Understanding Application Behaviours for Android Security: A Systematic Characterization

Haipeng Cai
Department of Computer Science
Virginia Tech
hcai@vt.edu

Barbara Ryder
Department of Computer Science
Virginia Tech
ryder@vt.edu

ABSTRACT

In contrast to most existing research on Android focusing on *specific* security issues, there is little broad understanding of Android application run-time characteristics and their security implications. To mitigate this gap, we present the *first dynamic* characterization study of Android applications that targets such a broad understanding for Android security. Through lightweight method-level profiling, we have collected 33GB traces of method calls and inter-component communication (ICC) from 114 popular Android applications on Google Play and 61 communicating pairs among them that enabled an extensive empirical investigation of the run-time behaviours of Android applications. Our study revealed that (1) the Android framework was the target of 88.3% of all calls during application executions, (2) callbacks accounted for merely 3% of the total method calls, (3) 75% of ICCs did not carry any data payloads with those doing so preferring bundles over URIs, (4) 85% of sensitive data sources and sinks targeted one or two top categories of information or operations which were also most likely to constitute data leaks. We discuss the security implications of our findings to secure development and effective security defense of modern Android applications.

1. INTRODUCTION

The Android platform and its user applications (commonly called *apps*) now dominate the mobile computing arena, including smartphones, tablets, and other consumer electronics [12, 43]. Android developers are increasingly creating apps that cover a growing range of functionalities. Meanwhile, accompanying the rapid growth of Android apps is a surge of security threats and attacks of various forms [12, 43]. In this context, it becomes crucial for both developers and end users to *understand* the particular software ecosystem of Android for effectively developing and securing Android apps.

While written in Java, Android apps have set themselves apart from traditional Java programs by how they are built and the environment in which they execute. Android apps are usually highly reliant on the Android SDK and other third-party libraries [23, 44]. In fact, many of the distinct characteristics of Android apps have led to unique challenges in developing sound and effective code-based security analyses, resulting in specialization and customization for Android of analyses originally designed for traditional object-oriented programs.

Specifically, the framework-based nature of Android apps requires substantial modeling of the Android runtime for static analyses [23, 30, 44] to achieve accuracy. Implicit invocation between components in Android apps through a mechanism called inter-component communication (ICC) requires special treatments (e.g., ICC resolution [37, 38]) for a soundy [29] whole-program analysis. In addition, the event-driven paradigm in Android programming accounts for many challenges in Android security analyses, such as determining application and component lifecycles [1, 30, 44] and callback control flows [1, 48].

Most current security solutions aimed at *specific* issues and threats [43], with merely a few works offering broader views of security characteristics of Android [9, 14, 15, 17, 32]. Recently, more studies on Android apps appeared [16, 17, 25, 34, 41] but targeted *static* characterizations by examining the code rather than run-time behaviours of the apps. Existing *dynamic* studies for Android address specific apps (e.g., [45]) or focus on malware only (e.g., [49]) rather than performing a general behavioural characterization. Thus, there remains very little understanding of Android apps from the perspective of run-time characteristics and behaviours that underlie their security, the focus of our study.

We randomly chose 114 of the latest Android apps from Google Play and 61 linked pairs among them, extensively exercised each app and app pair with automatically generated inputs, and gathered 33GB traces of ordinary function calls and ICCs. From these traces, which capture how Android apps are typically used, we calculated metrics relevant to security analysis challenges including the interaction between user code and libraries, distribution of components and ICCs, classification of callbacks, and categorization of security-sensitive data accesses. The metrics constitute the first dynamic characterization of Android apps that informs both a broad understanding of Android security and the development of future security defense solutions for Android.

The main contributions of this work include:

- A dynamic study of the layered structure and functionality distribution of Android apps, which sheds light on the security implications of their run-time construction. The study reveals that (1) Android apps are extremely framework-intensive, with 90% of all callers and callees being SDK methods and (2) 60–90% of all exercised components are *Activities*, which receive over 60% of all exercised lifecycle callbacks.
- An intensive investigation of ICC, the main communication mechanism in Android, which suggests optimization strategies for ICC-involved security analyses of Android apps. The investigation reveals that (1) 75% of ICCs do not carry any

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

data payloads and (2) ICCs that carry data payloads largely favor using bundles (20%) over URIs (5%) to do so.

- A detailed characterization of sensitive API calls during long Android app executions, which informs code-based analysis of sensitive-data accesses for improved cost-effectiveness. The characterization reveals that (1) over 92% of exercised API calls accessing sensitive data focus on network information and (2) nearly 85% of exercised API calls possibly leaking sensitive data are account setting or logging operations.
- An open-source dynamic study toolkit including various categorizations that can be used for future characterization studies and general understanding of Android applications.
- A benchmark suite of 45 unique, dynamically communicating app pairs (via ICC) that can be used for other Android studies and analyses, especially for *dynamic inter-app* analyses.

2. BACKGROUND

Android is now the most popular operating system (OS) running on smartphones and other types of mobile devices. To facilitate the development of user applications, the Android OS provides a rich set of application APIs as part of its sophisticated SDK framework, which implements comprehensive functionalities commonly used on various mobile devices. These APIs serve as the only interface for applications to access the device, and the framework-based paradigm allows for quick creation of user applications through extending and customizing SDK classes and interfaces. The Android framework communicates with applications and manages application executions via various callbacks, including *lifecycle methods* and *event handlers* [1].

