MR-Droid: A Scalable and Prioritized Analysis of Inter-App Communication Risks

Fang Liu[1], Haipeng Cai[2], Gang Wang[1], Danfeng (Daphne) Yao[1], Karim O. Elish[3], and Barbara G. Ryder[1]

[1]Department of Computer Science, Virginia Tech
[2]School of Electrical Engineering and Computer Science, Washington State University
[3]Department of Computer Science, Florida Polytechnic University
fbeyond@cs.vt.edu, hcai@eecs.wsu.edu, {gangwang,danfeng}@cs.vt.edu, kelish@flpoly.org, ryder@cs.vt.edu

***Abstract*—Inter-Component Communication (ICC) enables useful interactions between mobile apps. However, misuse of ICC exposes users to serious threats, allowing malicious apps to access privileged user data via another app. Unfortunately, existing ICC analyses are largely insufficient in both accuracy and scalability. Most approaches rely on single-app ICC analysis which results in inaccurate and excessive alerts. A few recent works use pairwise app analysis, but are limited by small data sizes and scalability.**

In this paper, we present MR-Droid, a MapReduce-based computing framework for *accurate* and *scalable* inter-app ICC analysis in Android. MR-Droid extracts data-flow features between multiple communicating apps to build a large-scale ICC graph. We leverage the ICC graph to provide contexts for inter-app communications to produce precise alerts and prioritize risk assessments. This scheme requires quickly processing a large number of app-pairs, which is enabled by our MapReduce-based program analysis. Extensive experiments on 11,996 apps from 24 app categories (13 million pairs) demonstrate the effectiveness of our risk prioritization scheme. Our analyses also reveal new real-world hijacking attacks and collusive app pairs. Based on our findings, we provide practical recommendations for reducing inter-app communication risks.

I. Introduction

Inter-Component Communication (ICC) is a key mechanism for app-to-app communication in Android, where components of different apps are linked via messaging objects (or *Intents*). While ICC contributes greatly to the development of collaborative applications, it also becomes a predominant security attack surface. In the context of *inter-app* communication scenarios, individual apps often suffer from risky vulnerabilities such as Intent hijacking and spoofing, resulting in leaking sensitive user data [8]. In addition, ICC allows two or more malicious apps to collude on stealthy attacks that none of them could accomplish alone [5], [26]. According to a recent report from McAfee Labs [1], app collusions are increasingly prevalent on mobile platforms.

To assess ICC vulnerabilities, various analytics methods have been proposed, ranging from to inter-app Intent analysis [8], [29] to static data flow tracking [2], [38], [5]. Yet, most of these approaches focus one *individual* app at a time, ignoring its feasible communication context, *i.e.*, how the vulnerable interface can be exploited by other real-world apps. In addition, single-app analyses produce overly conservative risk estimations, leading to a high number of (false) alarms [8], [11], [29]. Manually investigating all the alerts would be a daunting task for security analysts.

Recently, researchers start to co-analyze ICCs of two or more apps simultaneously. This allows researchers to gain empirical contexts on the actual communications between apps and produce more relevant alerts, *e.g.*, identifying privacy leak [22] and permission escalations [3], [34]. However, existing solutions are largely limited in scale due to the high complexity of pair-wise components analyses ($O(N^2)$). They were either applied to a much smaller set of apps (only a few hundred, versus a few thousand in single-app analyses), or small sets of inter-app links. The only system that successfully processed a large number of apps (*e.g.*, 11K apps) is PRIMO [27]. However, PRIMO is an ICC mapping tool for app pairs, which does not provide security and risk analysis. Moreover, PRIMO is designed to run on a single workstation. The large memory consumption makes it challenging for market scale analysis.

In this paper, we present MR-Droid, a MapReduce-based parallel analytics system for *accurate* and *scalable* ICC risk detection. Our goal is to empirically evaluate ICC risks based on an app's inter-connections with other real-world apps, and detect high-risk pairs. Our intuition is that an ICC is of high-risk not only because it has a proof-of-concept vulnerability, but more importantly the app is actually communicating with other apps through this ICC interface. This intuition is similar to a recent work that prioritizes proof-of-concept vulnerabilities in CVE [33]. Ideally, all vulnerabilities should be addressed, but our priority needs to be on those that are causing a real-world impact.

To achieve this goal, we construct a large-scale *ICC graph*, where each node is an app component and the edge represents the corresponding *inter-app* ICCs. To gauge the risk level of the ICC paris (edge weight), we extract various features based on app flow analyses that indicate vulnerabilities. For instance, we examine whether the ICC pair is used to pass sensitive data, or escalate permissions for another app. With the ICC graph, we then *rank* the risk level of a given app by aggregating all

This research is supported by the DARPA APAC award FA8750-15-2-0076. The authors would like to thank Amiangshu Bosu, Xinming (Simon) Ou, Timothy Fraser, Michael Gordon, Matthew Might, Michael Ernst, and Ali Butt for their helpful suggestions and feedback.

its ICC edges connected to other apps.

Our system allows app market administrators to accurately pinpoint market-wise high-risk apps and prioritize risk migration. Individual app developers can also leverage our results to assess their own apps. In addition, the ICC graph provides rich contexts on how an app vulnerability is exploited, and by (or with) which external apps. For the purpose of this paper, we customize our features to capture three major ICC risks: app collusion (malicious apps working together on stealth attacks), intent hijacking and intent spoofing (vulnerabilities that allow unauthorized apps to eavesdrop or manipulate inter-app communications).

To scale up the system, we implement MR-Droid with a set of new MapReduce [10] algorithms atop the Hadoop framework. We carefully design the ⟨*key, value*⟩ pairs for each MapReduce job and balance the workload among MapReduce nodes. Instead of performing pair-wise ICC mapping, we leverage the hash partition functionality of MapReduce and achieve near-linear complexity. The high-level parallelization from MapReduce allows us to analyze millions of app pairs within hours using commodity servers.

