Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications

Amiangshu Bosu[‡], Fang Liu[*], Danfeng (Daphne) Yao[*], Gang Wang[*]
Department of Computer Science, Southern Illinois University, Carbondale, IL, USA[‡]
Department of Computer Science, Virginia Tech., Blacksburg, VA, USA[*]
abosu@cs.siu.edu, {fbeyond, danfeng, gangwang}@vt.edu

ABSTRACT

Inter-Component Communication (ICC) provides a message passing mechanism for data exchange between Android applications. It has been long believed that inter-app ICCs can be abused by malware writers to launch collusion attacks using two or more apps. However, because of the complexity of performing *pairwise* program analysis on apps, the scale of existing analyses is too small (e.g., up to several hundred) to produce concrete security evidence. In this paper, we report our findings in the first large-scale detection of collusive and vulnerable apps, based on inter-app ICC data flows among 110,150 real-world apps. Our system design aims to balance the accuracy of static ICC resolution/data-flow analysis and run-time scalability. This large-scale analysis provides real-world evidence and deep insights on various types of inter-app ICC abuse. Besides the empirical findings, we make several technical contributions, including a new open-source ICC resolution tool with improved accuracy over the state-of-the-art, and a large database of inter-app ICCs and their attributes.

Keywords

Android, Security, Collusion, Inter-component communication

1. INTRODUCTION

An active and continuous operational effort is necessary to detect stand-alone malicious apps. A recent report showed that hundreds of Trojanized apps were missed by Google's detection and some popular DressCode apps were downloaded over 100,000 times before taken off the Google Play Market [11]. In the meantime, researchers (e.g., [4, 5, 19, 20]) have identified more complex threats associated with app pairs, i.e., inter-app communication security.

Inter-app data-flow analyses go beyond the scope of a single app by bridging the data flows of two potentially communicating apps and analyzing the resulting longer paths for data leaks. For example, an app A accesses the location of the phone, passes the data to another app B, which sends it to an external server. Android apps typically use Inter-Component Communication (ICC), a message passing mechanism, to exchange data. Again, components within the same app also use ICC to communicate with each other [23, 35]. App pairs may communicate via explicit or implicit intent, depending on whether the specification describes the name of the target component (in the explicit intent) or only the attributes (in the implicit intent).

Inter-app data-flow analyses are motivated by the need for characterizing the collective security behavior of two (or more) applications. For example, to answer questions such as *How many inter-app data flows are sensitive and result in external data-leaks and/or privilege escalations? Do these leaks use explicit intents or implicit intents for the communication between the pairs? What is the most commonly observed sensitive information that is leaked?* There may be different reasons behind sensitive inter-app data leaks. During an intentional malware collusion, two apps would work together to complete an attack goal [24]. Because of the evolving nature of attacks and defenses, this new threat is indeed conceivable. With collusion, malware writers can develop multiple benign looking apps to evade the existing single-app screening mechanisms. These apps can complement each other's privileges and accomplish attack goals. Single-app scanners (e.g., [1, 2, 12, 30]) cannot provide complete data-flow characterizations essential for inter-app threat analyses.

Inter-app data leaks may also be due to vulnerable apps being exploited for privilege escalations [9, 33]. Therefore, another relevant question is *Can we distinguish intentional data leaks (i.e., collusion) from unintentional data leaks (due to vulnerable apps)?* Answers to these questions will help securing the mobile ecosystem, but have not been answered in the literature.

The expensive nature of pairwise app analysis is a main obstacle to answering these questions. It requires an in-depth data-flow analysis for all apps, beyond the ICC interfaces. In addition, because of the intrinsic worst-case quadratic complexity ($O(N^2)$, where N is the total number of apps) of the ICC linking operation, scaling the analysis to hundreds of thousands of apps is challenging. Despite recent efforts on inter-app ICC analysis, no satisfactory solution exists that can support a *large-scale* pairwise analysis. For instance, ApkCombiner extracts suspicious inter-app ICCs by combining multiple apps into a single app, and then performs a conventional single-app data-flow analysis [20]. This approach is barely scalable, since an expensive data-flow analysis is repeated for all possible combinations of app-pairs [33]. COVERT [4] and DidFail [19] eliminate the need for redundant data-flow analysis by analyzing each app only once. However, COVERT uses formal model checkers incurring high overhead. DidFail's experimental evaluation is small and uses an erroneous ICC intent resolution mechanism [29]. Although, PRIMO predicts the likelihoods of inter-app ICC occurrences [26], it is not designed for collusion detection.

In this work, we develop a scalable and accurate tool **DIALDroid** (**D**atabase powered **I**CC **A**na**L**ysis for an**Droid**) for inter-app ICC analysis. We use DIALDroid to perform the first systematic

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3052973.3053004

large-scale security analysis on inter-app data-flows among 110,150 apps, including 100,206 most popular apps from the Google Play, and 9,944 malware apps from the Virus Share[1]. DIALDroid completes such a large-scale analysis within a reasonable time frame (6,340 total hours of program analysis and 82 minutes of ICC linking and detection). Our key design characteristics include an adaptive and pragmatic data flow analysis, highly precise ICC resolution, fast ICC matching, and ability to execute fast queries on an optimized relational database. Our paper provides empirical evidence on app collusion and privilege escalations. We summarize our contributions as follows.

- We develop an Android security tool, DIALDroid, for analyzing ICC-based sensitive inter-app data flows. Our design leverages relational database for a scalable matching of ICC entry and exit points, and fast analysis. DIALDroid outperforms state-of-art solutions (IccTA+ApkCombiner[2] and COVERT) on benchmark apps, with a higher accuracy (precision 100 %, recall 91.2%) and orders of magnitude shorter processing time. In addition, DIALDroid's ICC extractor is more accurate than the state-of-the-art solution IC3 [28], with 28% more identified intents and 33% less failed cases.

- We use DIALDroid to analyze the sensitive inter-app ICCs among 100,206 apps from the Google Play Market, and characterize them into 6 threat categories (in Table 3). Our threat categorization is based on threat types (collusive data leak or privilege escalation) and intent types (explicit or implicit).

  We found that collusive data leaks and privilege escalations mostly use implicit intents but did not observe any explicit-intent based collusion. These findings suggest that collusive data leak research should start to examine implicit intents, rather than focusing on explicit intents (e.g., [13]). Our case studies revealed a number of privilege escalation cases among same developer app pairs.

  Although the total numbers of sensitive ICCs and app pairs are extremely high, the number of sender apps initiating these ICCs is surprisingly small. E.g., 1,785,102 inter-app ICCs exhibiting privilege escalation behavior (without collusive data leaks) are originated from 62 sender apps. We also had similar observations in other threat categories. We found that the majority of inter-app ICCs ($> 99\%$) do not carry any sensitive data. This property implies that the typical workload of inter-app ICC analysis is much lower than the worst case workload.

- Our dataset and tools can potentially benefit the broader Android community. We have open-sourced our entire tool-suite on GitHub[3] and have made our database available[4] for other researchers. Our database contains extremely rich data-flow attributes of 100,206 apps from the Google Play and 9,944 apps from the Virus Share. These attributes are extracted by FlowDroid static program analysis, organized into multiple relational tables. We envision the database being useful to both the security and data mining communities to tackle open research questions. In addition, we have released a benchmark suite for inter-app collusion analysis, DIALDroid-Bench[5], which contains 30 *real-world* apps from Google Play. To our knowledge, this is the first inter-app collusion benchmark using real-world apps, as opposed to proof-of-concept apps.

[1] http://virusshare.com
[2] IccTA [21] is designed for intra-app ICCs in a single app. It needs to use ApkCombiner [20] to combine app-pairs for inter-app ICC analysis.
[3] https://github.com/dialdroid-android/.
[4] http://amiangshu.com/dialdroid/
[5] https://github.com/dialdroid-android/dialdroid-bench/

2. THREAT MODEL

Before we present our threat model, we first provide a brief overview of the Android ICC architecture and how malicious apps can leverage inter-app ICCs to leak sensitive information. Components are the basic building blocks of Android apps. There are four types of components[6]. *Activities*, the most common components, represent user interfaces. *Services* perform background processing. *Broadcast receivers* respond to system-wide broadcast announcements (e.g., Wifi connected). Finally, *content providers* manage a shared set of application data. Components communicate using URIs and Intents, within an application (i.e., intra-app ICC) or between applications (i.e., inter-app ICC). An *ICC exit point* requires an `intent` object as a parameter. An intent is either *explicit* (i.e., its recipient is explicitly named) or *implicit* (i.e., only a general action is declared). Android system resolves intents at runtime. Android's intent resolution[7] is based on (1) attributes of an implicit intent (namely *action*, *category*, and *data*), and (2) `IntentFilter` fields declared in the manifests (`AndroidManfest.xml`) of potential receiver apps. The receiver app provides the *ICC entry point*. Resolving intents through string analysis is a key to the detection accuracy. Failure to identify matching ICC exit and entry points results in missed detection (i.e., false negative).

Our inter-app security analysis is oriented around identifying pairwise data flows from a sender app A to a receiver app B that result in two types of threats: *collusive data leak* or *privilege escalation*. Privilege escalation (aka the confused deputy problem) is a well-defined threat where the receiver app B gains unauthorized permissions or sensitive data as a result of its ICC communications with the sender app A [6, 9]. Although the concept of collusive data leak has been described in the literature [5, 24], it has not been formally defined. In our work, we define collusive data leak as a threat where the receiver app B exfiltrates the sensitive data obtained from its ICC communications with the sender app A to an external destination (e.g., via disk output or network output).

Our analysis aims to detect sensitive data flows that result in privilege escalation, collusive data leak, or both. Our labeling of sensitive source and sink statements follows the SuSi project [31], based known sensitive APIs (e.g., API to access deviceID). Insensitive data flows and sensitive data flows that do not exhibit collusive data leak and privilege escalation threats are excluded from the analysis. Our threat model excludes intent spoofing, where the sender app forges intents to mislead receiver apps [9]. We consider both explicit and implicit intents.