Four types of components are defined in the Android framework, *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*, as the top-level abstraction of user interface, background service, response to broadcasts, and data storage, respectively, in user applications [11]. The SDK includes dedicated APIs for ICC via which components communicate by passing messages called *Intents*. ICCs can link components both within the same app (i.e., *internal* ICC) and across multiple apps (i.e., *external* ICC). Application components send and receive Intents by invoking ICC APIs either explicitly or implicitly. For an *explicit* ICC, the source component specifies to which target component the Intent is sent; for an *implicit* ICC, the component which will receive the Intent is determined by the framework at runtime.

Some information on mobile devices is security-sensitive, such as device ID, location data, and contacts [10, 11]. Taint analysis commonly identifies sensitive information leakage by detecting the existence of feasible program paths, called *taint flow*, between predefined taint *sources* and taint *sinks* [1, 8]. In Android, taint *sources* are the APIs through which apps access sensitive information (i.e., *sensitive* APIs). The Android SDK also provides APIs (inclusive of those for ICCs) through which apps can send their internal data to other apps either on the same device or on remote devices (e.g., sending data to network and writing to external storage). These APIs potentially constitute operations that are security-critical as they may lead to data leakage (i.e., *critical* APIs or taint *sinks*).

3. EXPERIMENTAL METHODOLOGY

We traced method calls and ICCs to understand the dynamic features of applications in Android. The resulting traces capture coarse-grained (method-level) control flows but not data flows. Nonetheless, such traces can reveal a broad scope of important dynamic characteristics regarding the typical behaviours and security-related traits of Android apps. Next, we elaborate on the design of our empirical study—benchmark apps, inputs used for the dynamic analysis, metrics calculated, and study procedure.

3.1 Benchmarks and Test Inputs

Our goal was to study at least 100 unique Android apps among which at least 50 pairs could potentially communicate through explicit or implicit inter-app ICCs. We first downloaded 3,000 free apps from Google Play that were ranked the most popular at the end of 2015. Then, we statically analyzed the ICCs of each app using the most precise current ICC analysis [37], and found potentially communicating app pairs by matching the ICCs across apps [3]. This process led to a pool of over one million such pairs linked via either explicit or implicit ICCs, or both.

Next, we randomly picked 20 different pairs and removed them from the pool, performed our instrumentation, and then ran the instrumented code on an Android emulator [19]. To ensure that we gathered useful traces and that our study reflected the use of the latest Android SDK features, we discarded pairs in which at least one app was built on a version of Android SDK lower than 5.0 (API 21) or failed to run on the emulator after the instrumentation. We repeated this random selection until reaching >=50 pairs. Eventually, we obtained 61 different app pairs that included 114 unique apps, which covered 25 of the total 27 Google Play app categories. Accordingly, 114 single-app traces and 61 app-pair (inter-app) traces formed the basis of our study. This benchmark suite is considerably larger than those used by existing *dynamic* analyses for Android [2, 7, 8, 26, 31, 33].

Since the objective of our study is to understand the general characteristics of latest Android apps from function call traces, it is essential that the traces are able to reveal typical Android application behaviours. Thus, the quality of inputs to be used for generating the traces is critical.

Previous dynamic studies of Android apps, using much smaller benchmark suites, mostly resorted to manual (expert) inputs [2, 7, 8, 26, 33], because the coverage of automatically generated Android inputs was regarded as too low. We chose to use automatically generated inputs for two reasons. First, manually manipulating various Android apps is expensive, subject to human bias and uneven expertise, and an unscalable strategy for dynamic analysis. Second, state-of-the-art automatic Android input generators can achieve practically as high code coverage as human experts [31] and are scalable. The latest, most comprehensive comparison of such generators showed that the Monkey [21] tool, part of the Android SDK, won over its peer approaches in terms of both (statement) coverage alone and overall evaluation [4].[1]Therefore, we utilized the Monkey tool shipped with the Android SDK 6.0 (the latest release). Although Monkey does not generate many system events directly, it triggers those events indirectly through UI events. By overall average, our test inputs achieved statement coverage of 54.5%, close to the highest coverage reported in the literature (60%) which was attained with experienced-human inputs on a much smaller benchmark suite [4].

3.2 Metrics

We express the dynamic characteristics of Android apps via three high-level categories of metrics, each consisting of several supporting measures, as defined as follows.

General metrics—concerning the composition and distribution of app executions with respect to their usage of different layers of functionalities: user code (*UserCode*), third-party libraries

[1]DynoDroid [31] was developed for outdated Android platforms; we confirmed that it did not work for the latest apps in our study.

(*3rdLib*), and the SDK (*SDK*). Specific measures include (1) the distribution of function call targets over these layers, (2) the interaction among the layers (i.e., calling relations and frequency), and (3) the extent and categorization of callback usage.