We evaluate our systems on a large set of 11,996 Android apps collected from 24 major Google Play categories (13 million ICC pairs). Our manual post-analysis confirms the effectiveness of our approach in reducing false, excessive alerts. For apps labeled as high-risk, we obtain a 100% true positive rate in detecting collusion, broadcast injection, activity- and service-launch based intent spoofing, and a 90% true positive rate for activity hijacking and broadcast theft detection. For app pairs labeled as medium- or low-risk, manual analysis show their actual risks are substantially lower, indicating the effectiveness of risk prioritization. Our system is also highly scalable. With 15 commodity servers, the entire process took less than 25 hours for an average of only 0.0012 seconds per app pair. More importantly, our runtime experiment shows the computation time grows near-linearly with respect to the number of apps.

Our empirical analysis also reveals new insights to app collusion and hijacking attacks. We find previously unknown real-world colluding apps that leak user data (*e.g.*, passwords) and escalate each other's permission (*e.g.*, location or network access).

In addition, we find a more stealth way of collusion using *implicit intents*. Instead of "explicitly" communicating with each other (easy to detect), the colluding apps using high customized (rare) actions to mark their implicit intent to achieve the same effect of explicit intent. This type of collusion cannot be detected by analyzing each app independently. Finally, we find third-party libraries and the automatically-generated apps have contributed greatly to today's hijacking vulnerabilities.

Our paper makes four key contributions.

- We propose an empirical analytics method to identify risks within inter-app communications (or ICC). By constructing a large *ICC graph*, we assess an app's ICC risks based on its empirical communications with other apps. Using a risk ranking scheme, we accurately identify high-risk apps.
- We design and implement a highly scalable MapReduce framework for our inter-app ICC analyses. With carefully designed MapReduce cycles, we avoid full pair-wise ICC analyses and achieve near-linear complexity.
- We evaluate our system on 11,996 top Android apps under 24 Google Play categories (13 million ICC pairs). Our evaluation confirms the effectiveness of our approach in reducing false alerts (90%-100% true positive rate) and its high scalability.
- Our empirical analysis reveals new types of app collusion and hijacking risks (e.g., transferring user's sensitive information including password[1] and leveraging rarely used implicit intents[2]). Based on our results, we provide practical recommendations for app developers to reduce ICC vulnerabilities.

II. Models & Methodology

In this section, we describe our threat model and the security insights of large-scale inter-app ICC analysis. Then we introduce our research methodology.

A. Threat Model

Our work focuses on security risks caused by inter-app communications through ICCs. We consider three most important classes of inter-app ICC security risks.

- **Malware Collusion.** Through inter-app ICCs, two or more apps collude to perform malicious actions [25], [26], [35] that cannot be achieved by each app alone. This is done either by passing Intents to exported components, or by using the same `sharedUserId` among the colluding apps. Malware collusion can result in disguised data leak and system abuse.
- **Intent Hijacking.** Sending an intent via an implicit ICC may not reach the intended recipient. Instead, it may be intercepted (hijacked) by an unauthorized app. This threat can be categorized into three subclasses based on the type of the sending component: broadcast theft, activity hijacking, and service hijacking [8]. Intent hijacking may lead to data/permission leakage and phishing attacks.
- **Intent Spoofing.** A typical scenario is that a vulnerable app has a component that only expects intents from itself (or Android system). However, if the component is exposed, other malicious apps can send forged intents spoofing this vulnerable app to trigger undesired actions. Intent spoofing can be classified into three subclasses by the type of the receiving component: broadcast injection, malicious activity launch, and malicious service launch [8].

[1] `uda.onsiteplanroom` and `uda.projectlogging`

[2] `org.geometerplus.fbreader.plugin.local_opds_sanner` and `com.vng.android.zingbrowser.labanbo-okreader`

B. Security Insights for Inter-app Analysis

Next, we discuss the key differences between single-app analysis and pair-wise app analysis. We explain why large-scale pairwise analysis is critical to threat detection and mitigation.

We start with an example, where two apps A and B communicating via a vulnerable ICC. Suppose A has access to location information and sends it out through an implicit Intent. The manifest of B defines an Intent filter that restricts the types of Intents to accept. Once A's Intent passes through the Intent filter, B then sends the received data out through SMS to a phone number (`phoneNo`).

With single-app analysis (*e.g.*, ComDroid [8]), B would be identified as *always* vulnerable to Intent spoofing attacks because its `SendSMSService` component is exported without permission protection (other apps can send SMS through B). Likewise, A would be detected as *always* vulnerable, as other apps could hijack its implicit Intent and receive the location information. However, single-app analysis does not consider specific communication contexts. It cannot track the destination of the implicit sensitive Intent in A, and unable to identify which communicating peers may hijack the intent and abuse the sensitive data. The Android design of ICC makes this type of vulnerabilities prevalent. ComDroid [8] reported that over 97% of the studied apps are vulnerable. However, reporting an app as generically vulnerable or malicious is overly-conservative, and lead to insufficient precision and excessive alerts.

To this end, we propose to prioritize apps' ICC risks based on their communication contexts. In practice, the ICC risk becomes real when an app has actual communications with other real-world apps. For example, A's security risk increases if malicious apps are found being able to leak A's location information. More external hijacking apps would indicate a higher security risk because users have a higher possibility to install A and the malicious apps at the same time.

The idea of prioritizing vulnerabilities is not just for ICC risks. Researchers recently identified a similar problem in CVE, the largest public database of security vulnerabilities [33]. Due to the large volume of proof-of-concept vulnerabilities, many of them remain unaddressed. There is a strong need to prioritize vulnerabilities based on how likely the vulnerability will be exploited in the real-world.

To evaluate the risk level of a given app, we not only need to check its own ICC properties, but need to examine its external communications with peer apps. This can only be done by pair-wise app analysis. The analysis should keep track of which apps can hijack the Intent of A or collude with A to abuse the location information, and provides a detailed list of apps and their corresponding components (*e.g.*, `B.SendSMSService`) involved in the attacks. To fully reconstruct the communication contexts for an app, we also need to scale-up our analysis to cover a large number of apps and app-pairs.

In the rest of this paper, we demonstrate how such a solution can be realized by a highly-scalable distributed ICC graph, and a neighbor-aware risk ranking scheme. Our solution not only seeks to detect vulnerable apps but also label them with risk levels and communication contexts. This provides a big picture for the mobile app market to identify the most vulnerable apps to prioritize risk mitigation. At the same time, app developers can be better guided to fix vulnerabilities when they are aware of risks from external apps.