Next, we first give our definitions for the security terms used in the paper, including ICC exit leak, ICC entry leak, and sensitive ICC channel. We then give formal definitions of both privilege escalation and collusive data leak threats. Our experiments further distinguish 6 different subtypes of threats, based on various ICC and security properties (in Table 3).

A sensitive ICC channel refers to an ICC link between two components, from an ICC exit point (i.e., an outgoing ICC such as `startActivity`, `bindService`, and `startActivityForResult`) to an ICC entry point (i.e., an incoming ICC such as `onActivityResult` and `getIntent`) that transfers intents containing sensitive information. Our analysis

[6] http://developer.android.com/guide/components/fundamentals.html
[7] http://developer.android.com/guide/components/intents-filters.html#Resolution

is focused on sensitive ICC channels and excludes non-sensitive ICC channels.

A property of *ICC exit leak* is that an app's ICC exit point is data dependent on a sensitive data source, i.e., there exists a data-flow path from the sensitive source to the ICC exit. In the context of inter-app ICCs, we use the ICC exit leak to describe the sender app. Intuitively, ICC exit leaks identify sender apps that leak sensitive data via inter-app ICCs.

A property of *ICC entry leak* is that an app's ICC entry point is the source of data-flow paths of sensitive sinks that send the received data externally (e.g, via networks). In the context of inter-app ICCs, we use the ICC entry leak to describe the receiver app. Intuitively, ICC entry leaks identify receiver apps that leak received data externally. Next, we use the terminology introduced above to define privilege escalation and collusion data leak.

- *Collusive data leak* is a threat associated with a sensitive ICC channel between a sender component A in an app and a receiver component B in another app, where A has an ICC exit leak and B leaks the received data from A via an ICC entry leak.

- *Privilege escalation* is a threat associated with a sensitive inter-app ICC channel between a sender component A in an app and a receiver component B in another app, where A has an ICC exit leak and B does not have the permission to access the data from A.

A collusive data leak may also result in a privilege escalation. Because of the overlap between the two threats, we further divide them into 6 sub-categories of threats in Table 3. Inter-app ICCs that result in neither collusive data leak or privilege escalation are not recorded.
Intentional vs. Unintentional Attacks. One of the difficulties in collusion detection is to confirm the cause of an observed problematic data flow. It is well known that vulnerable sender apps (e.g., with exposed broadcast ICC interfaces) cause privilege escalations [9], i.e., the receiver app can intentionally exploit the vulnerabilities. However, intentional collusion between two apps may also result in privilege escalation. Similarly, collusive data leak may be intentional or inadvertent. Regardless of the causes, these data flows can potentially compromise the device and data security. Our large-scale empirical study helps expose and pinpoint these disguised threats.

3. DIALDROID OVERVIEW

The workflow of our inter-app ICC security analysis involves four key operations: ICC ENTRY / EXIT POINT EXTRACTION, DATAFLOW ANALYSIS, DATA AGGREGATION, and ICC LEAK CALCULATION. They are briefly described below.

- ICC ENTRY / EXIT POINT EXTRACTION: Given an app, we extract the permissions and the attributes of the intent filters from the `AndroidManifest.xml` file. We perform static analysis to determine the attributes of the intents passing through ICC exit points.

- DATAFLOW ANALYSIS: We use static taint analysis to determine ICC exit leaks and ICC entry leaks in an app. We dynamically adjust the precision configuration of taint analysis to ensure the timely completion of each app.

- DATA AGGREGATION: We aggregate the data extracted in previous two steps to store in a relational MySQL database. DIALDroid database schema is composed of 42 tables and is designed to facilitate efficient storage and fast data retrieval.

- ICC LEAK CALCULATION: We use fine-grained security policies to detect potential sensitive inter-app ICC channels. Using SQL stored procedures and SQL queries, we compute ICCs with collusive data leaks and privilege escalations.

DIALDroid executes the first three steps once for each app (complexity $O(N)$, where N is the total number of apps being analyzed). The complexity of ICC leak calculator is $O(mN)$, where m is the number of apps with ICC exit leaks and in the worst case, $m = N$. However, for real-world apps m would several times smaller than N. In our study, we found m is 28 times smaller than N (explained in the appendix).

3.1 ICC Entry / Exit Point Extractor

This operation identifies all the ICC end points (both entries and exits) from apps, by performing a single pass of analysis on each app. We describe our new tool IC3-DIALDroid for this purpose. We have made this tool open source.[8] Our ICC entry point extractor subsystem extracts the manifest file from the apk, parses the permissions requested by the app, and parses the ICC entry points of that app from `IntentFilters`. We use static analysis to identify intent values, similar to prior studies [28, 29, 35]. Although our implementation uses the libraries provided by IC3, the state-of-the-art ICC extractor [28], our IC3-DIALDroid has several significant enhancements providing better robustness and higher intent discovery than IC3, which are described next.

IC3 conservatively adopts call graph generation procedure from FlowDroid skipping incremental callback analysis, which incrementally extends the call graph to include the newly discovered callbacks, and the scan is run again since callback handlers are free to register new callbacks on their own. This process is repeated until the call graph reaches a fixed point [3]. One pass callback analysis improves the runtime performance of IC3. However, it results in missed intents due to imprecise Android lifecycle modeling. In comparison, IC3-DIALDroid implements incremental callback analysis, which significantly increases the number of discovered intents.

IC3-DIALDroid analyzes on Android .apks directly. It does not require the Dare tool for reverse engineering [27], and can directly extract the attributes of ICC exit points. In comparison, IC3 is dependent on the Dare tool. Although Android apps are developed in Java, those are compiled into Dalvik bytecode (a custom format developed by the Android project), instead of traditional Java class file. Thus, IC3 requires the Dalvik bytecode to be retargeted using Dare, which not only requires additional preprocessing time, but also may introduce inaccuracies [27].

We identified several defects in IC3, specifically in handling different types of real-world apks and the constraint solver's failure to reach a fixed point even after a long time for some apps. We fixed those defects and implemented code to identify and break race conditions.

We compared the performance of IC3-DIALDroid with IC3 on 29 applications from DroidBench 3.0 and 1,000 randomly selected apps with a timeout of 15 minutes for each app. Table 1 shows a comparison between the two tools. On DroidBench, DIALDroid took 13 seconds (8.6%) less than IC3 to compute entry and exit points and identified the same number of intents. On the 1,000 randomly selected real-world apps, DIALDroid identified 28% more intents and encountered 33% less failed cases. However, due to more precise lifecycle modeling, IC3-DIALDroid spent 13.3% more time.

[8] Available at: https://github.com/dialdroid-android/ic3-dialdroid.

Table 1: Comparisons of our ICC extractor tool IC3-DIALDroid with the state-of-the-art IC3, in terms of robustness, accuracy, and runtime on benchmark apps and 1,000 real-world apps. Our tool identifies 28% more intents and has 33% fewer failed cases for real-world apps.

	DroidBench 3.0			**1,000 Real-World Apps**		
	Failed	Intents found	Time	Failed	Intents found	Time
IC3	0	27	151s	**123**	**30,640**	43hrs
Ours	0	27	138s	**83**	**39,080**	48hrs

3.2 Dataflow Analyzer

We considered the three state-of-the-art static analysis tools for Android apps, 1) FlowDroid [3], 2) Amandroid [35], and 3) DroidSafe [15], to build a dataflow analyzer. While DroidSafe [15] claims to be the most precise static analysis tool, it is 20 to 50 times slower compared to FlowDroid and Amandroid. While both the FlowDroid and the Amandroid offered similar runtime performances, we found that FlowDroid rarely failed to analyze an app. Therefore, we build our Dataflow analyzer based on the FlowDroid, but make several pragmatic improvements.

1) *Number of sources / sinks*: Static taint analysis requires a set of sources (e.g., originating methods of sensitive data, such as API calls to retrieve a user's location) and a set of sinks (e.g., methods through which data can exit the application or device). The number of sources / sinks in an app impacts the taint analysis time. To manage the number of sources and sinks, the dataflow analyzer analyzes an app in two steps. First, for each of the ICC exit points, we investigate if the intents sent through that point can potentially include any sensitive information (i.e., determine the ICC exit leaks). The dataflow analyzer labels the sensitive API calls identified by SuSi [31] as sources and labels all the methods that initiate ICCs as sinks. Second, for each of the ICC entry points, we investigate if the data extracted from intents can potentially flow out of the application (i.e., determine the ICC entry leaks). The dataflow analyzer labels the methods to access intent data (e.g., `getIntent`, and `onActivityResult`) as sources and labels all sinks identified by SuSi [31] as sinks.

2) *Retry with a less precise configuration*: We used two types of configuration for the dataflow analyzer.

High precision configuration: This configuration supports a context-sensitive algorithm with an access path length = 3. In this configuration, the dataflow analyzer builds the complete taint paths. (An access path is of the form $a.b.c$, where a is a local variable or parameter and b and c are fields. The variable $a.b.c$ has an access path length = 2. An access path length = 0 means a simple local variable or parameter (i.e., in this case a) [3].)

Low precision configuration: This configuration supports a context insensitive algorithm, which does not consider calling context. It is significantly faster, but may have false positives. In this configuration, the access path length is set to 1 and the dataflow analyzer only identifies the sources and sinks, but skips building the complete taint paths.

By default, the dataflow analyzer runs with the high precision configuration. However, if a precision-analysis fails to complete within our specified time (i.e., 5 minutes), DIALDroid abandons the analysis, and retries with the low precision configuration. To recover from possible deadlocks, we limit the analysis time for each app to 20 minutes (i.e., if the analysis for an app does not complete within 20 minutes, DIALDroid abandons that analysis).