ICC metrics—concerning the primary Android mechanism for inter-component interaction within single apps and across multiple apps. ICC has been a major security attack surface in Android [3, 32, 39] as well as a feature of Android application programming that sets it apart from ordinary Java programming. Specifically, we measure (1) the distribution of the four types of components (see Section 2) in Android app executions, (2) the categorization of run-time ICCs with respect to their scope (internal/external) and linkage (implicit/explicit), and (3) the data payloads carried by ICC Intents with respect to different ICC categories.

Security metrics—concerning the production, consumption, and potential leakage of sensitive data in Android app executions. We measure (1) the extent of use of the producers (i.e., sources) and consumers (i.e., sinks), (2) the categories of information accessed by executed sources and operations performed by executed sinks, and (3) the occurrence of sensitive data leaks via control-flow paths (call sequences) connecting sources to sinks.

We examine two views of each measure: *instance* and *unique*, except for call frequency and ICC categorization reported only for all instances of calls. The instance view is associated with the full execution traces we analyzed; the unique view is associated with the source code covered by the test inputs. Therefore, the instance view captures the run-time behaviour of the apps with call frequencies while the unique view considers specific callsites in the source code that are executed at least once. These two views are complementary to each other, together conveying the dynamic characteristics and behaviours of Android applications.

3.3 Procedure

To collect the operational profiles of the benchmark apps, we first ran our tool to instrument each app for monitoring ICC Intents and all method calls. Next, we ran each instrumented individual app separately and then each app pair, gathering 114 single-app traces and 61 inter-app traces (when multiple target apps in inter-app ICCs are available, one was randomly chosen). All of our experiments were performed on a Google Nexus One emulator with the latest Android SDK (6.0/API level 23), 2G RAM, and 1G SD storage, running on a Ubuntu 15.04 host with 8G memory and 2.6GHz processors. To avoid possible side effects of inconsistent emulator settings, we started the installation and execution of each app or app pair in a fresh clean environment of the emulator (with respect to built-in apps, user data, and system settings, etc.).

For each individual app, Monkey inputs were provided for up to one hour of execution. For each app pair, the two apps ran concurrently, taking Monkey inputs alternately for an hour. These one-hour-long runs have allowed Monkey to achieve the highest coverage among existing automatic input generators for Android as substantiated by the latest comparative study [4]. In both the single- and inter-app settings, there were events generated by Monkey that led to system halts of the host machine before the timeout was reached; we discarded the corresponding traces and repeated the same run again until we obtained an hour long trace. Similarly, we ignored app crashes and had Monkey restart the app.

To reduce the impact of possible non-determinism in the benchmarks, we repeated each experiment three times and took the average of these repetitions. We checked the repeated traces for each app and app pair, and found only very small deviations among them. Thus, we used the mean over the repetitions for each metric as the final metric value per app and per app pair.

4. STUDY TOOLKIT

Figure 1 depicts the workflow of our toolkit DROIDFAX, including its three phases as well as inputs and outputs.

Pre-processing. After obtaining the benchmark app pairs as described in Section 3.1, the static code analysis instruments the Android (Dalvik) bytecode of each app for method call profiling and ICC Intent tracing. This first phase also produces relevant static information for each app using class hierarchy analysis (CHA), including the component type each class belongs to (i.e., the top component class it inherits) and callback interface each method implements in the app. This information is used for computing trace statistics in the third phase. Both the instrumentation and CHA are implemented on top of Soot [28].

Profiling. The second phase runs the instrumented code of each individual app and app pair to produce the single- and inter-app traces in the respective settings. DROIDFAX records method calls and ICC Intents using the Android logging utility and collects the traces using the `logcat` tool [20] that is part of the SDK.

Characterization. The third phase analyzes the traces by first building a *dynamic call graph*. Each node of the graph is the signature of a method (executed as a caller or callee), and each edge represents a dynamic call which is annotated with the frequency (i.e., number of instances) of that call. Also, for each ICC, the graph has an edge going from the sending API (e.g., `startActivity`) to the receiving API (e.g., `getIntent`) of that ICC. This phase computes various metrics using the call graph and the static information computed in the first phase. From single-app traces, DROIDFAX calculates all three categories of metrics. From inter-app traces only the ICC metrics are computed. In order to categorize event handlers, DROIDFAX utilizes a predefined categorization of callback interfaces, which we manually produced from the uncategorized list used by FlowDroid [1]. We did the categorization based on our understanding of each interface gained from the official Android SDK documentation. (Categorization of lifecycle callbacks was done using CHA.) Another input to this phase is the lists of sources and sinks that we defined by manually improving the training set of SuSi [40] hence producing a more precise categorization. To facilitate reproduction and reuse, the entire implementation of DROIDFAX is open source. Also publicly available are our study results, the categorization of event handlers we created, and the improved source and sink categorization.[2]

5. RESEARCH QUESTIONS

With the three classes of metrics described above, our empirical study seeks answers to the following research questions.

RQ1: How heavily are the SDK and other libraries used by Android apps? This question addresses the construction of Android apps in terms of their use of different layers of code and the interaction among them. Answering this question offers empirical evidence on the *extent* of the framework-intensive nature of Android apps—previous works only *suggested* the *existence* of that nature through static analysis [23, 30]. RQ1 is answered using the first two measures of the general metrics.