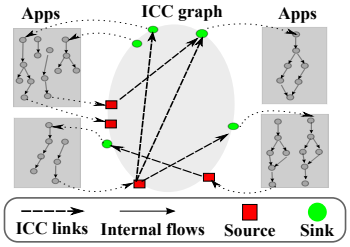

Fig. 1: The construction of an ICC graph. Our MapReduce algorithms compute the complete ICC graph for a set of apps. We only consider inter-app ICCs.

C. Method and Computational Goal

We build MR-Droid for ICC risk analysis with two key steps:

1) We build a complete inter-app ICC graph with distributed ICC mapping. This is done by identifying all communication app pairs for a given set of apps.
2) We perform neighbor-aware inter-app security analysis on top of the ICC graph and rank the apps and app pairs with respect to their risk levels.

Fig. 1 shows how the ICC graph is constructed.

Definition 1: Inter-app ICC graph is a directed bipartite graph $G = (V_S, V_D, E_{V_S \to V_D})$, where each node $v \in V_S \cup V_D$ represents the specifications of an entry or exit point of external ICC of an app, and each edge $e \in E_{V_S \to V_D}$ represents the Intent-based communication between one app's ICC exit point $v_s \in V_S$ to another app's ICC entry point $v_d \in V_D$. We refer to a node in the set V_S in G as a *source*. We refer to a node in the set V_D as a *sink*.

In Section III, we introduce the details of distributed ICC mapping, which involves a preprocessing step 1. IDENTIFY ICC NODES and two MapReduce jobs referred to as 2. IDENTIFY ICC EDGES, 3. GROUP ICCS PER APP PAIR. Section IV illustrates how we perform neighbor-aware inter-app risk analysis with respect to intent hijacking, intent spoofing and collusion attacks.

III. Distributed ICC Mapping

We now present the distributed ICC mapping algorithms in MR-Droid. The purpose of the algorithms is to construct the ICC graph in a scalable manner. Our distributed ICC mapping algorithms involve three major operations: IDENTIFY ICC NODES, IDENTIFY ICC EDGES, and GROUP ICCS PER APP PAIR, where the last two operations are performed in MapReduce framework. The workflow is shown in Fig. 2.

A. Identifying ICC Nodes

The purpose of *Identify ICC Nodes* is to extract all the sources and sinks from all available apps. We customized the Android ICC analysis tool IC3 [28] for this purpose.

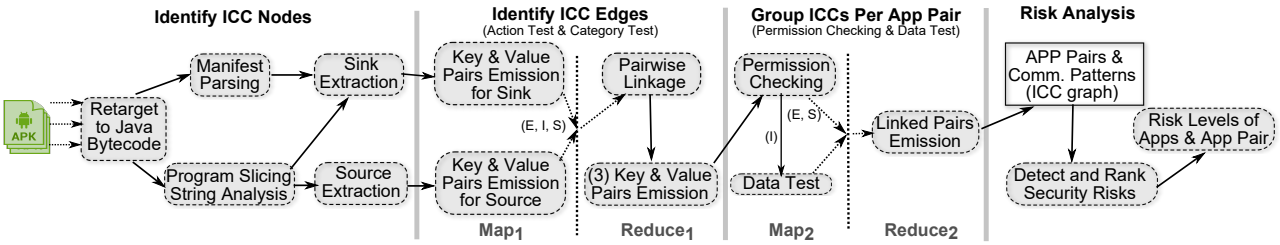

Fig. 2: Our workflow for analyzing Android app pairs with MapReduce. The dashed vertical lines indicate redistribution processes in MapReduce. E, I, S represent explicit edge, implicit edge, and sharedUserId edge respectively.

Sink Extraction. We analyze each app's decoded manifest and retrieve the attributes for sinks. We also parse the dynamic receivers of each app as exported receiver components.[3] A public open-access component may generate multiple sinks. The number is determined by the number of the component's Intent filters, as each Intent filter is extracted into a sink in ICC graph. A component is extracted into a sink only when it has no Intent filters.

Source Extraction. The attributes of sources are obtained by propagating values in fields of outbound Intents. We utilize IC3 [28] to perform program slicing and string analysis. String analysis may generate multiple possible values for one Intent field, which is due to multiple paths from where the string value is defined. We track all the possible values, and label them with the same identifier. Thus, there may be multiple values associated with an attribute of a source.

B. Identifying ICC Edges

IDENTIFY ICC EDGES operations are performed together in MapReduce. The purpose is to identify all matching source and sink pairs, which are connected with edges in the ICC graph. There are three types of edges: *explicit edge*, *implicit edge*, and *sharedUserId edge*.The explicit edge and implicit edge correspond to explicit and implicit intent. The sharedUserId edge corresponds to the private ICC communication using sharedUserId.

Our Map_1 algorithm transforms sources or sinks into the $\langle key, value \rangle$ pairs. During the redistribution phase (after Map_1), the $\langle key, value \rangle$ pairs that have the same *key* are sent to the same reducer. $Reduce_1$ algorithm identifies qualified edges from the redistributed records. Qualification is based on action test and category test[4] for implicit edges and exact string match for explicit and sharedUserId edges. These tests are efficiently performed in the redistribution phase with only the edges that pass the action test and category test sent to $Reduce_1$.

Outputs at the end of $Reduce_1$ phase are $\langle key, value \rangle$ pairs, where *key* consists of package names (the app identifier) of a communicating app pair (the edge), and *value* contains properties of the edge.

[3]The main difference between a dynamic receiver and the components in the Android manifest is that the former can only receive Intents when it is in pending or running status.

[4]https://developer.android.com/guide/components/intents-filters.html

C. Multiple ICCs Per App Pair

An app pair may have multiple ICC data flows between them. For our subsequent risk analysis in Section IV, we will need to identify and cluster all the inter-app ICCs that belong to an app pair. Therefore, the purpose of GROUP ICCS PER APP PAIR performed in MapReduce is to group ICCs that belong to the same app pair. In addition, we perform permission checking on all the three types of edges and data test on implicit edges.The *key* is the package names of an app pair. A reducer in the $Reducer_2$ algorithm records the complete set of inter-app ICCs between two apps that pass the tests.