3.3 Data Module and ICC Leak Calculator

The data module of DIALDroid aggregates the attributes of an app extracted by the ICC entry / exit point extractor (Section 3.1) and the dataflow analyzer (Section 3.2). The data module stores the aggregated data in a MySQL database. DIALDroid leverages the power of relational databases to overcome scalability issues. Relational databases provide efficient data storage. More importantly, modern relational database management systems facilitate powerful query capabilities to easily transform and retrieve data. DIALDroid uses a highly normalized database schema to efficiently store data and uses indexes on the comparison attributes to support efficient query computation. Our database is composed of 42 tables with a total of 161 attributes. A supplementary website[9] provides a detailed diagram of the database schema.

Algorithm 1 A SQL query to detect inter-app ICC based collusive data leaks.

```
SELECT sender.app AS senderapp, idl.method,
    idl.leak_path AS sender_app_path, receiver.app AS
    receiverapp, entryLeaks.leak_path AS
    receiver_app_path, entryLeaks.leak_receiver,
    icc_type FROM SensitiveChannels
INNER JOIN Applications sender ON
    SensitiveChannels.fromapp=sender.id
INNER JOIN Applications receiver ON
    SensitiveChannels.toapp=receiver.id
INNER JOIN EntryPoints ep ON
    ep.class_id=SensitiveChannels.entryclass
INNER JOIN ICCEntry_DataLeaks entryLeaks ON
    entryLeaks.entry_point_id=ep.id
INNER JOIN ICCExit_DataLeaks idl ON
    idl.exit_point_id=SensitiveChannels.exitpoint
LEFT JOIN Intent_Extras ON
    Intent_Extras.intent_id=SensitiveChannels.intent_id
 -- either no data is passed via putExtra
 WHERE (Intent_Extras.id IS NULL) or
 -- OR data is passed via putExtra and receiver path
     also contains the key
(Intent_Extras.extra IS NOT NULL AND
     entryLeaks.leak_path LIKE CONCAT
     ('%',Intent_Extras.extra,'%'))
```

We implement the key `calculateSensitiveChannels` procedure inside the database as a SQL stored procedure. This design minimizes potential data transmission delays and leverages the speed, optimization, and efficient queries provided by the database management systems. Because all the inter-app ICC threats in our attack model concern sensitive ICC channels (specifically, requiring ICC exit leaks in sender apps), it is unnecessary to compute ICC links for the intents that cannot possibly contain any sensitive information. It drastically reduces the computation complexity.

While matching explicit intents are straightforward, the resolution of an implicit intent involves matching the action, category and data fields with compatible `IntentFilter`, known in the Android development guide as action test, category test, and data test, respectively. We write SQL queries to compute all the sensitive ICC links originating via implicit intent from a specific app. Due to the complex matching rules, we create two SQL procedures: `categorytest(intent_id,filter_id)` and `datatest(intent_id,filter_id)`, which implement the category test and data test, respectively. Queries to compute ICC channels via explicit intents are much simpler. Algorithm 3 in the appendix shows the pseudocode for calculating sensitive ICC channels to and from an app.

For computing privilege escalations, we test if the receiver app in a sensitive ICC channel has permissions to access the data transmitted via the carried intent. For computing collusive data leaks, we check if a sensitive ICC channel is joining an ICC exit leak in an app

[9] https://github.com/dialdroid-android/dialdroid-db

with an ICC entry leak in another app. We show an example query for detecting collusive data leaks in Algorithm 1. In addition, our supplementary website provides scripts to generate all the SQL procedures and the SQL queries.

The intent resolution in DIALDroid is based on the libraries provided by IC3. However, in some cases, string analysis in IC3 cannot accurately determine possible values and therefore generates safe over-approximated sets (e.g., '.*', a regular expression matching any string constant) [28]. A recent study found that 95% of the ICC links generated by the intents with attributes (i.e., package, component, action, or category) resolved as '.*' were infeasible [26]. Therefore, the strict intent matching rules implemented by our ICC leak calculator ignores such over-approximated regular expressions. While this modification may introduce a few false negatives, it greatly reduces the number of false positives in the subsequent detection.

4. EVALUATION AND FINDINGS

Our evaluation aims to answer the following questions.

1. How does DIALDroid compare with other inter-app ICC analysis tools (namely IccTA+ApkCombiner and COVERT), in terms of both detection accuracy and runtime over benchmark apps? (In Section 4.1) Similarly, for conventional intra-app ICC analysis? (In Section B.1 in the appendix)
2. Are there explicit-intent based privilege escalation or collusive data leak pairs? How many cases are via implicit intent based ICCs? Which threat is more common, privilege escalation or collusive data leak? (In Section 4.2)
3. What are the detected app pairs and what do they leak? (Case studies in Section 4.3)
4. How many apps have ICC exit leaks? How many apps have the ICC entry leaks? What is the distribution of sensitive ICC channels across app categories? (In Sections 4.4 and 4.5)
5. What are the top 10 leaked permissions in privilege escalation cases? What categories of Google Play apps cause the most collusive data leaks? (In Section 4.6)
6. What are the reasons for unintended ICCs with mismatched data types? (In Section 4.7)
7. How long does DIALDroid take to analyze hundreds of thousands of real-world Android apps? (In Section 4.8)

In addition, we also released a benchmark consisting of real-world apps for comparing the detection capabilities for collusive data leaks. Unless specified, experiments were conducted on a Dell Tower Precision 7810 workstation running Ubuntu 14.04LTS 64bit with 16 core Intel Xeon 2.4GHz CPU, 64GB RAM, and an SSD drive. We enlisted four virtual machines for the large-scale experiment in 4.8.

We evaluate both real-world apps and benchmark suites. Our three datasets are described below.

- *Dataset I (Benchmarks).* We evaluate benchmarks below.

  DroidBench 3.0: DroidBench is the most comprehensive benchmark suite to evaluate the effectiveness of Android taint analysis tools. Among the 174 test cases provide by the DroidBench 3.0 [10], 10 test-cases aim to evaluate intra-app leaks and 11 test-cases aims to evaluate inter-app collusions.

  DroidBench (IccTA): IccTA introduced 23 test cases for intra-app leaks and 6 test-cases for inter-app leaks and are available in the IccTA branch [11] of the DroidBench.

  ICC-Bench: ICC-Bench [12], introduced by Amandroid [35], provides 11 test cases for Intra-app leaks. While ICC-Bench did not mention about inter-app leaks, we found and verified 9 inter-app leaks in ICC-Bench.

  For inter-app ICC analysis, our comparison is on 21 inter-app ICC test cases from these benchmark suites. We also evaluate 44 intra-app ICC test cases for completeness. A test case may contain multiple ICC leaks.

- *Dataset II (Google Play apps).* Dataset II consists of 100,206 most downloaded Android apps (as of June, 2015) belonging to 16 popular categories from Google Play. Table 7 in the appendix shows the distribution of the apps across the categories.

- *Dataset III (All real-world apps).* Dataset III (total 110,150 apps) consists of all apps from Dataset II as well as 9,944 malware apps from Virus Share.

4.1 Inter-app ICC Benchmark

Table 2 shows the benchmark comparison results of our inter-app ICC analysis. DIALDroid has the highest precision [13] (100%), the highest recall [14] (91.2%), and the highest F-measure (0.95) among the three tools. IccTA performed poorly (12.5% recall), mainly because ApkCombiner was unable to combine the majority of the app pairs (62%). For the successfully combined apks, IccTA can only detect the inter-app leaks that are in DroidBench-IccTA (i.e., the benchmark that was developed by the same authors). Due to inaccurate intent resolutions, COVERT reported a high number of false positives (323). COVERT failed to detect all inter-app leaks from the DroidBench 3.0.

We performed manual inspection on our failed cases. Among the 21 inter-app pairs, nine lead to privilege escalation. DIALDroid was able to detect five of those with a 100% precision and 55.5% recall. DIALDroid failed to report transitive (indirect) privilege escalations (i.e., data leaked via an intermediate component with the same level of permissions as the source component). In contrast, COVERT failed to report any of those nine privilege escalations.

We compare the inter-app analysis runtime of COVERT, IccTA+ApkCombiner, and DIALDroid, with 57 randomly selected apps from Google Play Market. Out of the 1,596 pairs, ApkCombiner was able to combine only 501 pairs (31%) and IccTA took 203 hours to complete on the combined apps. COVERT ran for 26 hours and then crashed during the formal model generation step [4]. In comparison, DIALDroid took 6.1 hours to complete. It only abandoned two apps, as DIALDroid was unable to finish within 20 minutes during those two cases.

For completeness, benchmark evaluation on intra-app ICCs is described in Section B.1 in the appendix.

4.2 Threat Breakdown for Dataset II

We break down the threats into six disjoint categories, which are listed as threat types I to VI in Table 3. The categories are disjoint in that an inter-app ICC belongs to one and only one category. Some sender apps may appear in ICCs of multiple categories. We then run

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

[11] https://github.com/secure-software-engineering/DroidBench/tree/iccta

[12] https://github.com/fgwei/ICC-Bench

[13] Precision is the percentage of identified cases that are true leaks.

[14] Recall is the percentage of present leaks that are detected.

Table 2: Comparisons on inter-app ICC analysis with DroidBench 3.0, DroidBench (IccTA branch), and ICC-Bench. Multiple circles in one row means multiple inter-app collusions expected. An all-empty row: no inter-app collusions expected and none reported. †indicates the tool crashed on that test case.
✓⃝= a correct warning, *= a false warning, ◯= a missed leak, Ⓟ= a privilege escalation reported, ‡= did not test or N/A.