RQ2: How intensively are callbacks invoked in Android apps? It is well known that callbacks, including lifecycle methods and event handlers, are widely *defined or registered* in Android app code [1, 44, 48]. This research question addresses their *actual usage* in Android app executions, that is, the frequency of callback invocation and the distribution of different types of callbacks. RQ2 is answered using the third measure of the general metrics.

[2]Links to all these will be included in the camera-ready version.

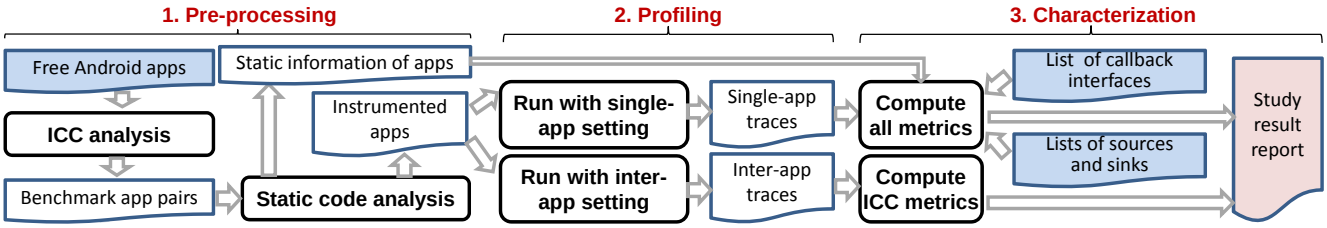


Figure 1: The three phases in the workflow of our study toolkit DROIDFAX, including its inputs and outputs.

RQ3: How do Android app components communicate using the ICC mechanism? Much prior research has targeted Android security concerning ICCs [3, 32, 37, 39], yet it remains unclear how often ICCs occur relative to regular function calls during the executions of Android apps, how different types of ICCs are used distinctly, and whether all ICCs constitute security threats. The answer to each of these questions is subsumed by RQ3, and is investigated using the ICC metrics.

RQ4: How is sensitive information accessed in Android apps? Addressing the secure usage of sensitive information has been the focus of various previous works, including taint analysis [1, 44], privilege escalation defense [2, 30], and data leakage detection [3, 50]. However, how often that usage is exercised or which kinds of sensitive information are mostly accessed has not been studied. RQ4 explores these unanswered questions, and also addresses how risky the accesses can be according to the reachability of API calls from taint sources to taint sinks, all using the security metrics.

6. EMPIRICAL RESULTS

This section presents the results of our study, reporting the three categories of metrics with respect to relevant research questions. For call frequencies, we reported the number of instances of each executed callsite throughout all single-app traces using scatterplots. For each of the other metrics, which was consistently expressed as a percentage, we first calculated the percentage (from the three repetitions as described above) for each app (or each app pair) separately. Then we reported either the distribution of all these percentages using boxplots or their summary statistics (mean and standard deviation) using tables. In each boxplot, the lower whisker, the lower and upper boundaries of the box, and the upper whisker indicate the minimum, the first and third quartiles, and the maximum, respectively; The horizontal line in the box indicates the median and the diamond indicates the mean. We have set the whiskers to extend to the data extremes (so no outliers are shown).

For each category of metrics, we first present the results in detail and then summarize and discuss the most important observations from an *average-case* perspective. We also offer insights into the implications of our empirical findings and demonstrate how our results can be used in future secure development and security defense of Android apps.

6.1 General Characteristics of Android Apps

To gain a general understanding of Android app behaviours, we investigated the structure of their execution in terms of three layers of functionality (i.e., *UserCode*, *SDK*, and *3rdLib*), the interaction among these layers, and the usage of callbacks.

6.1.1 Composition of Code and Execution

The composition of the method call trace of each Android app is characterized in terms of the percentages of executed callsites (i.e., in the *unique* view) and of call instances (i.e., in the *instance* view) accessing user code, third-party libraries, and the Android SDK. Figure 2 shows the distribution of these three layers in the single-app traces of the 114 benchmark apps, with each group of boxplots depicting both class and method granularity.

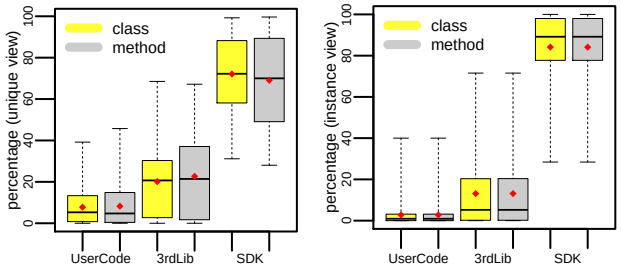


Figure 2: Percentage distribution (y axis) of the three code layers (x axis) over all executed callsites (left) and all call instances (right).

The *unique view* reveals that consistently all the subject apps heavily rely on library functionalities, especially the SDK, in performing their tasks. On average, at both class and method levels, only about 10% of all executed methods and classes were defined in user code, 25% to 30% were in various non-SDK libraries, and the rest were from the Android framework. Clearly, the results show that the SDK dominated these apps, and suggest that on average an Android app tends to be quite dependent on SDK functionality. At the coarser level of class, different Android apps exhibited even stronger uniformity in their execution composition, as implied by the relatively smaller interquartile ranges at that level of granularity.

