Identifying Mobile Inter-App Communication Risks

Karim O. Elish, *Member, IEEE*, Haipeng Cai, *Member, IEEE*,
Daniel Barton, Danfeng Yao, *Member, IEEE*, and Barbara G. Ryder, *Member, IEEE*

Abstract—Malware collusion is a technique utilized by attackers to evade standard detection. It is a new threat where two or more applications, appearing benign, communicate to perform a malicious task. Most proposed approaches aim at detecting stand-alone malicious applications. We point out the need for analyzing data flows across multiple Android apps, a problem referred to as *end-to-end flow analysis*. In this work, we present a flow analysis for app pairs that computes the risk level associated with their potential communications. Our approach statically analyzes the sensitivity and context of each inter-app flow based on inter-component communication (ICC) between communicating apps, and defines fine-grained security policies for inter-app ICC risk classification. We perform an empirical study on 7,251 apps from the Google Play store to identify the apps that communicate with each other via ICC channels. Our results report *four times fewer warnings* on our dataset of 197 real app pairs communicating via explicit external ICCs than the state-of-the-art permission-based collusion detection.

Index Terms—Android ICC, inter-app analysis, malware collusion, static analysis, risk assessment

1 Introduction

Most current Android malware detection techniques assume that malware apps are stand-alone and independent [1]. Thus, they only support the analysis of individual app programs. Malware collusion is a new emerging attack model, where two or more malicious apps work together to achieve their attack goals, while each individual app may appear benign under conventional detection. Because of the flexible inter-app communication infrastructural support in Android, namely inter-component communication (ICC), writing colluding malware seems to be a natural next step for malware writers to evade detection.

Performing program analysis on multiple apps for detecting collusion is challenging and has not been systematically reported in the literature [2]. Existing individual app-screening solutions are inadequate. Virtually all existing ICC-based program analyses are for detecting vulnerable-yet-benign apps (e.g., due to inexperienced developers). For example, CHEX [3] identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps. ComDroid [4] and Epicc [5] identify application communication-based vulnerabilities and describe two main categories of abuses (i.e., intent stealing and intent spoofing). These works address the confused deputy attack, where a malicious app exploits vulnerable component interfaces of a benign app. However, malware collusion has a different attack model, where all colluding apps are written by malicious developers and share common attack goals. Existing analyses on ICC (e.g., Appscopy [6], IccTA [7], and Epicc [5]) are not designed to provide *end-to-end pairwise* ICC flow analysis. They cannot be directly applied for collusion detection.

In this work, we point out the need for *end-to-end data-flow analysis* across application boundaries for characterizing behaviors of a chain of apps of length l. For app pairs ($l = 2$), end-to-end analysis requires inspecting data flows through inter-app communication and collectively analyze the flows for security. Compared to individual app analyses, an end-to-end analysis can provide a complete and in-depth report of how sensitive data is generated, passed, and consumed spanning multiple communicating programs. However, such an analysis has new security and scalability requirements, namely i) how to reduce false alerts and ii) how to achieve scalability: which are explained next.

- *[Flow-level feature extraction]* The pairwise app analysis needs to characterize the context associated with communication channels with fine granularity. Failure to do so will result in a high number of false alerts. For example, XManDroid is an existing (runtime) solution for Android collusion detection [8]. It specifies *classification policies* on inter-app communications. The policies are based on permissions that the source and destination apps request at the time of installation. Permission-based policies are

- *K.O. Elish is with the Department of Computer Science, Florida Polytechnic University, Lakeland, FL 33805. E-mail: kelish@floridapoly.edu.*
- *H. Cai is with the School of Electrical Engineering and Computer Science, Washington State University College of Engineering and Architecture, Pullman, WA 99163. E-mail: hcai@eecs.wsu.edu.*
- *D. Barton, D. Yao, and B.G. Ryder are with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. E-mail: danielb5@vt.edu, {danfeng, ryder}@cs.vt.edu.*

Manuscript received 26 May 2018; revised 5 Nov. 2018; accepted 11 Dec. 2018. Date of publication 24 Dec. 2018; date of current version 3 Dec. 2019. (Corresponding author: Karim O. Elish.)
Digital Object Identifier no. 10.1109/TMC.2018.2889495

coarse-grained. Hence, they are limited in distinguishing benign ICC flows from colluding ones.

- *[Scalability]* We need to provide scalable solutions with minimum complexity to vet a large number of app pairs. The complexity of straightforward pairwise inter-app ICC analysis (such as in ApkCombiner [9]) is $O(n^2m|E|)$, where n is the number of apps, $|E|$ is the maximum number of edges in an app's graphical representation, and m is the maximum number of inter-app ICC calls between any pairs of apps.

We present *a scalable and effective static program analysis algorithm* to screen apps for possible collusion. Our approach statically analyzes the sensitivity and the context of each inter-app ICC-based flow between two communicating apps. Each inter-app ICC-based flow has a varying degree of sensitivity, depending on the type of data it carries or action it invokes/requests. Thus, we perform comprehensive static data-flow analysis across both apps that infers the inter-app ICC sensitivity, detailing how the data is created, modified, and consumed. Our detection policies are based on this in-depth analysis and are more fine-grained than permission-based policies, reducing the number of false alerts on benign inter-app ICC calls as demonstrated by our experiments.

We describe a method that efficiently analyzes cross-app data-flows. The method detects important data-flows across app pairs and extracts behavioral features from the flows. These flows may lead to collusion-based data leaks and security abuses, as well as activity and service hijacks due to vulnerabilities such as exposed exported components.

Our technical contributions are:

- We present an end-to-end program analysis for app pairs (i.e., app-chain length $l = 2$) and demonstrate its application in collusion prediction. We store properties associated with each inter-app ICC flow, largely reducing the analysis complexity. Our algorithm computes and stores the ICC entry and exit points of an app only a constant number of times. Our detection has a complexity of $O(n^2m + n|E|)$, where n is the number of apps, $|E|$ is the maximum number of edges in an app's data-dependence graph (DDG), and m is the maximum number of inter-app ICC calls between any pairs of apps.
- We design a flow-level risk classification approach for computing risks of inter-app ICCs. The risk level indicates possible collusion-based data leak and system abuse threats in a pair of apps. Our classification is based on features extracted from inter-app ICC flows. The flow-level features are fine-grained. They cover data dependence, resource access, and protection properties of cross-app flow paths.
- We perform an empirical study on 7,251 apps from Google Play store, and found 197 app pairs communicate using explicit intent-based ICCs. We extracted inter-app flow-level features from these 197 real app pairs and classified their risk levels. We found that 19.7 percent of the app pairs exhibit risky inter-app ICC behaviors, such as when external explicit ICCs are not initiated by valid user triggers (in the source app) or when external target component (in the target app) invokes a critical operation. For comparison, we also classified them with permission-based policies [8]. Our results suggest that our inter-app ICC risk classification method generates four times fewer false warnings compared to the permission-based policies. We perform further case studies on sample app pairs to validate our findings and show the advantages of end-to-end flow analysis in *differentiating* varying security risk levels of an app in various contexts when communicating with different apps, where both single-app and existing inter-app analyses would produce false alerts and/or miss important risks.

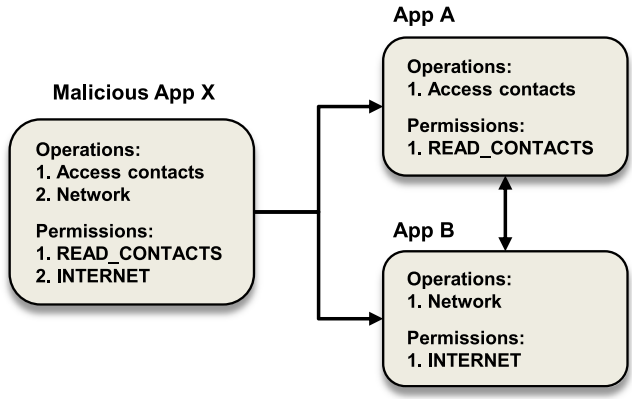

Fig. 1. An example of permissions and operations being split between colluding applications.