Source App	Destination App	# ICC Exit Leaks (Dest.)	# ICC Entry Leaks (Sink)	Privilege Escalation	COVERT	IccTA + ApkCombiner	DIALDroid (Ours)
			DroidBench 3.0				
SendSMS	Echoer	1	3	✓	◯	◯	✓⃝Ⓟ
StartActivityForResult1	Echoer	2	3	✓	◯◯	◯◯	✓⃝✓⃝Ⓟ
DeviceId_Broadcast1	Collector	2	1	✓	◯	◯†	✓⃝Ⓟ
DeviceId_ContentProvider1	Collector	2	1	✓	◯	◯†	✓⃝
DeviceId_OrderedIntent1	Collector	3	1	✓	◯	◯†	✓⃝
DeviceId_Service1	Collector	1	1	✓	◯	◯†	◯
Location1	Collector	2	1	✓	◯◯	◯◯†	✓⃝✓⃝Ⓟ
Location_Broadcast1	Collector	3	1	✓	◯◯	◯◯†	✓⃝✓⃝Ⓟ
Location_Service1	Collector	2	1	✓	◯	◯†	◯
Incorrect app pairings					(172 *)	‡	
			DroidBench (IccTA branch)				
startActivity1_source	startActivity1_sink	1	2		✓⃝	✓⃝	✓⃝
startSevice1_source	startService1_sink	1	2		✓⃝	✓⃝	✓⃝
sendbroadcast1_source	sendbroadcast1_sink	1	2		✓⃝	✓⃝	✓⃝
Incorrect app pairings					(104 *)	‡	
			ICC-Bench				
implicit_action	implicit_src_sink	1	1		✓⃝	◯†	✓⃝
implicit_action	implict_nosrc_sink	1	1		✓⃝	◯	✓⃝
implicit_mix1	implicit_mix2	1	1		◯	◯†	✓⃝
implicit_mix2	implicit_mix1	1	2		✓⃝	◯†	✓⃝
implicit_src_nosink	implicit_src_sink	1	1		✓⃝	◯	✓⃝
implicit_src_nosink	impliict_nosrc_sink	1	1		✓⃝	◯	✓⃝
implicit_src_nosink	implicit_action	1	1		✓⃝	◯	✓⃝
impilicit_src_sink	implicit_action	1	1		✓⃝	◯†	✓⃝
impilicit_src_sink	implicit_nosrc_sink	1	1		✓⃝	◯	✓⃝
Incorrect app pairings					(47 *)	‡	
			Sum, Precision, Recall, and F measure				
True positive (✓⃝), higher is better					11	3	22
False positive (*), lower is better					323	0‡	0
False negative (◯), lower is better					12	20	2
Precision, *p* = ✓⃝/(✓⃝+*)					**3.3%**	**100%**‡☛	**100%**
Recall, *r* = ✓⃝/(✓⃝+◯)					**45.8%**	**12.5%**	**91.2%**
F-measure = *2pr/(p+r)*					**0.06**	**0.22**	**0.95**

☛ Since we were unable to execute IccTA+ApkCombiner on most of the pairs, it's precision value is misleading and does not reflect it's actual performance.

DIALDroid on Dataset II (Google Play apps). For each threat type, we summarize our findings in Table 3. Because Google Play market is known to deploy app vetting mechanisms (e.g., Google Bouncer), it is reasonable to assume the apps in Dataset II have passed some single-app screenings.

We found no collusive data leaks or privilege escalations based on explicit intents, i.e., no inter-app ICCs of Threat Types I, II, III. This result suggests that explicit intent based collusion is very rare. (They might exist, but are out of the scope of our dataset.) Therefore, collusion analysis needs to be focused on implicit intents based ICCs, as opposed to explicit intents.

For inter-app ICCs via implicit intents, we distinguish three cases: both collusive data leak and privilege escalation in Threat IV, privilege escalation without collusive data leak in Threat Type V, and collusive data leak without privilege escalation in Threat VI. We highlight some key results next. The most severe threat type is Threat IV, where collusive data leak and privilege escalation occur simultaneously. We found 16,712 app pairs originating from 33 sender apps that exhibit both collusive data leak and privilege escalation behaviors via implicit intents. Because of the sensitive data from the sender app is leaked externally by the receiver app and the receiver app is under the disguise of having fewer permissions, apps in Threat IV is the most serious.

It is not surprising that we observe a huge number (1,785,102) of inter-app ICC channels with the privilege escalation threat in Threat V. Some app pairs may have multiple ICC channels between them. Interestingly, these 1,785,102 ICCs with privilege escalation threat originate from only 62 problematic sender apps.

For Threat Type VI (collusive data leak without privilege escalation), we found 6,783 such app pairs originating from 21 sender apps. That is, these app pairs exhibit collusive data leak behaviors; however, the receiver apps do not gain new permission privileges, i.e., the receiver apps have the authorization to access the received data. In addition, we found that a large number (20) of sender apps in Threat Type VI are also sender apps in Threat Type IV. We performed case studies for each of the Threat Types IV, V, and VI in Section 4.3. Some cases in Threat Type VI suggest that the collusive data leaks are unintentional.

Cases with HTTP and SMS Sinks: We want to identify the collusive data leak cases (of Threat Types IV or VI) that exfiltrate the sensitive information to remote destinations. We recompute the results with a small set of relevant sensitive sink methods, namely `java.net.URL`, `android.telephony.SmsManager`, and `org.apache.http.HttpResponse`. For Threat Type IV, we found 325 problematic app pairs with 16 distinct sender apps and 32 distinct receiver apps. There are a total of 1,054 Type IV ICCs. For Threat Type VI, the numbers are smaller. We found 19 pairs, with 12 senders and 3 receivers, and a total of 63 problematic ICCs. Our

Table 3: Summary of problematic inter-app ICC channels in each threat category. Sender apps and receiver apps are from Google Play Market. All the ICC channels shown are sensitive with ICC exit leaks in the sender apps (as defined in Section 2). Privilege escalation and collusive data leak are defined in Section 2.

Categorization				**Results**			
Threat Type	**Collusive Data Leak**	**Privilege Escalation**	**Intent Type**	**# of Distinct Source App**	**# of Distinct Receiver App**	**Total ICC Channels**	**Total App Pairs**
I	Yes	Yes	Explicit	0	0	0	0
II	No	Yes	Explicit	0	0	0	0
III	Yes	No	Explicit	0	0	0	0
IV	Yes	Yes	Implicit	33	1,792	77,104	16,712
V	No	Yes	Implicit	62	44,514	1,785,102	1,032,321
VI	Yes	No	Implicit	21	1,040	34,745	6,783

case study in Section 4.3 gives an example SMS-based collusive data leak.

Same-developer Privilege Escalations: We found 200 inter-app ICCs with same-developer privilege escalation. Same developer refers to that the sender and receiver apps have the same developer name. All such cases belong to Threat Type V and are related to location permissions (both fine and coarse). 194 ICCs appear somewhat benign, as only the country name (`getCountry()`) is involved. However, the other 6 privilege escalation ICCs appear more serious. They involve 3 pairs of apps (1 pair from Alamex Ltd and 2 pairs from NexTag Mobile) and specific locations (`getLastKnowLocation()`). Our case study in Section 4.3 gives an example of the same-developer privilege escalation cases.

4.3 Case Studies

Threat TYPE IV [escalation w/ collusive data leak]

1) `com.ppgps.lite`→`de.ub0r.android.websms`. The source app provides the real-time flight information to the pilots of paramotor, paraglider, glider or ultra light planes. `com.ppgps.PPGpSActivity` retrieves a user's location (i.e., `getLastKnownLocation`) and sends it via an implicit intent (action = `android.intent.action.VIEW`, Mime-Type= `vnd.android-dir/mms-sms`). The sink app lets a user send free or low-cost SMS messages via various web services. `de.ub0r.android.websms.WebSMS` defines an intent-filter to accept the above intent. Upon receiving the intent, the WebSMS activity retrieves and parses the data sent via `sms_body` field and leaks it via SMS to a phone number. Since the sink app does not have the permission to access location, it leads to both privilege escalation and collusive data leak.

2) `com.codalata.craigslistchecker`→`qubecad.droidtocad`. The source app helps users search craiglist worldwide. `com.codalata.craigslistchecker` retrieves SIM serial number (i.e., `getSimSerialNumber`, permission =`android.permission.READ_PHONE_STATE`) and sends via implicit intent (action= `android.intent.action.SEND`, Mime-Type=`plain/text`). The sink app is a location recording app. `qubecad.droidtocad.activities.AddDocumentActivity` defines an intent-filter to accept the above intent. Upon receiving the intent, the AddDocumentActivity activity retrieves and parses the data sent via `android.intent.extra.TEXT` field and leaks it to a log. Since the sink app does not have permission to access phone state, it leads to both a privilege escalation and a collusion.

Threat TYPE V [escalation w/o collusive data leak]

1) Same developer: `com.nextag.android`→`com.thingbuzz`. Both apps are developed by the Nextag Mobile. The sender app compares price across different e-commerce sites. `com.nextag.android` retrieves the user's location(i.e., `getLastKnownLocation`, permission =`android.permission.ACCESS_FINE_LOCATION`) and sends that via an implicit intent (*action* = `android.intent.action.MAIN`, *category* =`android.intent.category.INFO/ android.intent.category.LAUNCHER`). The receiver app, which provides shopping advice to users, defines an intent-filter to accept the above intent. However, `com.thingbuzz` does not have the permission to access user's location, this ICC communication leads to escalated privileges.

2) Different developers: `com.biganiseed.ladder.trial` → `ee.showm`. The sender app provides a VPN connection. `com.biganiseed.ladder.trial` retrieves network information (i.e., `getActiveNetworkInfo`, permission =`android.permission.ACCESS_NETWORK_STATE`) and sends that via an implicit intent (*action* = `android.intent.action.SENDTO`). The receiver app, which controls EE TV, defines an intent-filter to accept the above intent. However, `ee.showm` does not have the permission to access network information, this ICC communication leads to escalated privileges.

Threat TYPE VI [collusive data leak w/o escalation]

`com.ccmass.fotoalbumgpslite`→`com.ventri.cake.retrica`. The sender app organizes photos based on the locations where photos were taken. `com.ccmass.fotoalbumgpslite` retrieves user's location (i.e., `getLatitude` and `getLongtitude`, permission =`android.permission.ACCESS_FINE_LOCATION`) and sends that via an implicit intent (*action* = `android.media.action.IMAGE_CAPTURE`). The receiver app, which takes photos with various filters, defines an intent-filter to accept the above intent. Since `com.ventri.cake.retrica` have the permission to access location information, this ICC communication does not lead to escalated privileges. But upon reception, `com.ventri.cake.retrica` leaks the data to a log and therefore causes a collusion.