Counting all call instances, the *instance view* further confirms the framework-dependent nature of Android apps. This view shows that SDK code was executed the most frequently among the three layers, suggesting that run-time behaviours of the SDK dominate. The observation that almost 90% (on average) of all calls were to the SDK code in most apps corroborates that Android apps are highly framework-intensive. In contrast, Android apps tend to execute their user code relatively occasionally—in fact, only 25% of the apps had over 5% of these calls to user code targets and none had over 40%.

6.1.2 Inter-layer Code Interaction

Figure 3 scatter-plots the frequency of each executed callsite per app. The data points are categorized by the calling relationships, denoted in the format of *caller layer→callee layer*, among the three code layers. Each plot shows the call-frequency ranking for one of the nine categories of inter-layer interaction. The breadth of each plot indicates the total number of executed callsites in the corresponding interaction category, while the height suggests the range of frequencies of all those calls. To better distinguish categories, logarithmic scales are used on both axes. For instance, the rightmost plot represents the frequency ranking for calls between SDK methods (*SDK->SDK*), covering over 200K callsites with the highest individual call frequency of about 800K.

Consistent with the results in Figure 2, these plots confirm that (1) many more SDK and third-party library APIs were called than user methods and (2) the total number of unique SDK callees

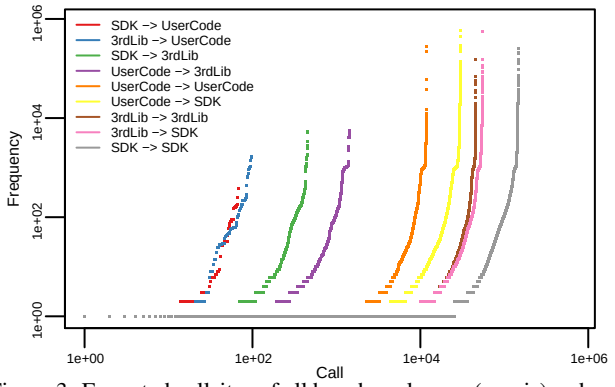


Figure 3: Executed callsites of all benchmark apps (x axis) ordered non-descendingly by their frequencies (y axis). The legend labels listed from the top to the bottom correspond to the scatter plots ordered from the left to the right by their maximal x-coordinates.

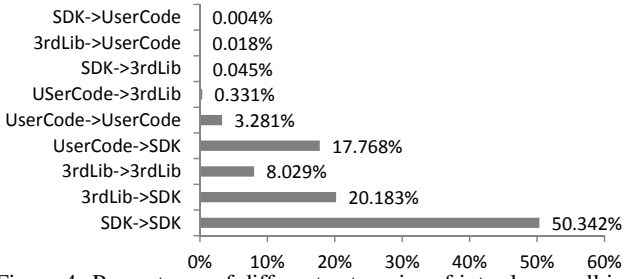


Figure 4: Percentages of different categories of inter-layer call instances over all benchmark app executions.

dominated all callees. The plots reveal that categories having larger numbers of callsites mostly had larger frequency maxima as well. The most frequently exercised calls were *from* SDK, which also received the calls of the highest frequency (from user code).

Figure 4 shows the percentage of call instances in each inter-layer interaction category over the total call instances in all benchmark app executions. Noticeably, the majority (61.7%) of call instances over all apps happened *within* the same layer—dominated by the SDK layer (50.3%)—rather than across layers. The busiest callees were SDK APIs (88.3% in total), invoked mostly from SDK (50.3%) followed by third-party libraries (20.2%). User functions were called very rarely by any callers (no more than 4%), reconfirming our previous observation from Figure 2. The results reveal that the vast majority of calls to third-party library functions were from the same layer of code. Calls to *UserCode* from *SDK* or *3rdLib* were callbacks from the framework and other libraries to application methods. The much smaller numbers and lower frequencies of such calls show that user-code callbacks were executed comparatively rarely.

In summary, results on inter-layer code interaction further confirm the highly framework-intensive nature of Android apps, indicating that the Android framework tends to do the majority of application tasks while user code often just relays computations to the SDK and various other libraries.

6.1.3 *Usage of Callbacks*

We examined the extent of callback usage (over the three code layers) in the benchmark apps through the distribution of percentages of callback method invocations. As shown by Figure 5, on average no more than 3% of all executed callsites (in the *unique view*) targeted either lifecycle callbacks or event handlers. Overall, there were more unique lifecycle callbacks than distinct event handlers used by these apps. In a few apps, no more

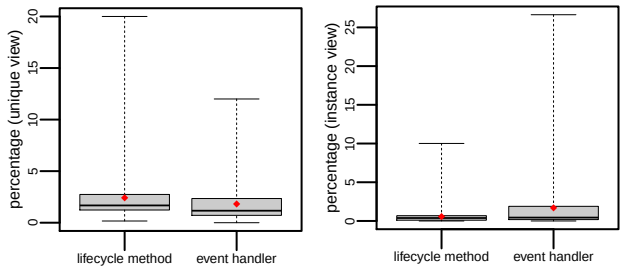


Figure 5: Percentage distribution (y axis) of callbacks (x axis) over all executed callsites (left) and all call instances (right).