The proposed technique offers a *proactive* solution against inter-app collusion vulnerabilities, and demonstrates the need for *end-to-end* flow analysis of app chains for Android security.

We envision that it can be used by the app store to perform massive screening against *potential* collusion, as well as by end users to do so with respect to existing apps on their device.

Malware collusion is a relatively new class of security threat, thus, there is no existing benchmark suite for app collusion. Therefore, we use real-world Android apps to evaluate our technique.

2 Attack Model

With malware collusion, privileged Android APIs necessary for completing attacks can be distributed into multiple applications as shown in Fig. 1. The colluding apps communicate directly or indirectly with each other in order to combine their privileges and perform malicious tasks that threaten data privacy and/or system integrity.

Computation in many existing app scoring systems is based on properties of how privileged operations are used in a single app (e.g., [10], [11], [12], [13], [14], [15]). Distributing privileges across multiple apps results in apps appearing less risky, substantially weakening the power of these detection systems.

There are two main categories of collusion-induced malware abuse:

- *Collusion for data leak*: app X has the access to some sensitive data (e.g., contact information), but app X does not request the capability to access the network,

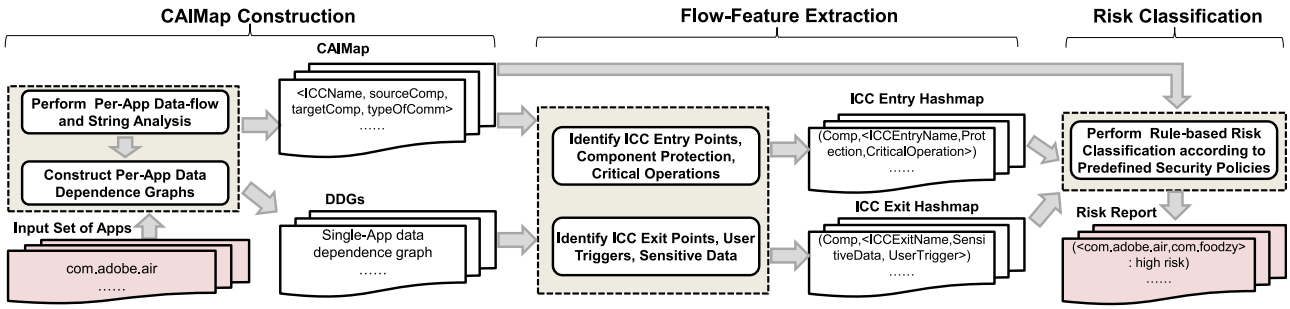

Fig. 2. Our inter-app ICC analysis workflow, with a set of apps as input (bottom left) and a risk report (bottom right) as output after three major phases (as labeled atop).

in order to appear innocuous to the conventional stand-alone app screening. App Y (written by the same malware author) has the permission to access the network, but not the access to sensitive data. App X asks app Y to send the sensitive data to the remote attacker through ICC.

- *Collusion for system abuse*: apps X and Y work together to send spam SMS messages. App X prepares the spam SMS messages, and requests app Y to send them. App X does not need to request permission for sending SMS.

Colluding apps may communicate indirectly, e.g., via shared files, or through covert channels as demonstrated in [16]. Marforio et al. [16] thoroughly measured the efficiency of different overt and covert channels for apps collusion, but their work does not provide any concrete defense mechanisms. Detecting covert-channel based malware remains an open problem.

In this work, we focus on intent-based ICC channels, which are standard communication channels in Android. Our goal is to analyze the threats posed by colluding apps via the explicit intent-based ICCs, as they threaten both data privacy and system integrity in Android.

Our inter-app ICC analysis can be used for a number of security analyses (e.g., to detect app collusion and vulnerability). For app collusion analysis, our inter-app ICC analysis provides an effective means to capture sensitive inter-app ICC-based flows and to identify risky app pairs. For app vulnerability analysis, our inter-app ICC analysis provides more precise pairwise vulnerability analysis than individual state-of-the-art solutions (e.g., ComDroid [4] and Epicc [5]). These solutions assume that any unprotected public component is vulnerable regardless if there is a path from the public component to the critical operations or not. In contrast, we perform more in-depth end-to-end flow analysis across multiple apps and inside the public vulnerable components to check for the existence of such a path. Hence, this will reduce the false alarms when no path exists.

3 Cross-App Flow Analysis

To address security threats across multiple Android apps, we develop an inter-app ICC analysis, called *Cross-App ICC Map* (*CAIMap*), which underlies our *end-to-end flow analysis* hence enables our classification of security risks. The flow process, as shown in Fig. 2, involves the following three key operations (phases): CAIMap construction, Flow-feature extraction, and Risk classification, as described below.

In the first phase, we adopted static flow analysis for each individual app.

This static analysis first computes intraprocedural data dependencies [17] (i.e., based on local reaching definitions and live uses), followed by an Android-specific interprocedural control-flow analysis which connects those intraprocedural dependencies of individual methods both within each component of the app, via call and return edges, and across different components, via intent-based ICCs. For both of the intra- and inter-component cases, the analysis considers interprocedural control flows due to threading (including those through uses of `AsyncTask` [18]) as well as event handling and life-cycle callbacks [19]. Thus, the result of this analysis is an interprocedural data-dependence graph (DDG)—intraproc-edural DDGs connected through interprocedural control-flow edges.

In addition, a string analysis based on local string-constant evaluation [5] is employed to identify the target components of explicit ICCs hence produce the *CAIMap* graph.

The pseudocode of our cross-app flow-analysis procedure is shown in Algorithm 1. The algorithm starts with the initializations of helper data structures (lines 1–4), and then constructs the *CAIMap* (lines 5–13). Next, the second and third phases are performed on *only* the linked app pairs indicated by *CAIMap* entries (lines 14–19) rather than on all possible pairs, where `getApp` returns the containing app of a given component. The two subroutines, *getOutFlows* (lines 21–25) and *getInFlows* (lines 26–31), extract ICC and flow features from an app by backwardly and forwardly traversing the data-dependence graph of that app, respectively (as detailed in Section 3.2). Given the flow features of a pair of apps, the risk level classification (lines 18–19) is performed by consulting the security policies we elaborate in Section 3.3 (as summarized in Table 1). Finally, the algorithm outputs the risk levels of all linked app pairs in *CAIMap* (line 20).

3.1 CAIMap Construction

Our ultimate goal of constructing the *CAIMap* is to identify pairs of apps that interact with each other and to model the interaction with respect to each apps behaviors. It is based on these behavioral interactions that our cross-app flow analysis hence collusion analysis is performed. In particular, our cross-app flow analysis is realized by analyzing Intent-based ICC calls across apps (e.g., `startActivity(Intent i)`, `startService (Intent i)`). Various ICC analysis has been proposed in prior works, yet most of them focus on statically determining the field values of Intent objects for more precise ICC linking [5], [20], [21], [22], [23] or better understanding

TABLE 1
Our Risk Classification Policies for Explicit Intent Inter-App ICC

	Source Component/App		Destination Component/App		
Policy	User Trigger	Access to Sensitive Data	Exposed Component*	Critical Operation	Inter-App ICC Risk Level
1	No	Yes	Yes	Yes	High
2	No	Yes	No	Yes	Medium
3	No	No	Yes	Yes	High
4	No	No	No	Yes	Medium
5	Yes	Yes	Yes	Yes	Medium
6	Yes	No	Yes	Yes	Medium
All others	–	–	–	–	No Risk

**Component is public and not protected by permission(s).*

ICC induced app behaviors through characterization [24] or visualizations [25], [26], [27]. Other works address security vulnerabilities within apps [21], [28], [29] or do not address in-depth cross-app data flows for collusion analysis as we focus on here.