D. Workload Balance

The types of actions and categories are unevenly distributed with very different frequencies. This property leads to the unbalanced workload at different nodes in $Reduce_1$. This will significantly impact the performance because most of the nodes are idle while waiting for the nodes of heavy workload. To address this problem, we add a tag before each key emitted by the Map_1. The tag helps to divide the large amount of key-value pairs, which should be sent to one reducer, into m parts feeding m reducers. The tags incur additional communication and disk storage overhead. The optimal parameter m is selected to maximize performance and achieve high scalability.

Our distributed ICC mapping conservatively matches the sources with all the potential sinks even if the links are of low likelihood. The full set of links provides security guarantees for our risk analysis. For example, collusion apps may leverage rarely used implicit intent specifications for targeted communication. Ignoring any low-likelihood links would miss detecting such attacks. Our following security analysis incorporates all the possible links and gives a prioritized risk result to minimize security analysts' manual investigation.

IV. Neighbor-based Risk Analysis

Next, we present our neighbor-aware security risk analysis, covering the three key types of threats defined in our threat model. The input of our risk analysis is the ICC graph produced by the MapReduce pipeline, and the output is the risk assessment per communicating app pair in the graph for malware collusion threats, or per individual app for (Intent) hijacking and spoofing threats.

For the latter two types of risks for a single app, our neighbor-aware security analysis differs from previous approaches (e.g., ComDroid [8], Epicc [29]), because we exam-

TABLE I: Features for security risk assessment.

Feature	Definition	Hijacking	Spoofing	Collusion
Data linkage	# of outbound/inbound links that carry data	✓		✓
Permission leakage	# of apps connected to via links that involve permission leaks	✓	✓	✓
Priority distribution	Mean and standard deviation of priority set for outbound/inbound links	✓	✓	
Link ambiguity	# of outbound explicit links missing package prefix in the target comp.	✓		
Connectivity	# of outbound/inbound links and/or number of connected apps	✓	✓	✓

ine ICC sinks and sources in the app and all of its inter-app ICC links. Our approach adds more semantic and contextual information to the risk assessment, which helps to reduce false warnings. In addition, our risk assessment identifies the presence of a risk (*i.e.*, detection) and reports how serious that risk may be (*i.e.*, risk ranking). In this section, we explain how we compute the security risks associated with app vulnerabilities and collusions.

A. Features

To assess the security risks of ICCs, we extract five key features as shown in Table I. We utilize combinations of these features to detect different types of risks. Note that these features are all expressed in numerical values.

Regarding the definition of each feature: 1) we define that an ICC link carries data if the ICC Intent contains a non-empty data field; 2) a permission leak occurs when an unauthorized app gains access to the resources (via the other app) without obtaining the required permissions themselves; (3) the "priority" of a link is a property of the Intent filter for the corresponding ICC Intent; we use median and standard deviation of the priority values of all inbound/outbound links to characterize the distribution; (4) when specifying the target component for an explicit ICC, the package name of that component may be missing; (5) we use the number of ICC links and the number of connected apps to quantify *link-connectivity* and *app-level connectivity* respectively.

Using these features, we can empirically compute the risk level for each app. We first compute the risk with respect to individual features, and then aggregates them to obtain an overall risk value. Important features are assigned more weight during the aggregation. Meanwhile, some features (e.g., *priority distribution*) are of high importance and thus are used to determine the overall risk level directly without involving other features.

B. Hijacking/Spoofing Risk

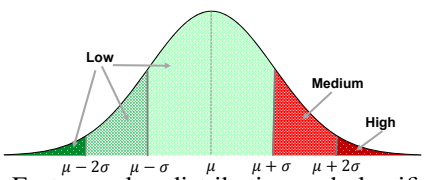

Fig. 3: Feature value distribution and classification.

We first describe the risk assessment method for hijacking and spoofing attacks. Given a feature f, the feature-wise risk of f for app a is evaluated based on the distribution of all f values in the ICC graph, and classified into three risk levels from `low` to `high` as illustrated in Figure 3. The rationale is that the risk level is proportional to the feature value and all feature values in the ICC graph are *normally* distributed. We normalize the categorical value of each feature f by mapping (`low, medium, high`) to (.1, .5, 1.0) respectively. To incorporate the varying contributions of different features on the overall risk, we assign customizable feature weights based on heuristics. The eventual risk value is computed as the weighted sum of feature-wise risk values. We upgrade the risk level if no apps are detected at the higher levels in order to normalize the risk levels.

1) Hijacking: We utilize all the five link features for detecting Intent hijacking. Since the Intent-sending app is the subject of evaluation in a hijacking scenario, we only consider outbound links for all relevant features. Also, since an Intent is much less likely to be hijacked if the target is specified explicitly than implicitly, only implicit links are considered. The only exception is for *link ambiguity*, computing which only involves explicit links. This special feature captures the possibility of hijacking via ambiguous target-component specification—ignoring the package prefix in the target component name. Using these five features, the overall risk value is computed as follows.

- If *link ambiguity* $\geq$ 1 or *priority* values are high (mean+STD$\geq$500), we report the risk as `high` and exit.
- We sum up the numerical feature values for *data linkage*, *permission leakage*, *app-level connectivity*, and *link-level connectivity*, with weights .3, .4, .2, and .1 respectively. Then we cast the resulting numerical values to its categorical risk level, according to where the value falls in the three equal-length subintervals of [0.1,1.0].

2) Spoofing: In a spoofing scenario, the Intent-receiving app is the subject of evaluation, thus we consistently consider inbound links for the three features involved (see Table I). In addition, we consider both implicit and explicit links except for the *priority distribution* (implicit links only) because the priority can only be set for implicit Intents. Finally, since spoofing attack is more likely to succeed when using explicit Intents than implicit ones, we weigh explicit links higher. Using these three features, the overall risk value is computed as following:

- If the *priority* values are high (mean+STD$\geq$500), we report the risk as `high` and exit.
- We sum up the numerical feature values for *permission leakage*, *app-level connectivity*, and *link-level connectivity* for explicit links using weights .4, .2, and .2 (total of .8). Then, we sum up the three feature values for implicit links with weights 0.1, 0.05, and 0.05 (total of 0.2). Finally, we add up the two numbers and cast the result back to its categorical risk level according to where

the value falls in the three equal-length subintervals of [0.1,1.0].