Table 1: Lifecycle methods breakdown over all categories

Category	Unique view	Instance view
Activity	59.63% (38.93%)	60.98% (40.78%)
Application	27.03% (33.22%)	29.24% (36.96%)
BroadcastReceiver	1.25% (3.96%)	0.29% (1.70%)
ContentProvider	3.12% (10.96%)	0.88% (6.52%)
Service	0.50% (2.82%)	0.13% (0.92%)

than 20% of all executed callsites were for lifecycle management while at most 12% were for handling other kinds of events.

The *instance view* indicates that callbacks were not invoked very frequently. The average percentage of either type of callback invocations was under 2%, indicating that callbacks, while prevalently defined and registered in Android apps [1, 48], tend to be only lightly used at runtime. This observation is consistent with the call frequency ranking of Figure 3, where we have seen that relatively small numbers and low frequencies of calls invoking user code from the SDK or other libraries. Comparing the two types of callbacks in the *instance view* reveals that (1) event handlers were called much more frequently than lifecycle callbacks and (2) there were apps executing event handlers substantially (up to 27% of total call instances) yet none of the 114 apps had more than 10% of calls targeting lifecycle methods.

To look further into the callback usage, we categorized lifecycle callbacks by their enclosing classes with respect to the four types of application components (see Section 2) and the *Application* type corresponding to the `android.app.Application` class defined in the SDK. The percentage of each category over the total lifecycle callbacks is listed in Table 1, including the means and corresponding standard deviations (in parentheses) of such percentages over all benchmark apps.

As shown, *Activity* lifecycle methods both dominated the targets of all executed callsites and were invoked most frequently. The second most handled lifecycle events were associated with the application as a whole. Events handled by the other three types of components were marginal, totaling less than 5% on average in terms of the number of executed callsites and merely 1.5% when counting all call instances of lifecycle methods. The large variances of these means suggest considerable differences in this categorization across the set of apps. Nevertheless, the majority of lifecycle method callsites and call instances were dealing with various *Activities*, possibly due to the fact that Android apps usually have abundant user interfaces (UI) and rely on frequent user interaction. This observation justifies focusing on *selected* callbacks in modeling lifecycles of an Android app as a whole, such as considering *Activity* only when analyzing static control flows for lifecycle callbacks [48], to reduce analysis complexity and/or to achieve better performance.

Exploring the data further, Table 2 presents a two-level breakdown for event handlers according to our manual

Table 2: Event handlers breakdown over significant categories

Category		Unique view	Instance view
UI event	App bar	14.71% (29.94%)	16.50% (35.93%)
	Dialog	2.65% (6.28%)	0.25% (1.16%)
	View	25.69% (35.96%)	24.74% (39.81%)
	Widget	1.41% (5.46%)	0.22% (1.76%)
System event	App mgmt.	43.92% (42.21%)	47.30% (47.58%)
	Hardware mgmt.	0.26% (1.61%)	0.04% (0.52%)
	Media control	0.43% (2.55%)	0.03% (0.19%)

categorization of those callbacks (see Section 4). If at least one of the benchmark apps had over 1% of all executed callsites and all call instances falling in a (second-level) category, we regarded that category as *significant*. We only report significant categories.

Overall, the total percentages of callbacks handling system events and such percentages of callbacks handling UI events are close in both views. The substantial variances of the means imply the existence of apps at two extremes—those predominantly having invocations of event handler callbacks responding to system events, and those predominantly having callbacks triggered by UI events. A more detailed look reveals that the majority of UI event handlers dealt with two particular kinds of user interfaces, *View* and *App bar*, while user events on *Dialog* or *Widget* were much less frequent. On average, most system event handlers responded to events that serve application management (*mgmt.*), with a few others dealing with hardware management and media control. Given these results, Android app analyses of event handlers [1, 44, 48] could be customized or optimized for better cost-effectiveness while remaining soundy [29] by prioritizing analysis of those in the most commonly used categories.

6.1.4 Summary and Discussion

The general metrics show that at runtime Android apps (1) depend heavily on the SDK—over 70% of unique methods executed are defined in the SDK and (2) are highly framework-intensive—almost 90% of all method instances are those of SDK methods, and the largest numbers (over 10K) of calls with the highest frequencies (over 100K) targeted methods in the SDK. Thus, *deeper understanding of the SDK and its interface is essential* for secure Android app development. Meanwhile, *the security of the Android framework itself should be well addressed* in securing the Android software ecosystem as a whole. The overwhelming dominance of SDK in app executions also implies *lesser impact of UserCode obfuscation* than expected, and uncovers potential *benefits of SDK optimizations.*

In addition, *Activity* as the dominating component type (accounting for 60–90% of all component instances) is also the predominant (about 60%) target of lifecycle method calls, which indicates that Android apps are generally rich in user interfaces. Therefore, Android app analyses should pay *sufficient attention to application features that are relevant to UI elements* (e.g., UI-induced data and control flows). Since invocations of various callbacks account for only small percentages (less than 5%) of all method calls, it may be practical and rewarding to *fully track callback data/control flows for fine-grained dynamic security analyses*. Finally, giving priority to the very few top categories of lifecycle methods and event handlers would render lifecycle modeling, taint analysis, and callback control flow analysis *more cost-effective (e.g., sacrificing some safety for higher scalability)*.