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

(1) internal communication: component X calls component Y, where both X and Y are internal components in app A, and (2) external communication: component X calls component Y, where component X is in app A, and component Y is in app B.

For each app, we store the *CAIMap* as a set S consisting of multiple four-item tuples $<$*ICCName*, *sourceComponent*, *targetComponent*, *typeCom*$>$, where

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

Our *CAIMap* construction takes as input the source or bytecode of the app and its manifest. The output is a set of *CAIMap* information $<$ICCName$_k$, sourceComponent$_k$, targetComponent$_k$, typeCom$_k$ $>$ for each ICC$_k$.

Specifically, the source and target components of an explicit ICC is identified by first trying to parse the Intent object associated with the ICC call. If the relevant Intent object field is empty, we then analyze calls to APIs that set up an Intent object before it is passed to the ICC call (e.g., `setClass()`, `setComponent()` or `setAction()`, etc.) to retrieve the target component. For implicit ICCs, the target component, which can be any one that is declared to hand a specific action, is identified by parsing the apps manifest file according to Intent filters (e.g., matching Intent Actions). For both explicit and implicit ICCs, the source component is enclosing class/component of the API call that launches the corresponding ICC. Whether an ICC is internal or external is also computed from the apps manifest file, by determining the scope of the matched target component.

We use *hashmap* to store some information related to ICC *exit* and *entry* points. Specifically, we use two different *hashmaps* one for *exit* points and one for *entry* points, namely, `SourceAppICCExitHashMap` and `TargetAppICCEntryHashMap`. Each *hashmap* has multiple entries, where each entry consists of `ComponentName` as the key, and a tuple as the value. We next describe how ICC *exit* and *entry* points with their related information are extracted and stored in our *hashmaps*.

ICC Exit Points. We identify the ICC *exit* points, user trigger, and sensitive data through static analysis. The ICC exit points include all intent-based ICC APIs such as `startActivity(Intent i)`, `startService (Intent i)`, and `sendBroadcast(Intent i)`. *User trigger* refers to a user's input or action/event on the app. For example, the user's input may be text entered via a text field, while the user's action/event is any click on an UI element, such as a button. Relevant API calls in UI objects that return a user's input value or listen to user's action/event are defined as triggers. The *sensitive data* refers to the APIs that retrieve private data, such as `getAccounts()` and `getPassword(...)`. We perform *backward depth-first traversal* on the DDG of the app under analysis from each ICC exit point to check if it involves sensitive data or user trigger and to store this information in `SourceAppICCExitHashMap`.

Each entry in this exit-point map consists of `SrcComponentName` as the key, and a tuple $<$`ICCExitName, SensitiveData, UserTrigger`$>$ as the value. `(compX, <startService(Intent i), getDeviceID(), onClick()>)`, for example, represents one entry where `compX` is the component name that initiates the inter-app ICC call `startService (Intent i)` with sensitive data `device ID` as part of the intent `i` and `onClick()` as the user event to trigger this call.

ICC Entry Points. We identify the ICC *entry* points and critical operations also through static analysis. The ICC entry points include all components' entry points such as `onStart()` and `onCreate(...)`. *Critical operation* is an API call which refers to a function call providing system service such as network operations, file operations, telephony services in the app. Additionally, we analyze the target app's manifest.xml file to get the information about the protection of each component (i.e., the permission(s) defined to access the component). We perform *forward depth-first traversal* on the DDG of the app under analysis from each ICC entry point to find any critical operations and store this information in `TargetAppICCEntryHashMap`.

Algorithm 1. Inter-App ICC Flow Risk Analysis

Input: *AppSet*: a set of Android apps
Output: *risk_levels*: a map from app pair to risk level (as high, medium, or no) for all explicitly linked app pairs in *AppSet*

```
1:  CAIMapInfo ← ∅ //a list of entries each being a tuple
    < ICCName, srcComp, tgtComp, typeCom >
2:  DDGmap ← ∅ //a map from the package name of an app
    to its data-dependence graph
3:  outFlows ← ∅ //a map from source component name to a
    tuple < ICCExitName, SensitiveData, UserTrigger >
4:  inFlows ← ∅ //a map from target component name to a tuple
    < ICCEntryName, ComProtection, CriticalOperations >
    /* CAIMap construction */
5:  for each app in AppSet do
6:    DDG ←construct the data dependence graph of app
7:    add (app, DDG) to DDGmap
8:    for each explicit ICC icc, in component srcComp, of app do
9:      tgtComp←find the target component of icc in AppSet
10:     if tgtComp == null then continue
11:     typeCom ← external
12:     if tgtComp is in app then typeCom←internal
13:     add < icc, srcComp, tgtComp, typeCom > to CAIMapInfo
14: for each entry e in CAIMapInfo do
    /* Flow-Feature extraction */
15:   if e.typeCom == internal then continue
16:   getOutFlows(DDGmap[getApp(e.srcComp)], outFlows)
17:   getInFlows(DDGmap[getApp(e.tgtComp)], inFlows)
    /* Risk classification */
18:     risk_level ←match the flow features in outFlows
      [e.srcComp] and inFlows[e.tgtComp] against predefined
      security policies
19:       add (< getApp(e.srcComp), getApp(e.tgtComp) >,
          risk_level) to risk_levels
20: return risk_levels
21: procedure getOutFlowsDDG, outFlows
22:   ICCList ← retrieve ICC-invoking call sites from DDG
23:   for each icc, from component comp, in ICCList do
24:       < sensData, userTrigs > ← collect backward-
          reachable sensitive API calls and user triggers
          from icc on DDG
25:       add (comp, < icc, sensData, userTrigs >) to outFlows
26: procedure getInFlowsDDG, inFlows
27:   CompEntList ← extract all component entries from
      DDG
28:   for each compEnt of component comp in CompEntList do
29:     crtOPs ← collect forward-reachable critical opera-
        tions from compEnt on DDG
30:     comProt ← check component protection for comp
31:     add (comp, < compEnt, comProt, crtOPs >) to inFlows
```

Each entry in this entry-point map consists of `TrgComponentName` as the key, and a tuple `< ICCEntry Name, CompProtection, CriticalOperations >` as the value. `(compY, <onStart(), No, java. io.FileOutputStream.write()>)`, for example, represents one entry in the `TargetAppICCEntryHashMap`, where `compY` is the component name that receives the inter-app ICC call, and `onStart()` is the entry point of `compY` which is not protected (hence the value `No`) and has critical operation `java.io.FileOutputStream. write()` reachable from the ICC call.

Matching ICCs. The creation of the hash data structures for each source and target component aims at efficiently computing ICC flows across communicating app pairs. We start with traversing the *CAIMap*. For each tuple therein, we retrieve the source and target components, with which we then look up the corresponding hash maps to retrieve the information on entry and exit points. The inter-app ICCs are then matched according to the entry and exit points, while obtaining the complete data flow path associated with the app pair. These matched pairs form the basis for our next step of policy engineering and risk classification.

3.2 Cross-App Flow-Level Features Extraction

We extract four types of features from ICC flows, two from the source app and two from the destination app, as described next.