C. Collusion Risk

For collusion attacks, we analyze two apps together in each app pair. For the three features involved (see Table I), we consider both inbound and outbound links, and both implicit and explicit links. In addition, since explicit link indicates collusions on-purpose, we weight explicit links higher than implicit links. The overall risk level is computed as following:

- For *connectivity*, we assign value 5 for an unidirectional explicit connection (*i.e.*, one app connects to the other app via explicit ICCs but the other app connects back via implicit links), 10 for a bidirectional explicit connection, and 3 for a bidirectional implicit connection. For *data linkage*, we assign value 3 and 2 for data transfer through explicit and implicit links respectively. For *permission leakage*, we assign value 3 if leakage exists (0 otherwise).
- We calculate the sum of the three numerical feature values above, and cast the result to a categorical risk level (`high`, `medium`, `low`) according to where the value falls in the three equal-length subintervals of [1,16].
- Particularly, we select apps with low implicit connectivity, and retrieve the corresponding app pairs with bidirectional implicit connections. Empirically, the connectivity is *low* if the number of inbound or outbound implicit links $<$20. Pairs with both apps having low implicit connectivity are set to `high` risk of collusion. Pairs with only one app having low implicit connectivity are set to `medium` risk.

V. Evaluation

We implemented our system with native Hadoop MapReduce framework. The input is the ICC sources and sinks extracted from individual apps using IC3 [28], the most precise single-app ICC resolution in the literature. We modified IC3 to accommodate the MapReduce paradigm. The Hadoop system is deployed on a 15-node cluster, with one node as master and the rest as slaves. Each node has two quad-core 2.8GHz Xeon processors and 8GB RAM.[5]

Datasets. For our evaluation, we apply our system to 11,996 most popular free apps from Google Play. We select the top 500 apps from each of the 24 major app categories (4 apps were unavailable due to bugs in program analysis). We downloaded the apps in December 2014 with an Android 4.2 client. We ran our inter-app ICC analysis and security risk assessment on 12,986,254 app pairs.

In addition to this empirical dataset, we also test our system on DroidBench[6], the most comprehensive Android app benchmark for evaluating Android taint analysis tools. The latest suite DroidBench 3.0 consists of 8 app-pair test cases for evaluating inter-app collusions.

[5]The algorithms can also be implemented with Spark [41] with faster in-memory processing. It will require much larger RAMs to hold all the data.

[6]https://github.com/secure-software-engineering/DroidBench/tree/develop

Validation. Due to the lack of ground truth on the empirical data, we devote substantial efforts to manually inspect the apps for validation.[7] In addition, we will evaluate the performance of the distributed ICC analyses by gauging the running time of each phase of the pipeline. Our evaluation seeks to answer the following questions.

- *Q1: What are the risk levels of app pairs?* (Section V-A)
- *Q2: How accurate is our risk assessment and ranking?* (Section V-B)
- *Q3: What do detected attacks look like?* (Section V-C)
- *Q4: What is MR-Droid's runtime, including per-app ICC resolution and ICC graph analyses?* (Section V-D)

A. Q1: Results of Risk Assessment

We apply our system to the collected app dataset. The resulting ICC graph contains 38,134,207 source nodes, 26,227,430 sink nodes and 75,123,502 edges. On the per-app level, there are in total 12,986,254 app pairs that have at least one ICC link. Each app averagely connects with 1185 external apps (9.9% of all apps), confirming the overall sparsity of the graph. For non-connected app pairs, we can safely exclude them during the security analysis. Our security analysis focuses on all potential security risks related to Intent hijacking, Intent spoofing and app collusion (Section II-A). We quantify and rank security risks into as categorical risk levels following the procedures detailed in Section IV. In total, our system identified 150 high-risk apps, 1,021 medium and 10,825 low risk apps.

Table II summarizes the results, highlighting the numbers of apps or app pairs vulnerable to the *high* and *medium* level of risks. Prominently, stealthy attacks such as *activity hijacking* and *broadcast theft* dominate medium and high risks of any type. These attacks often involve passively steal user data.

The more intrusive types of attacks such as *service launch* and *broadcast injection* are less prevalent. In addition, considerably collusion-attacks are revealed by our analyses. There are six colluding app pairs are of high risk, and 169 are of medium risk.

App Categories. By analyzing the categories of detected apps, we find PERSONALIZATION apps have the most significant contribution to high&medium-risk pairs (19.3%). These apps usually help users to download themes or ringtones using vulnerable implicit intents. For example, the app `com.aagroup.topfunny` lets user to download other apps from the web. A malicious app can easily hijack the implicit intent and redirect user to downloading malicious apps or visiting phishing websites. ENTERTAINMENT apps are also heavily involved high&medium-risk ICCs (8.9%). For example, app `com.rayg.soundfx` offers downloading ringtones via vulnerable implicit intents. Another high&medium-risk category is LIFESTYLE (7.3%). These apps often require sensor data (*e.g.*, GPS, audio, camera) for their functionalities. One vulnerable app is `com.javielinux.andando`. It is a location tracking app but it broadcasts GPS information

[7]We plan to share these manually labeled datasets as benchmarks to the Android security community.

TABLE II: Numbers of apps vulnerable to Intent spoofing/hijacking attacks and potentially colluding app pairs reported by our technique, and true positives (in parentheses) manually validated as indeed vulnerable or colluding, per risk category and level. The DroidBench test result is not included in this table.

	Intent Hijacking			Intent Spoofing			Collusion Pairs
	Activity Hijacking	Service Hijacking	Broadcast Theft	Activity Launch	Service Launch	Broadcast Injection	
High	94 (9/10)	10 (7/10)	15 (9/10)	17 (10/10)	4 (4/4)	7 (7/7)	6 (6/6)
Med.	790 (8/10)	32 (6/10)	303 (7/10)	9 (8/9)	8 (8/8)	0	169 (14/169)
Low	11,112 (2/10)	11,954 (0/10)	11,678 (1/10)	11,970 (0/10)	11,984 (0/10)	11,989 (0/10)	12,986,079 (0/10)

through an implicit broadcast intent. Other apps can eavesdrop the broadcasted intent and acquire location data even if they don't have location permissions.