Threat TYPE IV, V, VI [vulnerable sender app]

App `com.koranto.mkmn` provides prayer times for Muslims around the world. The MainActivity of `com.koranto.mkmn.activities` retrieves the user's location (i.e., `getLastKnownLocation`, permission =`android.permission.ACCESS_FINE_LOCATION`) and sends it via an implicit intent (*action*=`android.intent.action.SEND`, *MimeType* = `text/plain`).

We found 1,540 receiver apps that can possibly accept this intent. Among those possible receivers, 32 apps and the resulting inter-app ICCs exhibit both collusive data leak and privilege escalation behaviors (Type IV), 839 apps and the resulting ICCs exhibit only privilege escalations (Type V), and 7 apps and the resulting ICCs exhibit only collusive behaviors (Type VI).

For example, `br.com.coderev.acumapa`, which provides an acupuncture map overlaid on the image captured by the camera, can receive this intent and write the retrieved location information to a file (Type IV). `do.adoubleu.toy`, which is an integrated diary and messenger app without access to user's location, can accept this intent (Type V). `com.du.android`, which is a to-do list management app with access to user's location, can accept this intent, extract location information sent via `android.intent.extra.TEXT`, and leak it to a log (Type VI).

It is extremely challenging for us to infer the *true* intentions behind these implicit-intent based collusive data leak or privilege escalation behaviors. Is the developer's intention malicious (e.g., for deliberately evading detection or stealing sensitive data) or benign (e.g., due to poor programming practices)? We further discuss the security implications in Section 5.

4.4 Statistics on ICC Exit and Entry Leaks

For Dataset III, the number of sender apps with ICC exit leaks is an order of magnitude fewer than the number of receiver apps with ICC entry leaks. Specifically, DIALDroid identified a total of 30,453 ICC exit leaks that are caused by 3,372 sender apps (3.06% of the total apps). DIALDroid identified a total of 249,263 ICC entry leaks that are caused by 32,855 receiver apps (29.82% of the total apps).

Out of the 3,372 sender apps with ICC exit leaks, 1,792 of them ($\approx$ 1.62% of total apps) initiate sensitive ICC channels (more information in Section 4.5). Although it does not necessarily mean that the remaining apps are threat-free, as they may communicate with apps outside of our dataset, the number of problematic sender apps is somewhat surprisingly small. However, because of the use of implicit intents in the inter-app ICCs, these 1,792 sender apps generate millions of ICC links (presented in Section 4.5).

Figure 1 in the appendix shows the percentages of leaking apps out of each app category. For Google Play apps, Personalization has the highest percentage of apps with ICC exit leaks (in sender apps), which is only slightly lower than the Virus Share category. For ICC entry leaks in receiver apps, the percentages are rather high across all the Google Play app categories, with Photography and Business being the highest.

4.5 Statistics on Sensitive ICCs

For Dataset III, DIALDroid found 5,715,046 ($\approx$ 5.7 million) potentially sensitive ICC channels. Most of the ($\approx$ 99.6%) sensitive ICC channels are inter-app, and the rest are intra-app. These sensitive ICC channels originate from only 1,792 apps.

Table 4 shows how the sender apps involved in sensitive ICC channels or collusive data leaks are distributed across different app categories for Dataset III. Intuitively, this table summarizes the problematic sender apps and their categories. We highlight the categories with at least one percentage over 7%.

For Google Play apps, Transportation (11.18%) and Travel & Local (9.05%) apps initiate the most sensitive ICC channels, which is most likely due to passing the user's location information to another app. In contrast, this category has a relatively low percentage of collusive data leak cases, which indicates the location or other sensitive information being passed is likely consumed by the receiver app, as opposed to being leaked via disk output or network output. Personalization and entertainment categories have high percentages of problematic sender apps for both types of inter-app ICC threats.

In comparison, sender apps from Virus Share are involved in a substantially higher number of detected sensitive ICC channels and collusive data leaks, which is expected. Although they account for 9.03% of the apps in Dataset III, 43.18% of the sensitive ICC channels and 52.33% of the collusive data leaks are originated from apps in Virus Share. The high percentage (52.33%) of collusive data leaks originating from malware apps indicates that malware apps actively seek and transfer sensitive information.

Table 4: The distribution of sensitive ICC channels and collusive data leaks among app categories for Dataset III. An app may have multiple sensitive ICC channels.

Category	**% of total Apps**	**% sensitive ICC channels (origin)**	**% collusive data leaks (origin)**
Books & Reference	7.40%	0.01%	0.00%
Business	5.40%	0.00%	0.00%
Comics	1.87%	0.00%	0.00%
Communication	0.05%	0.05%	0.04%
Entertainment	7.43%	4.49%	**8.97**%
Lifestyle	6.69%	0.03%	0.11%
Medical	1.64%	0.00%	0.00%
Personalization	6.75%	**17.09**%	**13.31**%
Photography	7.30%	4.33%	6.35%
Productivity	6.88%	0.01%	0.59%
Shopping	5.75%	2.16%	1.57%
Social	6.24%	3.23%	1.84%
Sports	6.40%	2.14%	4.41%
Tools	7.36%	2.66%	**7.16**%
Transportation	5.74%	**11.18**%	3.29%
Travel & Local	3.04%	**9.05**%	0.02%
Virus Share	9.03%	**43.18** %	**52.33**%

Table 5: Top permissions leaked via privilege escalation in Dataset III.

Permission	**# Cases**
android.permission.ACCESS_FINE_LOCATION	1,155,301
android.permission.ACCESS_COARSE_LOCATION	1,163,769
android.permission.READ_PHONE_STATE	880,645
android.permission.ACCESS_WIFI_STATE	433,887
android.permission.ACCESS_NETWORK_STATE	486
android.permission.BLUETOOTH	153
Total:	3,634,241

4.6 Permission and Method Distributions

Table 5 shows the number of different permissions leaked via all privilege escalation scenarios for Dataset III. Recall that Dataset III includes Google Play apps and apps from Virus Share. The results suggest that user's location, device information, and current cellular network information are overwhelmingly more likely to be transferred to apps that do not have corresponding access permissions. The permission ACCESS_NETWORK_STATE gives the app authorization to access NetworkManager to monitor network connections, which is useful for device fingerprinting. Similarly, the permission ACCESS_WIFI_STATE provides the access to WifiManager and can be used for fingerprinting.

Table 6 (first two columns) shows the most common sensitive source methods in collusive data leak cases in Dataset III. Methods to uniquely identify a user (i.e., `getDeviceId`, `getConnectionInfob` and `getSubscriberId`) are the most common sources of ICC leaks. Other common sources include methods to retrieve a user's location (i.e., `getLastKnownLocation`, `getLatitude`, and `getLongtitude`). Similarly, Table 6 (last two columns) shows the most common sensitive sink methods. `SharedPreferences` and `Log` are the mostly used for collusive data leaks. Other APIs are related to file, network, and SMS. In Section 5, we discuss how relaxing sensitive source and sink definitions impacts the results.

Table 6: Top sensitive source and sink methods involved in collusive data leaks in Dataset III.

Sensitive Source	%	Sensitive Sink	%
getDeviceId	36.69%	android.content.SharedPrefs	49.0%
getConnectionInfo	33.44%	android.util.Log	48.3%
getSubscriberId	4.36%	java.io.OutputStream	1.1%
getLastKnownLocation	4.32%	java.net.URL	0.9%
getLongitude	4.18%	java.io.FileOutputStream	0.7%
getLatitude	4.03%	org.apache.http.HttpResponse	0.1%
getSimSerialNumber	3.09%	android.telephony.SmsManager	0.03%
getLine1Number	2.78%		
getActiveNetworkInfo	2.10%		
getCountry	1.35%		
others	3.65%		

4.7 Unintended ICCs & Inaccurate Manifests

The main source of false positives in our detection is unintended ICCs with mismatched data types. We randomly selected 10 app pairs (4 of Type IV, 5 of Type V, and 1 of Type VI) and manually investigated their decompiled source code. The pairs have distinct receiver apps. We found 5 receiver apps overclaim the types of data it can receive in their Manifest files. For those apps, the sensitive intents pass our static action test, category test, and data test, which are equivalent to Android's runtime tests. However, the code in the receiver app is not designed to process the sensitive incoming intent. At runtime, the receiver app may crash or simply do nothing. For example, for a pair with Threat Type IV, we found that the source app `com.americos.selfshot` sends implicit intent with data field "android.intent.extra.TEXT" containing user's device ID. However, the code in a matched receiver app `qubecad.droidtocad` assumes the data field in the incoming intents to be file paths. This suggests that this inter-app ICC is not intended. Unintended ICCs with mismatched data types may lead to false positives.

The fundamental reason for these false positives is loose or no restrictions on incoming data in the receiver app's Manifest file. If the format of incoming data is not well specified, then Android system is likely to assume the receiver app can receive all types of data at runtime. Judging based on our manual analysis, such cases are quite common. We discuss this issue further in Section 5.

4.8 Runtime on 110K Apps

For scalability evaluation, we measure how long DIALDroidtakes to analyze our largest dataset, Dataset III with 110,150 apps. We used four virtual machines, each with 4 processor-cores, 64GB RAM, and 1 TB hard drive to analyze the apps. We stored the results to a MySQL database hosted on a server with an eight-core processor and 80GB RAM. The ICC Leak Calculator module of DIALDroid computed all the sensitive ICC channels among the 110,150 apps in 82 minutes. This computation is fast, because although the total number of ICC links is huge, the percentage of sensitive ones is extremely low (about 0.57% as estimated by our experiment. [15] Non-sensitive entries are not touched in the computation. Our relational database schema is efficient and consumes only 6.3 GB space for storing the information for 110,150 apps.