- Features extracted from the source app are:
  User trigger validation. We compute how much the app actively involves a user in implicitly authorizing inter-app ICC calls. Previous 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) are effective (e.g., [12], [30]). User inputs may be entered through onClick(), onItemClick(), etc. Researchers found that benign single apps typically have a high degree of user-trigger dependence, whereas malware, often performing activities under stealth mode, does not [12], [31]. We extend that user-trigger dependence characteristics to the context of app collusion. We use the data-dependence graphs to quantify the ICC calls' dependence relations with user inputs. This feature captures whether inter-app ICC calls are initiated by the users (in the source app).
  Access to sensitive data. This feature is based on whether the inter-app ICC call involves transmitting any sensitive data (e.g., `getPassword(...)`), specifically exfiltrating sensitive data to external receivers.
- Features extracted from the destination app are:
  Permission checking. This feature captures whether the target component (in the target app) is protected by permission checking when responding to requests from external intents. This feature is related to the risk associated with malware exploiting vulnerable target app as part of the collusion.
  Critical operation. The feature captures if inter-app ICC calls involve critical operation (e.g., network and file operations) in the target component.

Next, we give a detailed description of our risk classification policies based on these features.

3.3 Risk Classification Policies

Risk classification is performed on each communicating app pair based on flow-level features and classification policies, evaluating risks associated with collusion vulnerability of the pair. We extract flow-level features from both ends of the inter-app ICC communications, and then apply a rule-based classification by inspecting the sensitivity of each inter-app ICC flow.

We formulate our classification policies based on the flow features which collectively approximate security risks via inter-app ICCs as follows:

- *Category 1*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 does not have a valid user trigger and has access to sensitive data and the target component C_2 is not protected by permission checking and has critical operation, then this ICC channel is classified as a *high risk* inter-app ICC channel.
- *Category 2*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 does not have a valid user trigger and has no access to sensitive data and the target component C_2 is not protected by permission checking and has critical operation, then this ICC channel is classified as a *high risk* inter-app ICC channel.
- *Category 3*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 does not have a valid user trigger and has access to sensitive data and the target component C_2 is protected by permission checking and has critical operation, then this ICC channel is classified as a *medium risk* inter-app ICC channel.
- *Category 4*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 does not have a valid user trigger and has no access to sensitive data and the target component C_2 is protected by permission checking and has critical operation, then this ICC channel is classified as a *medium risk* inter-app ICC channel.
- *Category 5*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 has a valid user trigger and has access to sensitive data and the target component C_2 is not protected by permission checking and has critical operation, then this ICC channel is classified as a *medium risk* inter-app ICC channel.
- *Category 6*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 has a valid user trigger and has no access to sensitive data and the target component C_2 is not protected by permission checking and has critical operation, then this ICC channel is classified as a *medium risk* inter-app ICC channel.
- *Category 7*: Suppose component C_1 in app P_1 calls component C_2 in app P_2. If the ICC exit point in C_1 has a valid user trigger and has no access to sensitive data and the target component C_2 is protected by permission checking and has critical operation, then this ICC channel is classified as a *no risk* inter-app ICC channel.

We introduced the checking of the component's exposure in the target app as one of the policies in our classification. The reason is that the attackers may exploit vulnerable target app, which is not originally developed for this purpose, as part of their apps collusion attack. In particular, any outside communication coming to the vulnerable app with unprotected components can be classified as collusion, although the vulnerable app is not designed to collude with any apps.

Our classification policies are summarized in Table 1. The final classification decision of the app pair is determined as follows:

- The app pair is classified as high risk if there is at least one high risk inter-app ICC channel between the apps.
- The app pair is classified as medium risk if there is at least one medium risk inter-app ICC channel and no high risk ones between the pair.
- Otherwise, the app pair is classified as no risk.

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

4 Empirical Study

This section presents the experimental results that characterize the effectiveness of our approach in classifying external explicit intent-based ICCs. In particular, the objective of our evaluation is to answer the following questions:

1) How many communicating app pairs have risky communication? (Section 4.1)
2) What are the merits of end-to-end over single-app analysis? (Section 4.2)
3) What are the specific characteristics of high risk inter-app ICC channel? (Section 4.3)
4) Do our policies have fewer false alerts than permission-based policies? (Section 4.4)
5) What are the causes of false alerts in permission-based policies? (Section 4.5)
6) Is our analysis scalable for practical use on real apps in terms of total execution time? (Section 4.6)

We performed our empirical study on 7,251 free popular real-world Android apps randomly selected from Google Play store. These apps cover various application categories and different levels of popularity as determined by the user rating scale. In particular, we used 2,917 high popularity apps, 2,031 intermediate popularity apps, and 2,303 low popularity apps. We checked these apps using tools such as VirusTotal[1] that screen individual apps, and these tools labeled all the samples as benign. Our prototype is implemented in Java based on the Soot framework [32], which parses Dalvik bytecode and extracts the manifest from a given APK file.

4.1 Risky App Pairs Found

We constructed the *CAIMap* for each of the 7,251 apps. 197 of app pairs communicate using direct explicit intent-based ICC channels. We performed our analysis on these 197 app pairs to evaluate the effectiveness of our explicit inter-app ICC risk classification policies.

We found that 178 out of the 197 app pairs use Activity-based inter-app ICC channels (e.g., `startActivity(...)`), 29 app pairs use Service-based inter-app ICC channels, and 9 pairs use Receiver-based inter-app ICC channels. Some app pairs use more than one type of explicit inter-app

1. https://www.virustotal.com/

TABLE 2
Summary of Explicit Intent Inter-App ICC Classification Results on 197 App Pairs

Risk Level	# of app pairs (%)
High Risk	32 (16.2%)
Medium Risk	7 (3.5%)
No Risk	160 (81.3%)
Total # of app pairs	197 (100%)

High risk ICC channel is where external explicit intent ICC is not initiated by the user trigger in the source app, and exposed target component in the target app invokes critical operation(s). The medium risk app pairs found have user-trigger dependences, however, their target components are not protected.

ICC call. Service and Receiver components perform operations in the background without the user awareness as opposed to the Activity component. Thus, we expect the malware developers to utilize Service and Receiver-based inter-app ICC channels for the apps collusion to hide the communication.

Table 2 presents our classification results on 197 app pairs according to our external explicit intent-based ICC policies defined in Table 1. Note that 32 app pairs (16.2 percent) are classified as high risk by policy #3 defined in Table 1 (i.e., they have high risk ICC channels).

Seven app pairs (3.5 percent) have medium risk ICC channels according to our policy #6 defined in Table 1. Each of these two medium risk pairs has explicit inter-app ICCs which are initiated by user triggers, however the target components are not protected. Also, no sensitive data is involved in the inter-app ICCs between them. For 19.7 percent app pairs classified as high/medium risk, we found that the explicit inter-app ICCs initiate critical operations only in the target apps and no sensitive data involved as shown in Table 3.

Additionally, we found that the explicit inter-app ICCs between 160 app pairs (81.3 percent) do not pose any security risk according to our policies. This means that either (i) the external explicit ICCs are initiated by user triggers and target components are protected or (ii) no sensitive data or critical operations are involved in the external explicit ICCs.

We performed an additional evaluation with 1,433 standalone malware apps collected by [33] and [34]. We used our Cross-App ICC Map (CAIMap) tool to identify the set of malware apps that interact with each other in order to find malware app pairs and further conduct our analysis. Unfortunately, we found no malware app pairs, i.e., no interaction between these malware apps.

4.2 End-to-End versus Single-App Analysis

To assist users with inspecting and understanding the results of our analysis, we provide a visualization interface atop *CAIMap* depicting how the apps under analysis interact with one another. Fig. 3a gives such a visualization of the risk level associated with the explicit inter-app ICC calls among the 197 app pairs. As shown, the `com.adobe.air` app, the system runtime environment from Adobe that allows for cross-platform application development and deployment, appears to be of the highest risk based on its largest number of risky connections with other apps as per our security policies—as can be seen with another central (yet gray) node, degree centrality alone does not imply the level of risk.