We find that categories such as FINANCE are less involved in high&medium-risk ICCs (1.2%). Since these apps often deal with banking and financial payments, it is likely that these apps have put more efforts on security. Even so, we still find high-risk apps under these categories. For example the app `com.ifs.androidmobilebanking.fiid3383` has links in its app to visit its website. However, they trigger implicit intents, which can be easily redirected to phishing websites by malicious apps.

Validation with DroidBench. We have used DroidBench to confirm the effectiveness of our analysis. The latest suite DroidBench 3.0 consists of 8 app-pair test cases for evaluating inter-app collusion. Our approach detected all 8 of them: 6 were labeled as high risk under *collusion*, and 2 were labeled as high risk under *intent hijacking*. Our system put the two apps under hijacking category because the way they collude leads to a hijacking vulnerability. They use implicit intents with the default Category and "ACTION_SEND" Action. Our experiment shows that many other apps can receive this type of implicit intents and acquire sensitive information.

B. Q2: Manual Validation

To further assess the usefulness of MR-Droid, we manually examined over 200 apps to verify the validity of our detection results. More specifically, we randomly select 10 apps from all 7 attack categories at all 3 risk-levels — for category/level with fewer than 10 apps, we chose all of them. We carefully inspected the selected apps in two ways to check their behavior: (1) static inspection, by which we examine relevant Intent attributes of each app and manually match them against peer apps (in its neighbor set); (2) dynamic verification, in which we run individual apps and app pairs on an Android emulator and observe suspicious behaviors in the activity logs.

The validation result is presented in (the parentheses of) Table II. For each selected set of apps or app pairs, we report the percentage that was verified to be indeed vulnerable. Overall, this work justified our risk detection and ranking approach. We find apps labeled as high-risk have a much higher rate to be actually vulnerable. We have a 100% true positive rate in detecting collusion, broadcast injection, activity- and service-launch based intent spoofing. We have a 90% true positive rate for activity hijacking and broadcast theft detection. For apps labeled as low risks, the true positive rate is much lower: 5 out of 7 categories have a true positive rate of 0%. These results suggest that the rankings produced by our approach can help users and security analysts prioritize their inspection efforts.

Sources of Errors. Our approach still has a few false alerts (at high-risk level). We find that most of them were caused by unresolved attributes in relevant Intent objects. In those cases, we conservatively match unresolved source points to all the sinks in order to cover possible ICCs. This matching leads to false positives. One improvement this is to combine our static program analysis with probabilistic models to better resolve ICC attributes [27]. Regarding false negatives, we cannot give a reliable estimation since we did not scan all apps within the market and due to the lack of ground truth.

C. Q3: Attack Case Studies

Based on our manual analysis, we present a few case studies to discuss empirical insights on ICC-based attacks.

Stealthy Collusion via Implicit Intents. During our analysis, we find some colluding apps also use *implicit* intents in an effort to avoid detection. Colluding apps usually use explicit intent, since they know "explicitly" which app(s) they are colluding with. However, using explicit intent makes the collusion easier to detect, even by single-app analyses. The new collusion uses highly customized "action" and "category" to mark the implicit intent, hoping no other apps will accidentally interrupt their communication. For example, the app `org.geometerplus.fbreader.plugin.local_opds_scanner` has open interfaces with a customized action name `android.fbreader.action.ADD_OPDS_CATALOG` in its intent filter. Another app `com.vng.android.zingbrowser.labanbookreader` sends implicit intent with the same action and leverages the first app to scan the local WiFi network (the second app itself does not have the permission). This type of collusion cannot be detected if each app was analyzed individually.

Risks of Automatically Generated Apps. We found many of the high-risk apps were automatically generated by app-generating websites (e.g., www.appsgeyser.com). These apps send a large number of implicit Intents, attempting to reuse other apps' functionality as much as possible. For example, `com.conduit.app_39b8211270ff4593ad85daa1-5cbfb9c6.app` is an automatically generated app and it contains a number of unprotected interfaces including those for viewing social-media feeds from Facebook and Twitter. It

has 20,805 connections with other apps, *five* times more than the average.

Hijacking Vulnerabilities in Third-Party Libraries. We observed that a significant amount of vulnerable exit points are from third-party libraries. For example, a flash light app `com.ihandysoft.ledflashlight.mini.apk` bundles multiple third-party libraries for Ads and analytics. One of the libraries `com.inmobi.androidsdk` sends implicit intents to access external websites (*e.g.*, connecting to Facebook). A malicious app can hijack the Intent and redirect the user to a phishing website to steal the user's Facebook password.

Colluding Apps by the Same Developers. Colluding apps were usually developed by the same developers. For example, `org.geometerplus.zlibrary.ui.android` and `org.geometerplus.fbreader.plugin.local_opds_scanner` app pair. The first app is a book reader app with 167,625 reviews and 10 million – 50 million installs. It leverages the later app (100K – 500K installs) to scan the user's local network interface. The first app itself does not have the permission to do so. Both of the two apps were developed by "FBReader.ORG Limited".

Another example is the pair of `uda.onsiteplanroom` and `uda.projectlogging`. The first app is for document sharing in collaborative projects, and the second app is for project progress logging. With a click of a button, users will be redirected from the first app to the second app. User's sensitive information in the first app is also sent to the second app without user knowledge. Such information includes user's name, email address, password, etc. These two apps were written by the same developer "UDA Technologies, Inc.".

The practical risk of app collisions varies from case to case. The bottom line is, different apps written by the same developer should not open backdoors for each other to exchange user data and access privileges without the user knowledge.

Insecure Interfaces for Same-developer Apps. Usually, apps developed by the same developer have specialized interface to communicate with each other. The secure way to do this is to use *sharedUserID* mechanism to protect the data and interface from exposing to other apps. In our empirical analysis, we find 560 app pairs are developed by the same developer without using `sharedUserID` links. Instead, they use explicit intents to implement the communication interface, which is exposed to all other apps, leaving hijacking vulnerabilities.