DIALDroid was able to analyze more the 80% of the apps within five minutes. The average analysis time per app was 3.45 minutes. Figure 3 in the appendix shows the distribution of analysis time for the applications. Adding the individual analysis time for each app (i.e., as if all the apps were analyzed on a single machine), DIALDroid took a total of 6,339.6 hours to analyze the 110,150 apps. DIALDroid was able to complete 83.6% of the apps with a high precision configuration within five minutes (Section 3.2). For 10.7% of the apps, DIALDroid timed out in high precise configuration but was able to analyze successfully within five minutes when retried with a low precision configuration (Section 3.2). For the remaining 5.7% of the apps, DIALDroid failed to complete the analysis within the specified execution limit (20 minutes). Table 7 in the appendix shows statistics of the apps and our program analysis.

New benchmark released. In order to validate the detected collusion pairs and privilege escalations, we inspected the taint paths reported by DIALDroid. We further validated the leaks through manual inspections on the code. We converted the `.apk` files to `.jar` files using the `dex2jar` [16] tool. We decompiled the `.jar` files to Java source code using a Java decompiler[17]. We manually inspected the source codes to verify leaks. Based on our manual verification, we have compiled a benchmark suite, DIALDroid-Bench[18], to test inter-app collusion. Currently, the suite contains 30 *real-world* apps from the Google play. To our knowledge, this is the first such benchmark using real-world apps, as opposed to proof-of-concept apps.

5. DISCUSSION AND LIMITATIONS

Unintentional leaks and escalations. Although the reported collusive data leak and permission escalation cases may be unintentional (e.g., due to insecure design or poor development practices), these apps still pose threats to user's sensitive data and device. Several lessons can be learned by developers in order to prevent or reduce such threats. ICC sender apps should avoid transferring sensitive data through `Activity` or `Service` based implicit intents. Permission checking is needed for `Broadcast` intent with sensitive information. Whenever possible, explicit intents are preferred for communicating sensitive data between apps. For receiver apps, enforcing strict restrictions for each entry point (e.g., add `pathPattern` in `intentFilter`) reduces unintended and unexpected ICCs.

Sensitive source and sink definitions. The choice of sensitive source and sink impacts the number of reported ICC anomalies. A smaller set of sensitive sources and sinks generates a smaller number of alerts. For example, as shown in Table 6, `android.util.Log` accounts for 48.3% of the sensitive sinks (in receiver apps) in the detected ICC leaks. When excluding both Log and SharedPreferences from the sensitive sink list, our query returns a much reduced number (15,109) of collusive ICC links.

Our sensitive sources and sinks definitions follow SuSi [31], which includes Android logging and SharedPreferences. In the latest Android OS, the logged information is visible only to the app itself, which reduces its risk. However, advanced logging-based exploits (e.g., `LogCat` and `CatLog`) are still possible. Thus, our evaluation includes logging as a sensitive sink in our evaluation. SharedPreferences are key-value pairs maintained by the Android system. An app can read and write the value associated with the

[15] We first computed all the possible ICC links originating from 1,000 randomly selected applications and obtained ≈ 21.8 million ICC links originating from those 1,000 apps. Among those ≈ 21.8 million ICC links, only 124K (≈ 0.57%) ICC links were sensitive, i.e., sensitive ICC channels as defined in Section 2. The rest of them do not carry sensitive data.

[16] https://github.com/pxb1988/dex2jar

[17] http://jd.benow.ca/

[18] https://github.com/dialdroid-android/dialdroid-bench/

key. There are three modes for SharedPreferences: private (i.e., accessible only by the owner app), world-read (i.e., others can read), and world-write (i.e., others can read and write). In virtually all taint analyses, SharedPreference is labeled sensitive. Even if it is configured as private, other components of the same app can access it resulting in sensitive data flows.

The advantage of DIALDroid is that its database backend allows security analysts to easily adjust and customize sensitivity definitions to refine query results. In Section 4.2, we show the recomputed results with a much smaller sink set consisting of `java.net.URL`, `android.telephony.SmsManager`, and `org.apache.http.HttpResponse`.

App chain length. Three or more apps can possibly create a chain with ICC links to leak data. For example, three apps, A, B, and C, create an ICC chain, where app A transfers sensitive information to app B via an ICC exit leak; app B then leaks that information to app C. A chain of three or more apps is a special case of two app-based ICC collusion, where the receiver app leaks data extracted from an intent by initiating another ICC (i.e., ICC entry leak with ICC initiation methods as sink). Therefore, DIALDroid reports A→B link described above as an inter-app collusion.

Among the three benchmarks evaluated in Section 4, DIALDroid identified following two scenarios in the ICC-Bench, where three components work together to leak sensitive information.

1. $implicit5.MainActivity \rightarrow implicit5.FooActivity \rightarrow implicit5.HookActivity$
2. $implicit6.MainActivity \rightarrow implicit5.FooActivity \rightarrow implicit5.HookActivity$

Although we did not find any chain of more than two components among the 110K real-world apps, DIALDroid is capable of identifying such chains.

Risk prioritization. Security analysts need usable tools to prioritize the investigation of reported threats. Because of the quadratic growth of possible inter-app ICCs in the number of apps, this prioritization is a key to the usability. Relaxing the definitions of sensitive sources and sinks (i.e., smaller sets) reduces the number of alerts generated. In addition, quantitative metrics can be developed to prioritize the risks based on the type of inter-app sensitive ICC flows, through machine learning methods. PRIMO [26] can also be utilized to triage the ICC links detected by DIALDroid.

User applications. Although DIALDroid is for marketplace owners, Android users can also benefit from this tool. For example, enterprise users can check possible inter-app collusions using DIALDroid before allowing certain apps to be installed on the devices of their employees. Moreover, a large-scale public database similar to ours, when regularly updated, can be queried by users to find out possible inter-app communications to or from a particular app.

Limitations. Existing static analysis approaches are ineffective against the unintended ICCs problem caused by mismatched data (described in Section 4.7). The reason is that one needs to *infer* the intended data type that an app sends or receives based on how the code preceding or following an ICC. Such static semantic code inference is challenging and remains an open problem.

Similar to most other approaches based on static analysis, our approach shares some inherent limitations. For example, DIALDroid can resolve reflective calls only if their arguments are string constants. As mentioned in Section 3.3, since our strict intent matching rules ignore overapproximated regular expressions, DIALDroid may fail to compute some ICC links.

As we have mentioned in Section B.1, DIALDroid loses field sensitivity when intent objects carrying sensitive information goes through ICC channels, which can result in false positive collusion identification. DIALDroid uses a regular expression string search within the ICC entry leak path for the source data keys. As we encountered in `startActivity6` test case, this search may return false positives if the path contains any string that contains the key as a substring. We manually inspected 30 taint paths from real-world collusion pairs identified in our study and did not observe any such occurrence.

To enable large scale analysis, we limited our analysis time per app. Although DIALDroid failed to analyze only 5.7% of the applications within allocated time (i.e., 20 minutes), there is a possibility that some of those applications could cause collusions.

6. RELATED WORK

Collusion and privilege escalation. Davi *et al.* were the to describe the possibility of privilege escalation attacks in Android [10] and Marforio *et al.* gave a comprehensive description of possible collusion channels, including inter-app ICC [24]. Later ComDroid provided the first comprehensive analysis of inter-app ICC based threats, including broadcast theft and activity hijacking [9]. Since ComDroid analyzed individual apps, it's results may overapproximate the number of sensitive inter-app ICC flows, regardless of how the data is consumed by the receiver app. In comparison, our pairwise analysis performs *end-to-end* data-flow analysis, which is more fine-grained. Requiring the receiver app to have ICC entry leaks (as defined in Section 2) reduces the number of false positives (i.e., false alarms) allowing security analysts to better prioritize their investigation.

Elish *et al.* pointed out that collusion detection solutions may suffer from high false positives without in-depth pairwise data-flow analysis [13]. Researchers pointed out a third type of inter-app ICC attacks besides collusions and privilege escalation, called private activity invocation due to the misconfiguration of the intent scopes [33]. A recent study also reported the presence of collusive attacks to promote the rankings of apps in the Chinese Apple marketplace [8].

Single-app security. For static analysis on single apps, many general-purpose solutions such as DroidSafe [15] and Amandroid [35] can identify sensitive data flows. Researches have also proposed several other techniques for specific detection purposes. For example, CHEX [23] is focused on detecting data flows that enable component hijacking within a single app. AppIntent uses symbolic execution to determine whether a data transmission is intended by the user through analyzing its compatibility with the required GUI-operation sequences [39]. AAPL [22] utilizes peer voting for privacy leakage detection together with data-flow analysis. Wolfe *et al.* uses supervised learning to classify malware families [36].

Several researchers have also used dynamic program analysis for screening single apps. TaintDroid [14] dynamically tracks the information flows and detects privacy leaks through Android system instrumentation. INTENTDROID [16] utilizes debug breakpoints to dynamically detect the unsafe handling of incoming messages to identify possible component hijacking. IntentFuzzer uses fuzzing framework to identify exposed and vulnerable interfaces [38]. IntelliDroid aims to generate inputs for dynamic analysis [37].

App-pair security. Most of the dynamic analysis solutions modify Android system to enforce security policies to prevent inter-app threats. XmanDroid is the first among such tools to demonstrate runtime collusion detection by enforcing policies on the combined permission set of app-pairs [5]. FlaskDroid enforces mandatory access control policies to prevent privilege escalation and collusion attacks [7]. IntentScope enforces security policies during dynamic intent forwarding [17].

However, these dynamic analysis based solutions are designed to analyze a small set of apps (e.g., ones that are installed on the same phone). However, these approaches do not scale to hundreds of thousands of apps. In comparison, our solution is designed for security analysts who maintain large-scale app marketplaces or even medium-scale proprietary marketplaces owned by an organization for its employees, e.g., only approved apps from the internal app marketplace are allowed to be installed.