Our detailed investigation reveals that this app has as many as 198 ICC exits, of which 16 explicitly point to other apps with access to sensitive data (e.g., WiFi MAC address in its `com.abode.air.net` component). In the meanwhile, it contains 24 *exported* components (including activities, services, and receivers) that accept incoming ICCs from external apps of a broad scope (according to the associated intent filters as we checked). These incoming ICCs do not appear to be reached by sensitive data or user triggers (e.g., input or user actions).

Also, none of the exported components of `com.adobe.air` are protected by any permissions. Moreover, eight use permissions are declared inclusive of those allowing access to the Internet and the state of WiFi and cellular network. These permissions enable each of the incoming ICCs to (forwardly) reach five different critical operations (including `android.content.ContentResolver.delete(..)` and `java.io.BufferedReader.readLine()`) in various components (e.g., serializing messages to local storage in its `com.abode.air.wand.message` component). As a result, many external apps are connected to `com.adobe.air` with the inter-app communication classified as *high* risk as per our classification rules.

A particularly interesting and important observation is that some apps communicate with one set of apps at no risk while having high/medium-risk communications with other apps, as illustrated in Fig. 3b. In fact, among the risk-classification results of all the 197 app pairs, we have found 10 such cases. Discovery of these cases demonstrates the need for an end-to-end flow analysis over multiple apps, as well as the advantage of our analysis approach—neither a single-app analysis alone, even of fine-granularity and high precision, nor an inter-app analysis looking at coarse information only, such as those based purely on permission

TABLE 3
A Comparison between Permission-Based Policies and Ours on Explicit Intent Inter-App ICCs

	Permission-based	Our Work
	#app pairs	#app pairs
Inter-App ICC with Sensitive Data Only from Source App	11	0
Inter-App ICC with Critical Operations Only in Target App	33	39
Inter-App ICC without Sensitive Data or Critical Operations	108	0
Inter-App ICC with Sensitive Data and Critical Operations	0	0
Total # of app pairs	152	39

Out of 197 app pairs, 152 app pairs classified as collusion by the permission-based policies, and 39 app pairs classified as high/medium risk by our work. Our analysis has around four times fewer flags than the permission-based policies.

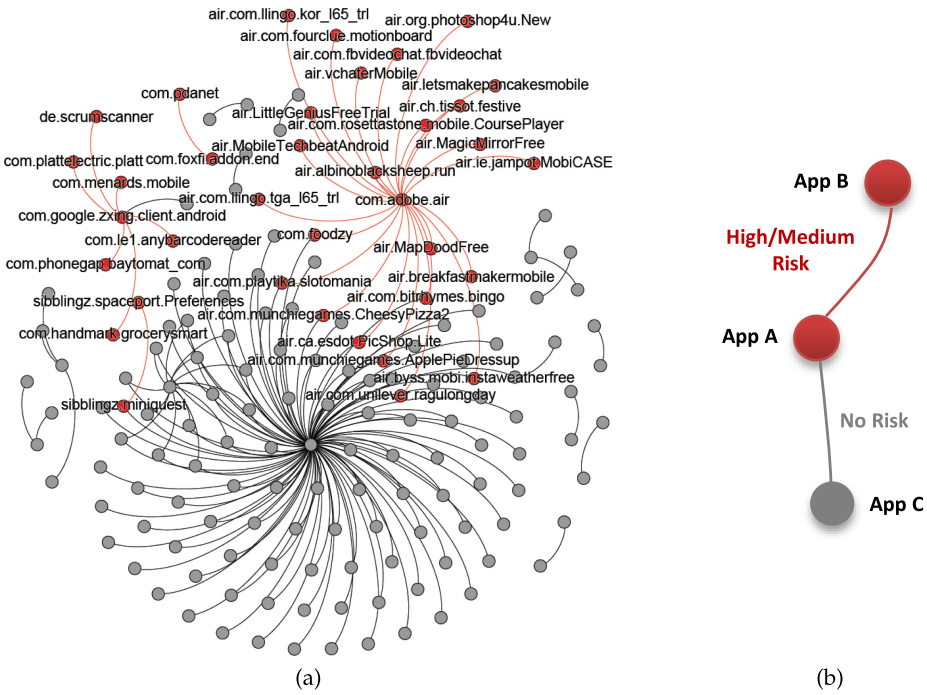

Fig. 3. (a) Visualization of the connections among the 197 app pairs via explicit inter-app ICC calls. Red nodes and edges represent high- and medium-risk inter-app ICC flows classified, while gray nodes and edges represent no-risk inter-app ICC calls, according to our security policies. (b) Illustration of the advantage of end-to-end flow analysis across multiple apps: App A exhibits high/medium-risk (color in red on the relevant node and edge) security behavior when communicating with app B, yet its communication with a different app C appears to be of no risk (denoted by the gray edge and/or node).

checking, would be able to distinguish the communications of an app with others that potentially occur at the same time yet carry different levels of security risk.

To illustrate, consider the example in Fig. 3b, a single-app analysis would decisively identify App A as either secure or risky. However, since it is secure when App C connects to it but becomes risky when it is connected to App B, the single-app analysis would either raise a false warning or miss a dangerous true risk, and potentially suffer from both issues when analyzing a set of apps. Similarly, a coarse permission-based inter-app analysis can have the same problem, which we illustrate through a dedicated case study in Section 4.5.

4.3 Case Study I

In this case study, we looked into one specific pair of communicating apps that is classified as high risk for the existence of high risk inter-app ICC channels between them. In this pair, the first app is called `com.pdanet` which allows the user to share the mobile device's Internet connection with any device through USB, Bluetooth or WiFi Hotspot. The second app is called `com.foxfi.addon` which enables free WiFi Tether on the mobile devices. Two components, `com.foxfi.al` and `com.foxfi.aq`, in the `com.pdanet` app communicate with the Receiver component `com.foxfi.addon.AddOnEvents` in the second app `com.foxfi.addon` by sending explicit inter-app ICC calls using `sendBroadcast(android.content.Intent)`. While these calls do not carry any sensitive data, the destination component `com.foxfi.addon.AddOnEvents` has many critical operations that are executed in response to the inter-app ICC call. The critical operations include `java.io.InputStream.read(...)` and `java.io.OutputStream.write(...)`. The inter-app ICC calls are considered to be high risk according to our policies because they are initiated without user awareness (no user trigger) in the source app `com.pdanet`. Furthermore, the destination component `com.foxfi.addon.AddOnEvents` in the target app `com.foxfi.addon` is exposed and not protected by the permission(s). This example shows a risky app pair executing critical operations in a stealthy mode. Further investigation is needed to confirm if it is indeed malware collusion.

4.4 Flow- versus Permission-Level Policies

The purpose of this experiment is to demonstrate that it is feasible to design flow-based policies for an effective classification. We compare our classification policies with a well-known permission-based collusion detection (XManDroid) [8]. XManDroid is a dynamic analysis tool monitoring communication channels between apps, and it defines communication classification policies based on certain permissions combinations. For example, an app with permission `READ_CONTACTS` must not communicate with an app that has permission `INTERNET`.

We evaluated a set of permission-based policies [8] on our 197 app pairs. Table 3 presents the classification results using permission-based policies [8] on 197 pairs of communicating apps. The permission policies classify 152 out of 197 app pairs as collusion according to permission-based policies. In other words, 77.1 percent of the app pairs are classified as collusion. Table 4 shows the number of the 197 app pairs that trigger alerts per permission-based policies [8]. Some app pairs trigger multiple alerts.