6.2 ICC Characterization

ICCs constitute the primary communication channel between the four types of components in Android. We first look at component distribution in the app executions before examining the interaction between them through ICCs. We then characterize whether data payloads are carried (i.e., *data carriage*) in the ICCs. We report the measures based on ICC Intents with respect to single- and inter-app traces separately and compare findings in these two communication settings for a more detailed ICC characterization. We calculated the percentage of ICC calls executed over all ICC callsites in each app and then averaged those percentages over all apps in our study obtaining a mean of 56.67% (standard deviation of 21.39%).

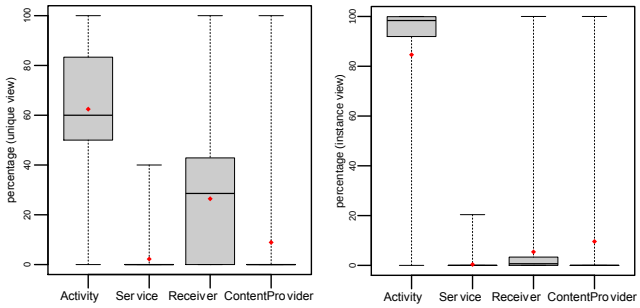


Figure 6: Distribution (y axis) of all executed callsites (left) and total call instances (right) over the four component types (x axis).

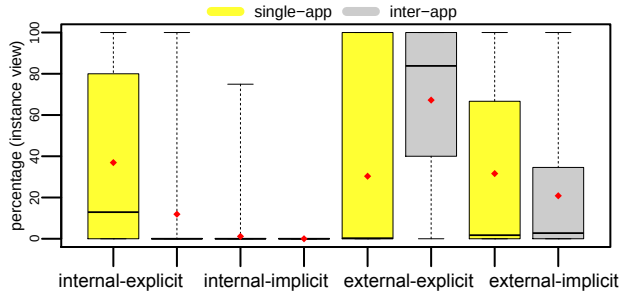


Figure 7: Percentage distribution (y axis) of all ICCs over four categories (x axis) in single-app versus inter-app executions.

6.2.1 Component Distribution

Figure 6 shows the distribution of executed callsites and call instances over different component types. In the *unique view*, despite the existence of (four) outlier apps which had *ContentProviders* dominate all their invoked components, by far the majority of our benchmark apps used *Activities* the most (over 60% on average) among all components executed. *Receivers* were used substantially too (about 30% on average), consistent with the previous observation that these apps had significant percentages of callbacks handling system events over all invoked callbacks (as seen in Table 2). In terms of run-time behaviour, the *instance view* further confirmed that *Activity* was the most often executed component type (over 90% on average).

6.2.2 ICC Categorization

Having an understanding of the usage of different types of components, we now break down all exercised ICCs (which link components), as shown in Figure 7, over four possible categories (on the x axis) in single- and inter-app traces separately. Each data point in the boxplots represents the percentage of ICCs in a particular category over all the ICCs for one app or app pair.

In the single-app setting, the results show similar mean percentages of internal-explicit, external-explicit, and external-implicit ICCs. By contrast, internal-implicit ICCs were rarely executed. Internal-explicit ICCs had the largest mean and median suggesting the dominance of that category in most apps, despite the apparent non-uniformity in the overall ICC distribution

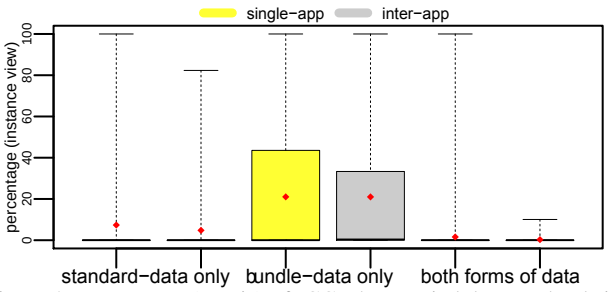


Figure 8: Percentage (y axis) of ICCs that carried data payloads in different forms (x axis) over all ICCs exercised.

among half of the apps (as seen from the generally small medians). Note that there were still considerable percentages of external ICCs in single-app traces because system and built-in apps (e.g., photo viewer, camera, maps, etc.) often communicated with our benchmark apps (usually via implicit ICCs).

In the inter-app setting, external-explicit and external-implicit ICCs combined (over 80% on average) dominated the internal ones (around 10%) in most app pairs. Dominating internal ICCs were seen only in a few outlier cases, and the majority of those ICCs were explicit. On average, external-explicit ICCs accounted for much higher percentages (70%) than external-implicit ones (20%). We inspected the traces and found that some implicit ICCs did not succeed because there were no receiver apps available on the device to handle the requests sent. In such cases, the execution was interrupted and the partial trace was ignored.

Our study reveals that less than 0.2% of method invocations were API calls for either sending or receiving ICCs. This marginal percentage implies that the overwhelming majority of calls were between methods *within* individual components. With respect to the total call instances, components communicated with other components only occasionally (e.g., when an intra-component computation completed and results were ready to deliver). About 47% of all ICCs were between two *Activity* components; among the other ICCs, more than 55% were initiated also by *Activities*. These observations are expected given the dominance of *Activity* among all component types (see Figure 6); this implies that the predominant use of ICCs in Android apps was for the communication between various user interfaces.