Among the static analysis based solutions, IccTA+ApkCombiner uses a straightforward approach by two apps into a single app (e.g., using ApkCombiner [20]) and then apply the existing single-app static analysis (e.g., IccTA [21]) to identify inter-app threats. However, this approach performs a large number of redundant program analyses, which significantly slows down the computation as shown in our experiments. Our evaluation also shows that the combination mechanism of ApkCombiner is fragile and failed on majority of the app-pairs. DidFail [19] and COVERT [4]) perform the data-flow analysis only once per app. COVERT uses formal methods (namely model checking) to detect suspicious inter-app ICC flows [4] but we found COVERT's formal model generation process fragile and having low scalability. DidFail [19] uses an approach very similar to us, however the intent resolution and intent-matching process of DidFail performs poorly. Moreover, DidFail did not enforce security policies to reduce the search space (e.g., we only match intents that can potentially carry sensitive information), and therefore is not scalable. Finally, DidFail does not limit dynamically adjust the precision of static taint analysis, therefore often fails to complete analysis of apps even after long time. In comparison, we implement a pragmatic adaptive mechanism that dynamically determines the accuracy-performance tradeoff during static taint analysis. FUSE is aimed towards single-app analysis, but can be extended to build a multi-app information-flow graph [32]. FUSE's intra-procedural string analysis is limited and error-prone. Existing inter-app analysis tools were evaluated on tens or hundred of apps and none of the tools were evaluated on 110K apps like DIALDroid.

PRIMO estimates the likelihoods of inter-app ICC connections using a probabilistic technique and provides ICC-link probabilities computed based on empirical evidence [26]. Although PRIMO is not designed to be a complete ICC security detection tool, it provides useful complementary information to security analysts to focus on the risky ICCs that are the most likely to occur in practice.

Others. Researchers proposed automatic patch generation for mitigating hijacking [41]. Similarly, applying third-party security patches for privilege escalation and capability leaks was proposed by Mulliner *et al.* [25]. Kantola *et al.* developed a heuristic policy to guide developers in writing safer apps [18]. Zhang *et al.* proposed monitoring network activities to identify stealth malwares [40].

7. CONCLUSIONS AND FUTURE WORK

We reported our findings in a large-scale inter-app ICC analysis for detecting collusions and privilege escalations. Accuracy and scalability are our key features, which we achieved through a new general-purpose Android intent resolution tool, database query systems, and pragmatic program-analysis execution management. Besides superior accuracy and runtime compared with state-of-the-art solutions, our analysis produces a number of real-world collusive data leak and privilege escalation pairs and a myriad of interesting statistics on ICC security. We have open-sourced our entire toolsuite on GitHub[19] and have made our database available[20] for other researchers.

[19] https://github.com/dialdroid-android/

[20] http://amiangshu.com/dialdroid/

Acknowledgments

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

8. REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining API-level features for robust malware detection in Android. In *Proc. of the Security and Privacy in Communication Networks*, pages 86–103. 2013.

[2] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[3] 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 *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[4] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. COVERT: Compositional analysis of Android inter-app permission leakage. *IEEE Transactions in Software Engineering*, 41(9):866–886, 2015.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[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 *Proc. of the 22nd USENIX Security Symposium*, pages 131–146, 2013.

[8] H. Chen, D. He, S. Zhu, and J. Yang. Toward Detecting Collusive Ranking Manipulation Attackers in Mobile App Markets. In *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS' 17)*, 2017.

[9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proc. of the 9th International Conference on Mobile systems, Applications, and Services*, pages 239–252, 2011.

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proc. of the International Conference on Information Security*, pages 346–360, 2011.

[11] E. Duan. DressCode and its potential impact for enterprise, September 2016. http://blog.trendmicro.com/trendlabs-security-intelligence/dresscode-potential-impact-enterprises/.

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

[13] K. O. Elish, D. Yao, and B. G. Ryder. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *Proc. of the IEEE Mobile Security Technologies (MoST)*, 2015.

[14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P.

Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[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 *Proc. of the Network and Distributed System Security Symposium*, 2015.

[16] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proc. of the ACM International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[17] Y. Jing, G.-J. Ahn, A. Doupé, and J. H. Yi. Checking intent-based communication in Android with intent space analysis. In *Proc. of the ACM Asia Conference on Computer and Communications Security*, pages 735–746, 2016.

[18] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *Proc. of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 69–80, 2012.

[19] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proc. of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[20] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon. Apkcombiner: combining multiple android apps to support inter-app analysis. In *IFIP International Information Security Conference*, pages 513–527. 2015.

[21] L. Li, A. Bartel, T. F. D. A. Bissyande, 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 *Proc. of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 280–291, 2015.

[22] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012.

[24] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proc. of the Annual Computer Security Applications Conference*, pages 51–60, 2012.

[25] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. PatchDroid: Scalable third-party security patches for Android devices. In *ACM Annual Computer Security Applications Conference*, pages 259–268, 2013.

[26] 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 *Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 469–484, 2016.

[27] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 6:1–6:11, 2012.

[28] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 77–88, 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: An essential step towards holistic security analysis. In *Proc. of the 22nd USENIX security symposium*, pages 543–558, 2013.

[30] 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. of the ACM conference on Computer and communications security*, pages 241–252, 2012.

[31] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[32] 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. of the ACM Program Protection and Reverse Engineering Workshop*, 2014.

[33] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic detection of inter-application permission leaks in Android applications. *IBM Journal of Research and Development*, 57(6):10–1, 2013.

[34] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proc. of the International conference on compiler construction*, pages 18–34, 2000.

[35] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1329–1341, 2014.

[36] B. Wolfe, K. Elish, and D. D. Yao. Comprehensive behavior profiling for proactive android malware detection. In *Proc. of the International Conference on Information Security*, pages 328–344. Springer, 2014.

[37] M. Y. Wong and D. Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[38] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. IntentFuzzer: Detecting capability leaks of Android applications. In *ACM Symposium on Information, Computer and Communications Security*, pages 531–536, 2014.

[39] 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 *Proc. of the ACM SIGSAC conference on Computer and communications security*, pages 1043–1054, 2013.

[40] H. Zhang, D. Yao, N. Ramakrishnan, and Z. Zhang. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security*, 58:180–198, 2016.

[41] M. Zhang and H. Yin. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

APPENDIX

A. ALGORITHM PSEUDOCODE

Algorithm 2 DIALDroid's algorithm for computing ICC leaks to / from an Android app

```
1: System DIALDROID($apkFile$)    ▷ Input: An Android apk file to analyze

2:     ▷ Subsystem: ICC Entry / Exit Point Extractor
3:     $permissions \leftarrow extractPermissions(apkFile)$
4:     $entryPoints \leftarrow extractIntentFilters(apkFile)$
5:     $exitPoints \leftarrow identifyExitPoints(apkFile)$    ▷ static analysis to determine the attributes of intents passing through the ICC exit points

6:     ▷ Subsystem: Dataflow Analyzer
7:     $timeout \leftarrow 300s$
8:     try then
9:         $iccExitLeaks \leftarrow identifyICCExitLeaks(apkFile, timeout, preciseConf)$    ▷ static analysis to identify ICC exit leaks with a high precise configuration
10:    catch TimeoutException
11:        $iccExitLeaks \leftarrow identifyICCExitLeaks(apkFile, timeout, lessPreciseConf)$    ▷ precise configuration failed, try a less precise configuration
12:    end try

13:    try then
14:        $iccEntryLeaks \leftarrow identifyICCEntryLeaks(apkFile, timeout, preciseConf)$    ▷ static analysis to identify ICC entry leaks with a high precise configuration
15:    catch TimeoutException
16:        $iccEntryLeaks \leftarrow identifyICCEntryLeaks(apkFile, timeout, lessPreciseConf)$    ▷ precise configuration failed, try a less precise configuration
17:    end try

18:    ▷ Subsystem: Data Module
19:    $appId \leftarrow saveToDatabase(permissions, entryPoints, exitPoints, iccExitLeaks, iccEntryLeaks)$    ▷ aggregate the data extracted by the Entry/Exit point extractor and Dataflow analyzer, and store in the DIALDroid db

20:    ▷ Subsystem: ICC Leak Calculator
21:    $calculateICCLeaks(appId)$    ▷ compute ICC leaks to/from this app
22: end System
```

Algorithm 3 Calculate sensitive ICC Channels to / from an app

```
1: Procedure CALCULATESENSITIVECHANNELS($appId$)    ▷ Computing sensitive ICC channels to / from the app identified by appId
2:     $appAsSrc \leftarrow getAppIntentsWithExitLeaks(appId)$
3:     $allLeakingIntents \leftarrow getAllIntentsWithExitLeaks()$
4:     $appIntentFilters \leftarrow getAppIntentFilters(appId)$
5:     $allIntentFilters \leftarrow getAllIntentFilters()$

6:     $sensitiveChannels \leftarrow array(,)$
7:     for all intent in appAsSrc do    ▷ computing sensitive channels originating from this app
8:         for all filter in allIntentFilters do
9:             if match(intent,filter) then
10:                $sensitiveChannels.append(intent.exit, filter.entry)$
11:            end if
12:        end for
13:    end for

14:    for all intent in allLeakingIntents do    ▷ computing sensitive channels ending at this app
15:        for all filter in appIntentFilters do
16:            if match(intent,filter) then
17:                $sensitiveChannels.append(intent.exit, filter.entry)$
18:            end if
19:        end for
20:    end for
           return $sensitiveChannels$
21: end Procedure
```

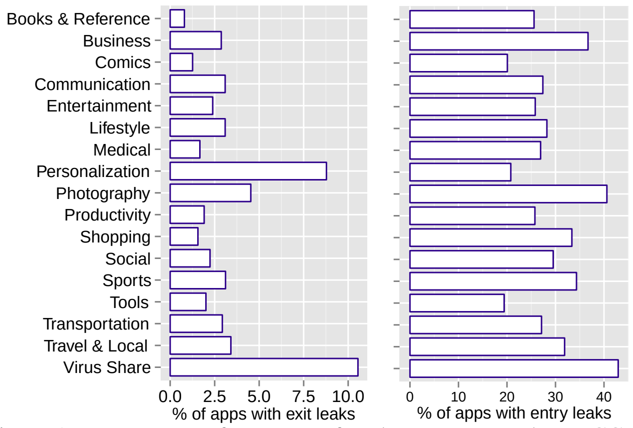

Figure 1: Percentages of apps out of each app category have ICC exit leaks (left) or ICC entry leaks (right) in Dataset III.