In order to investigate the correctness of the flags raised by permission-based policies and to estimate the false positives reported, we analyzed each inter-app ICC channel in 152 app pairs that are classified as collusion by XManDroid.

TABLE 4
Summary on the Numbers of App Pairs (#Pairs), Out of the Total 197 Pairs Studied, that Trigger Alerts per Permission-Based Policies

Permission-based Policy [8]	#Pairs
(8) A third party application with permission ACCESS_FINE_LOCATION must not communicate to a third party application that has permission INTERNET	149
(9) A third party application that has permission READ_CONTACTS must not communicate to a third party application that has permission INTERNET	137
(10) A third party application that has permission READ_SMS must not communicate to a third party application that has permission INTERNET	121
(11) A third party application that has permissions RECORD_AUDIO and PHONE_STATE or PROCESS_OUTGOING_CALLS must not directly or indirectly communicate to a third party application with permission INTERNET	119

Some app pairs trigger alerts for more than one policy.

We analyzed each inter-app ICC channel to check if it involves sensitive data, critical operations, or both.

For the 152 app pairs classified as collusion by permission-based policies, we found that 11 app pairs (5.1 percent) have inter-app ICC calls with sensitive data, and 33 app pairs (15.4 percent) perform critical operations due to inter-app ICC calls. 108 app pairs (54.9 percent) do not have sensitive inter-app ICC channels (i.e., no sensitive data or critical operations are involved) as shown in Table 3. We thus infer that permission-based policies produce a large number of false warnings: 113 (71 percent) out of 152 app pairs are misclassified as collusion. The main category of false positives is due to the non-sensitive inter-app ICCs, specifically a lack of sensitive data or critical operations involved in a benign explicit inter-app ICC call which does not pose any security concerns.

Compared to permission-based policies, our classification rules are more fine-grained and our analysis produces *four times fewer* flags as shown in Table 3. The results reveal that the majority of the app pairs classified as collusion by permission-based policies do not have sensitive explicit inter-app ICC calls (involving no sensitive data or critical operations). Note that although the XManDroid approach considers implicit ICCs too, here we evaluate it with respect to explicit ICCs only for a fair comparison to our approach.

4.5 Case Study II

We performed a second case study on a pair of communicating apps that is misclassified as colluding apps according to the permission-based policies [8]. In this study, one app `com.projectx.android.ScouterLite` is used for entertainment to measure the "power" level of the people and save pictures. The other app `com.cooliris.media.Gallery` is a gallery app used to show pictures taken from the camera app. These two apps communicate using an explicit inter-app ICC call `startActivity(android.content.Intent)`. We checked the sensitivity of the inter-app ICC call and found no sensitive data or critical operations involved. These apps request sensitive permissions without using them in the code. According to our policies, this app pair does not pose any security risks. Howeve, this app pair is classified as collusion by the permission-based policies [8] as it violates policy #8 in XManDroid: the app (`com.cooliris.media.Gallery`) with permission ACCESS_FINE_LOCATION must not communicate to the app that has permission INTERNET (`com.projectx.android.ScouterLite`).

The permission-based policies can not assess the sensitivity of the inter-app ICC calls well enough, and hence produce high number of false alerts. In contrast, our approach performs in-depth data flow analysis in the source and the destination apps and applies more fine-grained policies which are able to reduce the number of false alerts on benign inter-app ICC calls.

4.6 Performance Analysis

To gauge on the performance of our approach, we executed our analysis against all the studied pairs of apps on a laptop running Ubuntu 14.05 with an Intel i7-5600U CPU@2.6 GHz and 4 GB heap memory for the JVM. For the pool of apps ranging from 16 KB to 70 MB in APK size, our analysis took an average of 41.8 seconds per pair and maximum of 271.2 seconds.

Furthermore, to measure the execution time needed to perform the flow analysis on individual apps, not pair of apps, we compared our analysis with more fine-grained, more precise flow analyses, and evaluated two other popular apps, `airpass.MyMap` and `com.facebook.katana`, and run FlowDroid [19] versus our analysis (for the CAIMAP CONSTRUCTION and FLOW-FEATURE EXTRACTION phases). With the same experimental setup, FlowDroid did not finish in three hours (and we killed the execution) while our analysis completed in 3 and 10 seconds, respectively, for these two apps.

These results suggest that our approach is realistically efficient for practical use, and has substantially higher scalability than more fine-grained flow analysis like FlowDroid. In all, we trade precision for high efficiency in the most time-consuming step (i.e, the static analysis), enabling a reasonable balance between the cost and benefit of a static inter-app security analysis.

4.7 Summary of Results

We summarize our major experimental findings as follows.

1) We observed that the state-of-the-art approach using permission-based policies classifies 77.1 percent of the 197 app pairs we studied as collusion. In particular, 108 of the app pairs are classified as collusion do not have any sensitive inter-app ICC calls (i.e., no sensitive data or critical operations are involved). This suggests that purely permission-based collusion detection can generate a very-high number of false alerts.

2) 19.7 percent of the 197 app pairs were found to have risky inter-app ICC calls. We checked the sensitivity of each inter-app ICC call between these app pairs and found that no sensitive data is, yet critical operations are, involved in those ICC calls. This indicates that these app pairs do not leak sensitive data, but they may abuse system resources by utilizing critical APIs.
3) Our method significantly reduces the number of false alerts given by the state-of-the-art solution. Combined together, our static analysis and detailed security policies constitute a more fine-grained end-to-end risk-classification approach than purely permission based solutions.
4) Our end-to-end security analysis is able to *distinguish* varying risk levels of an app in variant communication contexts with different other apps, for which single-app analysis and coarse permission-based approaches can both give false alerts and miss true threats.

4.8 Limitations

In this work, we focus on analyzing explicit intent inter-app ICCs only. However, some apps my communicate via implicit intent inter-app ICCs. Moreover, some apps can utilize indirect communication channels such as shared files or external server for message passing to evade detection. We plan to address these two limitations by extending our approach to approximate implicit intent inter-app ICCs, and to investigate other communication channels than ICCs among Android apps for identifying potential collusion.

Similar to any static analysis based solution (e.g., [12], [21], [35]), our analysis share some inherent limitations in performing the analysis on programs that employ dynamic code loading, obfuscation techniques, or use of reflection. One can use hybrid analyses (static and dynamic) to provide precise and robust analysis tool. We performed our analysis on the reversed engineering Java bytecode using Dare tool to translate Dalivk bytecode to Java bytecode. The accuracy of our analysis is constrained by the accuracy of the reverse engineering tool.

Currently, our collusion analysis is primarily based on specific, manually-defined rules to classify communication risk levels between communicating pairs. In particular, the risk is determined according to cross-app ICC flows with respect to what are present in the two given apps. As a result, our risk classification might miss app information that could be potentially important for identifying existing risk levels we defined with higher precision and/or for identifying other risks. To over overcome this limitation, we plan to leverage machine learning techniques to deduce more information (from the training dataset) and utilize such information to learn a more capable classifier based on features based on ICC flows and other relevant features.

5 Related Work

Malware collusion is a new threat against Android security that has not been systematically studied. Most existing static ICC-based analyses are for detecting vulnerable-yet-benign apps. We address related work on i) app collusion analysis, ii) app vulnerability analysis, and iii) general-purpose app analysis.