D. Q4: Runtime of MR-Droid

Finally, we analyze the runtime performance of MR-Droid. Figure 4 depicts the time cost of our MapReduce pipeline (y axis) as the number of apps increases (x axis). Overall, the result shows that our approach is readily scalable for large-scale inter-app analysis. The running time of ICC node identification appears to dominate the total analysis cost, yet its growth is linear with the number of apps. In addition, given the sparse nature of the ICC graph (rarely does an app communicate to all apps), we manage to achieve near-linear complexity for edge identification and grouping ICCs. In total, it takes 25 hours to perform the complete analysis on 13 million ICC pairs for 12K apps.

Noticeably, grouping ICCs for all app pairs only took 44 minutes, rendering 0.0012 seconds per app pair. This speedup benefits from the reduced input size — a large portion of (unmatched) ICC links have been excluded in the previous phase. Our load balancing design also contributes to speeding up the process. Currently, our cluster has 15 nodes. We anticipate that increasing the cluster size would further speed up the inter-app ICC analysis.

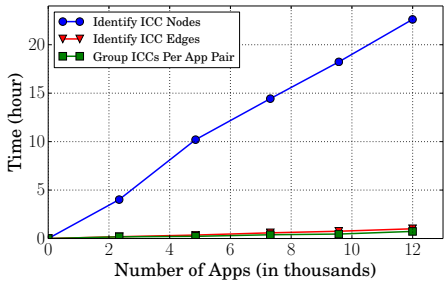

Fig. 4: Analysis time of the three phases in our approach.

TABLE III: Worst-case complexity of different phases.

Approaches	Operation Time	Complexity
Identify ICC Nodes	$\mathcal{T}n$	$O(n)$
Identify ICC Edges	$t_l mn$	$O(mn)$
Group ICCs Per App Pair	$t_g mn$	$O(mn)$
Our total (MapReduce)	$\mathcal{T}n + (t_l + t_g)mn$	$O(mn)$

Complexity Breakdowns. We show the computational complexities of different phases in Table III. n is the number of apps, and m is the average number of links per app. Note that $t_l + t_g \ll \mathcal{T}$ and $m \ll n$ because *the ICC graph is sparsely connected*, and unmatched app pairs do not take any analysis time.

Existing pair-wise app analysis methods are less scalable because they need to combine the code of the two apps first before running static analysis (*e.g.*, IccTA coupled with ApkCombiner). The time complexity for parsing n apps is $O(n^2)$. The operation time is $k\mathcal{T}\binom{n}{2}$ with $2\mathcal{T}$ being the average analysis time of each two apps.

Summary of Findings. Our manual verification confirms the accuracy of our system. For app pairs at high-risk level, we obtain a 100% TP rate for the detection of collusion, broadcast injection, activity- and service-launch based intent spoofing; We have a 90% TP rate for broadcast theft, activity- and service hijacking detection. On the other hand, app pairs at low-risk level indeed have substantially lower TP rate. This result indicates that our risk prioritization is effective. Our empirical analysis reveals new types of app collusion and hijacking risks (e.g., leveraging rarely implicit intents for more stealthy collusion). Our runtime experiments demonstrate that MapReduce pipeline scales well to a large number of apps. Analyzing 11,996 apps and performing ICCs matching took less than 25 hours. More importantly, the runtime cost has a near-linear increase with respect to the number of apps.

VI. Security Recommendations

Based on our empirical study and manual verification, we make the following recommendations for app developers to reduce potential security risks.

- When carrying out sensitive operations, developers are encouraged to use *explicit Intents* instead of implicit ones. For example, to access Facebook account, communicating to the Facebook app via explicit Intents is preferred over a `https` URL as an implicit Intent to the Facebook webpage.
- When possible, it is recommended to integrate all necessary functionalities into a single app. If the developer has to develop multiple apps that communicate with each other, it is encouraged to use the safe communication via *sharedUserID* to restrict other apps accessing the interfaces.
- Developers are encouraged to use customized actions and to enforce data and permission restrictions. These customized configurations (compared to using the default) will greatly reduce the chance to accidentally communicate with other apps, hence reducing risks.
- It is recommended to avoid automatically generating apps using tools or services. Developers should be aware that third-party libraries could be vulnerable or leak user's sensitive information.

VII. Limitations

Our risk analysis is primarily based on Intent attributes, while dismissing data flows that involve the Intents. It is possible that this approach misses true vulnerabilities. Alternatively, one can perform more in-depth and fine-grained data-flow analysis [2], [15]. However, these methods are more time-consuming and thus less practical for market-wide app analyses. Compared to static analysis, dynamic approaches, either single app analyses [12] or inter-app analyses [17] are typically more precise. The issue is that dynamic analysis is even slower, and it often misses true threats that are not triggered during the executions.

Our current analysis focuses on app pairs. It is possible for attacks to span across three or more apps. One way to generalize our approach is to cluster app pairs into app groups. This requires efficiently parallelizing the clustering process, which we consider as future work.

VIII. Related Work

Inter-app Attacks. Attacks with multiple apps involved have been proposed recently. Chin *et al.* [8] analyzed the inter-app vulnerabilities in Android. They pointed out that the message passing system involves various inter-app attacks including broadcast theft, activity hijacking, etc. Davi *et al.* [9] conducted a permission escalation attack by invoking higher-privileged apps, which do not sufficiently protect their interfaces, from a non-privileged app. Ren *et al.* [32] demonstrated intent hijacking can lead to UI spoofing, denial-of-service and user monitoring attacks. Soundcomber [35] is a malicious app that transmits sensitive data to the Internet through an overt/-covert channel to a second app. Android browsers also become the target of inter-app attacks via intent scheme URLs [36] and Cross-Application Scripting [16]. Inter-app attacks have become a serious security problem on smartphones.

Defense and Detecting Techniques. A significant amount of solutions have been proposed to perform single-app analysis and detect ICC vulnerabilities [14], [15], [40], [42], [37]. They mostly analyze sensitive data flow within an app and detects whether the sensitive information that flows out of the phone. ComDroid [8] and Epicc [29] identify Intents specifications on Android apps and detect ICC vulnerabilities. CHEX [24] discovers entry points and detects hijacking-enabling data flows in the app for component hijacking (Intent spoofing). On-device policies [7], [30] were also designed to detect inter-app vulnerabilities and collusion attacks.

