Enck and Adwait Nadkarni, and the reviewers for their insightful comments.

References

[1] F. Cohen, "Computer viruses theory and experiments," *Computers and Security*, vol. 6, pp. 22 – 35, 1987.

[2] K. Elish, X. Shu, D. Yao, B. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Computers & Security*, vol. 49, pp. 255–273, 2015.

[3] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 393–407.

[4] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: scalable and accurate zero-day Android malware detection," in *Proc. of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2012, pp. 281–294.

[5] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2015.

[6] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[7] H. Zhang, D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *Proc. of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2014.

[8] K. Alharbi, S. Blackshear, E. Kowalczyk, A. M. Memon, B.-Y. E. Chang, and T. Yeh, "Android apps consistency scrutinized," in *Proc. of the Extended Abstracts on Human Factors in Computing Systems*. ACM, 2014.

[9] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[10] C. Marforio, H. Ritzdorf, A. Francillo, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proc. of the 28th Annual Computer Security Applications Conference*. ACM, 2012.

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. of the 9th Int'l Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239–252.

[12] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. of the 22nd USENIX Security Symposium*, 2013.

[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting Android apps for component hijacking vulnerabilities," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 229–240.

[14] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. of the ACM Conference on Computer Communications Security (CCS)*. ACM, 2014.

[15] M. Gordon, D. Kim, J. Perkins, L. Gilhamy, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60, 1990.

[17] B. Wolfe, K. Elish, and D. Yao, "Comprehensive behavior profiling for proactive Android malware detection," in *Proc. of 17th International Information Security Conference (ISC)*, 2014.