App Collusion Analysis. XManDroid [8] is a runtime monitoring solution of communication channels between apps. It defines communication classification policies based on certain permissions combinations of communicating apps. We compared the effectiveness of this approach with our technique in Section 4. FlaskDroid [36] is another runtime solution that enforces mandatory access control policies to prevent collusion and privilege escalation attacks. It requires modifications in the Android middleware and kernel layer. These dynamic analysis for collusion detection is more appropriate for analyzing a small set of apps and do not scale to hundreds of thousands of apps. In comparison, our static-analysis approach is more feasible and scalable for analyzing app-pair security under this circumstance.

FUSE [37] presents a single-app and multi-app static information flow- and context insensitive analysis. Similar to XManDroid, FUSE defines coarse-grained information flow assertions based on permissions combinations. In contrast, our flow-level policies are more fine-grained. FUSE is focused on evaluating single apps. COVERT [38] proposes a formal method approach (namely model checking) to analyzing app pairs, using the first-order logic based formal specification language Alloy. The program analysis results are then subject to formal verification in a model checker. DidFail [29] combines FlowDroid [19] and Epicc [5] to track data flows between Android components. DidFail currently focuses on ICC flows between Activities only, and hence it does not track data flows across other components such as Service and Broadcast Receiver.

ApkCombiner [9] proposes to pack multiple Android apps (apk files) into a single app package for accommodating existing *intra*-app analysis tools (e.g., IccTA [7]) to *inter*-app analysis scenarios. While this approach might offer a straightforward solution to some particular problems (e.g., mapping ICCs across multiple apps), inter-app analysis problems are not *generally* reducible to intra-app ones. In fact, ApkCombiner drops certain individual-app information and can leave unresolved various conflicts that arise during the merging process thus lead to inexecutable resulting apps and/or incorrect inter-app analysis results. For instance, it would be inapplicable to our analysis because the combined package differentiates neither components from different apps nor the originally separate sets of permissions, both of which are needed for our risk classification.

BlueSeal [39] presents a flow permissions analysis to identify single-app flows as well as cross-app flows via IPC mechanism. BlueSeal requires modification to Androids package installer to add discovered flow permissions for display at installation-time and for cross-app analysis done on a phone. In contrast, our approach focuses on analyzing the sensitivity and context of each inter-app ICC flow (e.g., sensitive data, critical operations) and defines fine-grained security policies, as opposed to analyzing flow permissions. Moreover, BlueSeal requires modifying the app to include the newly-added flow permissions to display at installation-time. Our approach does not require any modification to the app and it is offline analysis. Although BlueSeal focuses only on permissions, it is complementary to our approach.

PRIMO [40] presents a probabilistic technique for estimating and prioritizing the likelihoods of inter-app ICC connections to reduce the workload of security analysts.

Although PRIMO does not provide a complete ICC security detection solution, it is complementary to our work. Marforio et al. [16] implements and measures the efficiency of different overt and covert channels for applications collusion. It does not provide any collusion defenses or classification policies.

App Vulnerability Analysis. Privilege escalation attack in the Android system was first demonstrated by Davi et al. [41]. However, they did not provide any defense techniques for this attack. Confused deputy is a special type of privilege escalation attack where a malicious app exploits a vulnerability of a trusted app to perform a critical operation. IPC Inspection [42] addresses confused deputy attack and found that a number of pre-installed apps are vulnerable to this attack. The idea of IPC Inspection is to reduce the permissions of an app when it receives a message from another app with less privilege. This approach is somewhat strict because the apps can not receive messages from a less privileged app for legitimate purposes. In addition, reducing the app's privileges can make the app malfunction or crash. Similarly, QUIRE [43] provides a lightweight provenance system to prevent the confused deputy attack. It tracks the call chain of ICC and denies the request if the caller app does not have the required permission. However, QUIRE is not designed for detecting the malicious colluding apps. Saint [44] presents policies for install-time permission granting and runtime inter-application communication based on their permissions. Their policies allow the application to control which applications can access it. Chan et al. [45] present a static analysis tool of Android apps to check if the apps can be exploited to launch privilege escalation attacks. However, this tool is coarse-grained since it analyzes the manifest file only, not the app's code.

ComDroid [4] and Epicc [5] identify application communication-based vulnerabilities. They analyze the intent object used by ICC API calls to describe two main categories of attacks, namely intent stealing and intent spoofing. The focus of the analysis is on individual applications. Yet, they did not provide classification policies nor examine inter-app information flow but rather check ICC exit and entry points only.

Also, they assume that any unprotected public component is vulnerable regardless if there is a path from the public component to the critical operations or not. However, this approach may increase the number of false alerts when there is no path from the public component to the critical operations. On the other hand, in comparison to the more precise resolution of ICC calls in Epicc which deals with both explicit and implicit ICCs, we adopted an approximated static analysis (including the local string-constant evaluation) to resolve explicit ICCs only. While this approximation deals well with explicit ICCs, a more sophisticated analysis like Epicc would be necessary for extending our current design to address end-to-end flows via implicit ICCs as well.

CHEX [3] identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps, using data-flow-based reachability analysis. CHEX is designed to detect vulnerable interfaces components within a single app and does not track data through app boundaries.

General-Purpose App Analysis. FlowDroid [19] and DroidSafe [46] present a general information flow analysis framework for Android applications to detect data leaks. Ernst et al. [47] proposes an information flow verification model for Android applications to guarantee that the applications are free of malicious information flows. Apposcopy [6] presents a static analysis approach for detecting stand-alone malicious apps based on extracting data- and control-flow properties of the target app and matching them against predefined malware-class specification (signature).

Amandroid [21] offers a framework for precise dependence analysis of Android apps, on which a range of dependence-based clients including taint analysis can be built. Yet, these frameworks work on individual apps and none of them address inter-app flows or provide classification policies against the apps collusion attack as we target in this work.

Other app security analyses exist, such as detecting risky Android apps based on the types of permissions requested [14], classifying Android apps based on the requested permissions and API calls of the app [11], and identifying Android malware through characterizing sensitive API calls without user-trigger dependence [12], [31]. In contrast, our approach not only considers permissions but also examines detailed inter-app data flows due to ICCs. The detection of malicious campaign in [48] shares similar goals to ours, yet it targets the cellular network as opposed to our focus on mobile apps.

6 Conclusions and Future Work

We presented a precise and scalable end-to-end flow analysis to identify the risk level associated with communicating Android applications. Our approach with fine-grained security risk classification policies is able to reduce the number of false alerts generated by the state-of-the-art permission-based solution. Our inter-app ICC analysis can be used for many useful security analyses which include, but are not limited to, applications collusion analysis and vulnerability analysis.

As an immediate next step, we are about to strength our current inter-app analysis by addressing two major limitations of the presented technique. First, we plan to extend our approach to approximate implicit intent-based inter-app ICCs and define more security policies to further improve the classification accuracy. Second, we plan to investigate other communication channels than ICCs among Android apps for collusion detection. Finally, we plan to expand our framework by learning the cross-app ICC flow features from a large set of apps and classifying unknown communicating apps with the trained predictor to cover risks that are not explicitly exposed in the given pair of apps.

References

[1] T. Wüchner, M. Ochoa, and A. Pretschner, "Robust and effective malware detection through quantitative data flow graph metrics," in *Proc. Detection Intrusions Malware Vulnerability Assessment*, 2015, pp. 98–118.

[2] K. Elish, D. Yao, and B. Ryder, "On the need of precise inter-app ICC classification for detecting Android malware collusions," in *Proc. IEEE Mobile Secur. Technol. Workshop Conjunction IEEE Symp. Secur. Privacy*, 2015, pp. 1–4.

[3] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.