Inter-app analysis has also been proposed in the literature [3], [4], [19], [20], [22], [31]. They mostly focus on the precision and analysis of in-depth flow information between apps. However, they are not designed to analyze large-scale apps. It is unclear how well they scale with real world apps. For example, IccTA [22] combines multiple apps into one and performs single-app analysis on the combined one. It would be extremely expensive if applied to a larger set of apps. DIALDroid [4] is much more efficient. However, it still takes over 6,339.6 hours for 110,150 apps on a 64 GB RAM machine. In addition, in straightforward implementations, expensive program analysis is performed repetitively, which is redundant. In comparison, we perform data-flow analysis of an app *only once*, independent of how many its neighbors are.

PRIMO [27] reported ICC analysis of a large pool of apps. Its analysis is on the likelihood of communications between two apps, not specifically on security. Goodman-Kruskal's γ rank correlation measure and 10-fold cross-validation are used to validate its probabilistic model. It does not provide any security classification. Moreover, PRIMO runs on a single host. Our evaluation shows that running PRIMO with 10K apps requires over 40GB memory even with highly compressed metadata. Using a single machine is not a practical solution to analyze market-wide apps with the $O(N^2)$ space complexity.

For related dynamic analysis, IntentFuzzer [39] detects capability leaks by dynamically sending Intents to the exposed interfaces. INTENTDROID [17] performs dynamic testing on Android apps and detects the vulnerabilities caused by unsafe handling of incoming ICC message. SCanDroid [13] checks whether the data flows through the apps are consistent with the security specifications from manifests. TaintDroid [12] tracks information flows with dynamic analysis and performs real-time privacy monitoring on Android. Similarly, FindDroid [43] associates each permission request with its application context thus protecting sensitive resources from unauthorized use. XManDroid [6] was proposed to prevent privilege escalation attacks and collusion attacks by ensuring the inter-app communications comply to a desired system policy. These dynamic analyses complement our static-analysis solution. However,

their feasibility under a large number of app pairs is limited.

MapReduce for Large Scale Analysis. MapReduce framework has been used in various areas such as data mining [18], data management [21], and network/internet security [23]. Our approach leverages the MapReduce framework to analyze large-scale apps for security analysis. To the best of our knowledge, it is the first distributed-computing solution tailored for addressing Android security problems.

IX. Conclusion

In this paper, we presented the design and implementation of MR-Droid, a MapReduce pipeline for large-scale inter-app ICC risk analyses. By constructing ICC graphs with efficient parallelization, our system enables highly scalable inter-app security analysis and accurate risk prioritization. Using MR-Droid, we analyzed 11,996 most popular Android apps (13 million app pairs) and examined their security risk levels against intent hijacking, intent spoofing, and collusion attacks. Our analysis reveals new types of app collusion and hijacking risks (e.g., collusion through stealthy implicit intent). We manually inspected a subset of the apps and validated our security risk analysis. Our empirical results and manual validation demonstrated the merits of MR-Droid in prioritizing various ICC risks and the effectiveness of the prioritization.

References

[1] Mcafee labs threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-may-2016.pdf, 2016.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

[3] H. Bagheri, A. Sadeghi, J. Garcia, and s. Malek. Covert: Compositional analysis of Android inter-app permission leakage. *TSE, IEEE*, 2015.

[4] A. Bosu, F. Liu, D. Yao, and G. Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *ASIACCS*, 2017.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical report, Technische Universität Darmstadt, 2011.

[7] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, 2011.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *ISC*, 2011.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[11] K. O. Elish, D. D. Yao, and B. G. Ryder. On the need of precise Inter-App ICC classification for detecting Android malware collusions. In *MoST, IEEE S&P*, 2015.

[12] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 2014.

[13] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certication of Android applications. Technical report, Department of Computer Science, University of Maryland, College Park, 2009.

[14] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks.

[15] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS*, 2015.

[16] R. Hay and Y. Amit. Android browser cross-application scripting (CVE-2011-2357). Technical report, 2011.

[17] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in Android. In *ISSTA*, 2015.

[18] Q. He, T. Shang, F. Zhuang, and Z. Shi. Parallel extreme learning machine for regression based on MapReduce. *Neurocomput.*, 2013.

[19] Y. Jing, G.-J. Ahn, A. Doupé, and J. H. Yi. Checking intent-based communication in Android with intent space analysis. In *ASIACCS*, 2016.

[20] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *SOAP*, 2014.

[21] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 2014.

[22] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE*, 2015.

[23] F. Liu, X. Shu, D. Yao, and A. R. Butt. Privacy-preserving scanning of big content for sensitive data exposure with MapReduce. In *CODASPY*, 2015.

[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, 2012.

[25] C. Marforio, A. Francillon, and S. Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical report, ETH Zurich, 2011.

[26] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC*, 2012.

[27] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *POPL*, 2016.

[28] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE*, 2015.

[29] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security*, 2013.

[30] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in Android. In *ARES*, 2014.

[31] 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 *PPREW*, 2014.

[32] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in Android. In *USENIX Security*, 2015.

[33] C. Sabottke, O. Suciu, and T. Dumitras. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *USENIX Security*, 2015.

[34] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic detection of inter-application permission leaks in Android applications. *IBM J. Res. Dev.*, 2013.

[35] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011.

[36] T. Terada. Attacking Android browsers via intent scheme URLs. Technical report, 2014.

[37] K. Tian, D. Yao, B. G. Ryder, and G. Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *2016 IEEE Security and Privacy Workshops (SPW)*, 2016.

[38] 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 *CCS*, 2014.

[39] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. IntentFuzzer: Detecting capability leaks of Android applications. In *ASIACCS*, 2014.

[40] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In *CCS*, 2013.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[42] H. Zhang, D. D. Yao, N. Ramakrishnan, and Z. Zhang. Causality reasoning about network events for detecting stealthy malware activities. *Comput. Secur.*, 2016.

[43] Y. Zhang, M. Yang, G. Gu, and H. Chen. FineDroid: Enforcing permissions with system-wide application execution context, 2015.