B. ADDITIONAL RESULTS

On average, an app in Dataset III accesses ≈ 7 sensitive APIs and invokes ≈ 13 ICC calls. Results of Kruskal-Wallis tests suggest that some categories of apps access a higher number of sensitive APIs ($\chi^2 = 5907.9, df = 16, p < 0.001$) or make a higher number of ICC calls ($\chi^2 = 3841.5, df = 16, p < 0.001$). Malware apps (in Virus Share) access a higher number of sensitive APIs or to invoke more ICC methods, as expected.

B.1 Intra-app ICC Comparisons

We compare DIALDroid with four leading single-app ICC tools (Amandroid, IccTA, DroidSafe, and COVERT). The comparison is conducted on 44 intra-app ICC test cases from the three benchmarks. Table 8 shows the accuracy results of the five tools against the 44 intra-app test cases in three benchmarks. DIALDroid has the highest precision (94.1%), the third highest recall (74.4%), and the highest F-measure (0.83) among the five tools in our experiment. IccTA's intra-app detection accuracy is comparable to DIALDroid. DroidSafe achieves the highest recall (100%) supporting its claim regarding the most precise Android life cycle modeling. However, DroidSafe appeared weak on intent resolution. It reports the highest number of false positives (36). Amandroid missed the most on the leaks against the Service and Provider related test cases, suggesting likely insufficient lifecycle modeling for those two types of components. COVERT had the poorest accuracy among the five tools, indicating inaccurate implicit intent matching as well as inadequate Android life cycle modeling.

We manually inspected the cases where DIALDroid failed. DIALDroid primarily failed to recognize two types of data leaks. One is via fake service calls, e.g., `bindService2`, `bindService3`, and `bindService4` that leak data within the same component using the `onServiceConnected` callback method. The other is via static fields, e.g., `ActivityCommunication1`. DIALDroid failed to recognize those leaks, due to the lack of intents carrying sensitive information. The two false positives (`startActivity6`, `startActivity7`) were due to the loss of field sensitivity. E.g., the sender stores sensitive information in a field with a key = `DroidBench` (e.g., `intent.putExtra("DroidBench",IMEI)`) but the receiver leaks `DroidBench2` (e.g., `Log.i(intent.getStringExtra("DroidBench2"))`. Due to the loss of field sensitivity, we performed a simple regular expression string search for the key (i.e., in this case `DroidBench`) in the ICC entry leak path. Since a search for 'DroidBench' will

Table 7: Statistics of our program analysis during the scalability evaluation of DIALDroid with 110,150 real-world apps.

Category	# Analyzed	High-precise config.	Less-precise config.	Time-out
Books & Reference	8,146	83.7%	13.8%	2.5%
Business	5,949	72.7%	18.0%	9.3%
Comics	2,057	80.5%	16.7%	2.8%
Communication	5,632	77.5%	13.1%	9.4%
Entertainment	8,189	77.4%	16.7%	5.9%
Lifestyle	7,368	76.6%	17.5%	5.9%
Medical	1,801	81.5%	13.6%	4.9%
Personalization	7,435	84.7%	14.1%	1.2%
Photography	8,041	79.5%	16.6%	3.9%
Productivity	7,582	82.6%	12.8%	4.6%
Shopping	6,336	77.1%	15.3%	7.6%
Social	6,870	71.4%	20.8%	7.8%
Sports	7,047	78.1%	16.2%	5.7%
Tools	8,105	85.0%	12.1%	2.9%
Transportation	6,323	81.9%	12.7%	5.4%
Travel & Local	3,344	73.8%	18.5%	7.5%
Virus Share	9,944	63.7%	26.5%	9.8%
Total*:	110,150	83.6%	10.7%	5.7%

**A few apps belong to multiple categories.*

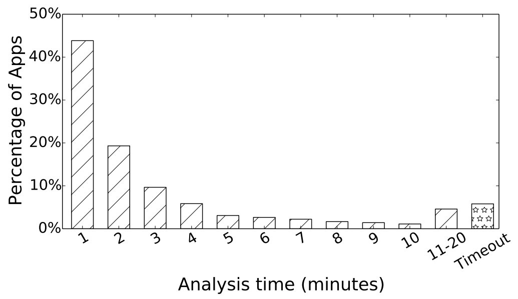

Figure 2: The distribution of DIALDroid's execution time for apps in Dataset III.

match a string containing 'DroidBench2', DIALDroid reports the above scenario as a leak.

We compare the execution time of five tools in Figure 3. The average is computed from three executions, for each data point. Only Amandroid's execution time was better than DIALDroid. All the tools, except Amandroid, are built on top of SOOT [34]. DIALDroid has the fastest execution among the four SOOT-based tools.

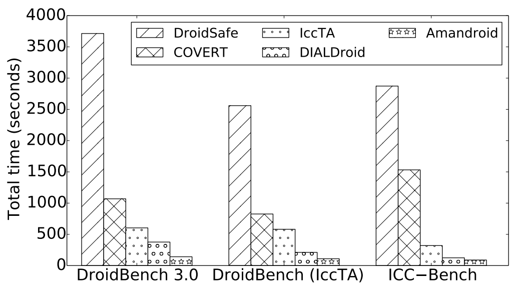

Figure 3: Comparisons of the averaged intra-app execution time on single-app benchmarks with four other state-of-the-art solutions.

Table 8: Intra-application ICC test results on DroidBench (develop branch), DroidBench (IccTA branch), and ICC-Bench. Multiple circles in one row means multiple intra-app leaks expected. An all-empty row: no leaks expected and none reported. †indicates the tool crashed on that test case.
☑= a correct warning, *= a false warning, ○= a missed leak

Test Case	# ICC Exit Leaks	# ICC Entry Leaks	Explicit ICC	COVERT	Amandroid	IccTA	DroidSafe	DIAL-Droid (Ours)
DroidBench 3.0								
ActivityCommunication1	1	2		○	○	☑	☑	○
ActivityCommunication2	1	2		○	☑*	☑*	☑*	○
ActivityCommunication3	1	5		○	☑	☑	☑*	☑
ActivityCommunication4	1	2		○	☑	☑*	☑*	☑
ActivityCommunication5	1	2	✓	○	☑	☑	☑*	☑
ActivityCommunication6	1	2		○	☑	○	☑*	○
ActivityCommunication7	1	2	✓	○	☑	☑	☑*	☑
ActivityCommunication8	1	2		○	☑*	☑*	☑*	○
BroadcastTaintAndLeak1	1	2		○	○	○	☑	☑
UnresolvableIntent1	1	2		○○	☑☑	☑☑	☑☑	☑☑
DroidBench (IccTA branch)								
startActivity1	1	2	✓	☑	☑	☑	☑*	☑
startActivity2	1	2	✓	○	☑	☑	☑*	☑
startActivity3	1	5	✓	○	☑	☑	☑*	☑
startActivity4	1	2			*	**		
startActivity5	1	2			**			
startActivity6	1	2	✓	*	*	*	**	*
startActivity7	1	2	✓	*	*	*	**	*
startActivityForResult1	1	2	✓	☑	☑	☑	☑*	☑
startActivityForResult2	1	2	✓	☑	○	☑	☑	☑
startActivityForResult3	1	3	✓	○	○	☑	☑*	○
startActivityForResult4	1	4	✓	○○	☑○	☑☑	☑☑**	☑○
startService1	1	2	✓	☑	☑	☑	☑*	☑
startService2	1	2	✓	☑	○	☑	☑*	☑
bindService1	1	2	✓	☑	○	☑	☑*	☑
bindService2	1	0	✓	☑	○	☑	☑	○
bindService3	0	0	✓	○	○	☑	☑	○
bindService4	1	2	✓	☑○	○○	☑☑	☑☑*	☑○
sendBroadcast1	1	1		☑	○	☑	☑***	☑
sendStickyBroadcast1	1	1		○	○	☑	☑*	☑
insert1	1	2		○	○	○†	☑***	☑
update1	1	2		○	○	○†	☑***	☑
delete1	1	2		○	○	○†	☑***	○
query1	1	1		○	○	○†	☑**	○
ICC-Bench								
implicit_action	1	1		○	☑	☑	☑	☑
implicit_category	1	1		○	☑	☑	☑	☑
implicit_data1	1	1		○	☑	☑	☑	☑
implicit_data2	1	1		○	☑	☑	☑	☑
implicit_mix1	1	2		○	☑	☑	☑	☑
implicit_mix2	1	1		○	☑	☑	☑	☑
implicit_Src_Sink	1	1		○	○	○	☑	☑
explicit1	1	1	✓	☑	☑	☑	☑	☑
explicit_Src_Sink	1	1	✓	☑	☑	☑	☑	☑
dynamicregister1	1	1		○	☑	☑	☑	☑
dynamicregister2	1	1		○	○	○	☑	☑
Sum, Precision, Recall, and F measure								
True positive (☑), higher is better				11	24	35	43	32
False positive (*), lower is better				2	7	7	37	2
False negative (○), lower is better				32	19	8	0	11
Precision, *p* = ☑/(☑+*)				**83.3%**	**77.4%**	**83.7%**	**53.7%**	**94.1%**
Recall, *r* = ☑/(☑+○)				**25.6%**	**55.8%**	**81.4%**	**100%**	**74.4%**
F-measure = *2pr/(p+r)*				**0.39**	**0.65**	**0.82**	**0.70**	**0.83**