[4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. Mobile Syst. Appl. Serv.*, 2011, pp. 239–252.
[5] 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. USENIX Secur. Symp.*, 2013, pp. 543–558.
[6] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 576–587.
[7] L. Li, A. Bartel, T. Bissyandé, J. Klein, Y. Le, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 280–291.
[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2012, pp. 1–18.
[9] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon, "ApkCombiner: Combining multiple android apps to support inter-app analysis," in *Proc. ICT Syst. Secur. Privacy Protection*, 2015, pp. 513–527.
[10] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust Malware detection in Android," in *Proc. Int. Conf. Secur. Privacy Commun. Netw.*, 2013, pp. 86–103.
[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Efficient and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–12.
[12] K. Elish, X. Shu, D. Yao, B. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Comput. Secur.*, vol. 49, pp. 255–273, 2015.
[13] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. Int. Conf. Mobile Syst. Appl. Serv.*, 2012, pp. 281–294.
[14] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 241–252.
[15] F. Liu, C. Wang, A. Pico, D. Yao, and G. Wang, "Measuring the insecurity of mobile deep links of Android," in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 953–969.
[16] C. Marforio, H. Ritzdorf, A. Francillo, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 51–60.
[17] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Languages Syst.*, vol. 12, pp. 26–60, 1990.
[18] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proc. 29th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2014, pp. 515–526.
[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, pp. 259–269.
[20] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 77–88.
[21] 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. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341.
[22] F. Liu, H. Cai, G. Wang, D. Yao, K. Elish, and B. Ryder, "MR-Droid: A scalable and prioritized analysis of inter-app communication risks," in *Proc. Mobile Secur. Technol. Workshop Conjunction IEEE Symp. Secur. Privacy*, 2017, pp. 189–198.
[23] A. Bosu, F. Liu, D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 71–85.
[24] H. Cai and B. Ryder, "Understanding Android application programming and security: A dynamic study," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 364–375.
[25] A. Sadeghi, H. Bagheri, and S. Malek, "Analysis of Android inter-app security vulnerabilities using COVERT," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 725–728.
[26] J. Jenkins and H. Cai, "ICC-inspect: Supporting runtime inspection of Android inter-component communications," in *Proc. 5th Int. Conf. Mobile Softw. Eng. Syst.*, 2018, pp. 80–83.
[27] J. Jenkins and H. Cai, "Dissecting Android inter-component communications via interactive visual explorations," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 519–523.
[28] W. Ahmad, C. Kästner, J. Sunshine, and J. Aldrich, "Inter-app communication in Android: Developer challenges," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 177–188.
[29] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proc. 3rd ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2014, pp. 1–6.
[30] B. Wolfe, K. Elish, and D. Yao, "Comprehensive behavior profiling for proactive Android malware detection," in *Proc. Inf. Secur. Conf.*, 2014, pp. 328–344.
[31] K. Elish, D. Yao, and B. Ryder, "User-centric dependence analysis for identifying malicious mobile apps," in *Proc. IEEE Mobile Secur. Technol. Workshop Conjunction IEEE Symp. Secur. Privacy*, 2012, pp. 1–4.
[32] R. Vall ée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a Java bytecode optimization framework," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.*, 1999, Art. no. 13.
[33] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.
[34] VirusShare. (2015). [Online]. Available: http://virusshare.com/
[35] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. 20th USENIX Secur. Symp.*, 2011, pp. 21–21.
[36] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 131–146.
[37] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-app security analysis with FUSE: Statically detecting Android app collusion," in *Proc. 4th Program Protection Reverse Eng. Workshop*, 2014, Art. no. 4.
[38] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: Compositional analysis of android inter-app permission leakage," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 866–886, Sep. 2015.
[39] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 515–526.
[40] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. L. Traon, "Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2016, pp. 469–484.
[41] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proc. 13th Int. Conf. Inf. Secur.*, 2010, pp. 346–360.
[42] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Secur. Symp.*, 2011, pp. 22–22.
[43] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "QUIRE: Lightweight provenance for smart phone operating systems," in *Proc. USENIX Secur. Symp.*, 2011, pp. 23–23.
[44] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel, "Semantically rich application-centric security in Android," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2009, pp. 340–349.
[45] P. P. Chan, L. C. Hui, and S. Yiu, "A privilege escalation vulnerability checking system for Android applications," in *Proc. Int. Conf. Commun. Technol.*, 2011, pp. 681–686.
[46] M. Gordon, D. Kim, J. Perkins, L. Gilhamy, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–16.
[47] M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 1092–1104.
[48] N. Boggs, W. Wang, S. Mathur, B. Coskun, and C. Pincock, "Discovery of emergent malicious campaigns in cellular networks," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2013, pp. 29–38.

Karim O. Elish received the PhD degree in computer science from Virginia Tech, in 2015. He is an assistant professor of computer science, Florida Polytechnic University. Before joining Florida Poly, he was an assistant professor with the Department of Computer Science and the associate director of the Information Analytics and Visualization (IAV) Center, Purdue University (Fort Wayne Campus). He received the Florida Polytechnic University Award for Excellence in Teaching, in 2017 for excellence in teaching practices reflecting the highest standards in pedagogy. His current research interests focus on mobile security, software security, and software engineering. One of his journal papers was placed in the top 10 most cited papers by the Elsevier *Journal of Systems and Software* (JSS). He is a member of the IEEE, ACM, Upsilon Pi Epsilon, and Phi Beta Delta Honor Society.

Haipeng Cai received the PhD degree in computer science and engineering from the University of Notre Dame, Notre Dame, in 2015. He worked on computer graphics and visualizations during his previous graduate studies and was a software developer in Internet search services and embedded systems. He is currently an assistant professor with the School of Electrical Engineering and Computer Science, Washington State University, Pullman. His research interests include software engineering and software systems in general with emphasis on program analysis and its applications for the quality, security, and reliability of evolving software. He is a member of the ACM and IEEE.

Daniel Barton received the MS degree in computer science from Virginia Tech, Blacksburg, Virginia. His research interests include mobile security, malware analysis and detection, and developing security tools for cyber analysts. He is currently a software engineer with Lockheed Martin.

Danfeng Yao received the BS degree from Peking University, Beijing, China, the master's degrees from Princeton University and Indiana University, Bloomington, and the PhD degree from Brown University. She is an Elizabeth and James E. Turner Jr.'56 faculty fellow and CACI faculty fellow. Her expertise is on software and system security. Her recent work on Java secure coding has driven multiple high-profile Apache projects to harden their code, including Apache Spark and Apache Ranger. In the past decade, she has been working on designing and developing deployable anomaly detection to defend against stealthy exploits and attacks. She holds multiple U.S. patents for her anomaly detection technologies. She is the lead author of the book *Anomaly Detection as a Service*. She has been named an ACM distinguished scientist for her contributions to cybersecurity research.

Barbara G. Ryder received the AB degree in applied mathematics from Brown University, in 1969, the master's degree in computer science from Stanford University, in 1971, and the PhD degree in computer science from Rutgers University, in 1982. She is a emerita faculty member with the Department of Computer Science, Virginia Tech, where she held the J. Byron Maupin professorship in engineering. From 2008-2015, she served as head of the Department of Computer Science, Virginia Tech, and retired on September 1, 2016. She served on the faculty of Rutgers from 1982-2008. Her research interests on static and dynamic program analyses for object-oriented systems, focus on usage in practical software tools for ensuring the quality and security of industrial-strength applications. She became a fellow of the the ACM in 1998, received the ACM SIGSOFT Influential Educator Award (2015) and the ACM President's Award (2008), was selected as a CRA-W distinguished professor (2004), and received the ACM SIGPLAN Distinguished Service Award (2001). She currently is an editorial board member of the *ACM Transactions on Software Engineering Methodology*, the *IEEE Transactions on Software Engineering*, the *Software: Practice and Experience*, and the *Science of Computer Programming*.