6.2.3 Data Carriage

Part of the reason for ICCs to become a major security attack surface is that they can carry, hence possibly leak, sensitive and/or private data. Thus, we investigated the ICC data carriage of single-app and app-pair executions. There are two ways in Android in which ICCs can carry data in an Intent: via the `data` field of the ICC Intent object, specified only through a URI, and via the `extras` field of that object (i.e., a bundle of any additional data accessible as a key-value structure). We refer to these two forms as *standard* and *bundle* data, respectively.

Figure 8 shows that there were only small differences in the percentage distribution between the two communication (i.e., single- and inter-app) settings, indicating that the setting did not much affect these observations. On average no more than 5% of ICCs transferred standard data only, while bundle-only ICCs were over 20% on average of the same total. However, very few ICCs carried both standard and bundle data at the same time. Regardless of the existence of outlier apps that passed either or both forms of data in almost all their exercised ICCs, the general observations are that (1) the ICCs that carried data account for a lesser proportion (25%) of the total and (2) bundles were favored (15% more) over URIs. An immediate implication of this observation to data-leak detection is that checking only the `data` field of Intents is inadequate as it misses the majority of potential data leaks. Instead, security analyses of ICC-based data leaks should carefully examine the bundles contained in ICC Intents [26].

Looking further at these data-carrying ICCs (see Appendix A for details), we also found that Intents containing standard data payloads (either standard-data only or both forms of data) were predominantly external-implicit ones, despite the setting.

6.2.4 Summary and Discussion

Our ICC categorization (Figure 7) reveals that components of the same apps communicate rarely through implicit ICCs (less than 1% on average) as opposed to through explicit ICCs (almost 40% and over 10% in the single- and inter-app settings, respectively); components across apps (i.e., in the inter-app setting) use implicit ICCs (about 20%) also much less often than using explicit ICCs (over 70%). This dominance of explicit ICCs over implicit ones suggests that *conservatively linking components via implicit ICCs (i.e., through the* `action`, `category`, `data` *tests [39, 44]) may lead to large imprecision* in static analyses of ICC-induced information flows. Our ICC data-carriage characterization (Figure 8) reveals that most (75%) ICCs do not carry any data and those carrying data tend to do so preferably via bundles (over 20%) instead of URIs (below 5%). Thus, security analyses including ICCs may benefit from *prioritizing examination of ICCs carrying data, especially those using bundles, to obtain more effective results* within a time budget. The preference of data-carrying ICCs for bundles also calls for *deeper analysis of the* `extras` *fields* in Intents. As almost all ICCs carrying data via URIs are implicit and *across* apps, *the* `data` *field of Intents needs attention* in inter-app data-leak detection.

The characteristics of ICCs in single-app executions are different from those in inter-app settings: (1) percentages of all internal ICCs are much more substantial in the single-app setting (nearly 40%) than in the inter-app setting (about 10%), whereas the latter saw much (40%) larger percentages of all external ICCs than the former; (2) percentages of all explicit ICCs are similar for both settings, whereas percentages of all implicit ICCs are noticeably (about 10%) higher in the single-app setting; (3) percentages of implicit data-carrying ICCs, either internal or external, are similar between the two settings, whereas percentages of explicit data-carrying ICCs are noticeably different. The single-app setting saw 35% more internal-explicit ICCs carrying data payloads than the inter-app setting, while the inter-app setting saw almost 50% more external-explicit ICCs carrying data payloads than the single-app setting. Due to these differences, an ICC-involved *single-app analysis can produce results considerably different from those given by an inter-app analysis* that deals with ICCs. There has been no concrete assessment of the effects of these differences. Nevertheless, an *accurate ICC analysis should consider its communication context* in terms of specific, potential communicating peer apps.

6.3 Security-Sensitive Data Accesses

Much of Android security analysis has been concerned with inappropriate accesses to security-sensitive data, including the abuse of sensitive data by an app and risky escape of the data out of an app. To understand the security implications of those accesses, we investigated in our benchmarks (1) the usage of sensitive and critical APIs, (2) the categories of sensitive data the APIs accessed and categories of critical operations the APIs performed, and (3) the possibility of data leaks at runtime.

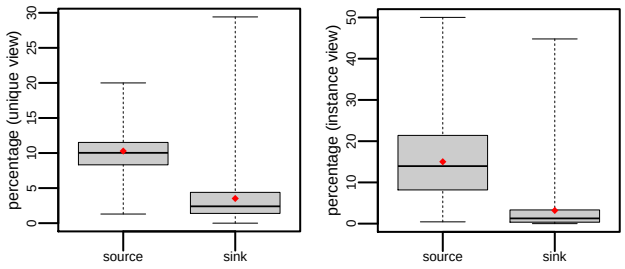


Figure 9: Percentage distribution (y axis) of sources and sinks (x axis) over all executed callsites (left) and all call instances (right).

Table 3: Source breakdown over significant categories

Category	Unique view	Instance view
ACCOUNT_INFO	0.03% (0.24%)	0.08% (0.90%)
CALENDAR_INFO	0.77% (2.58%)	0.69% (2.46%)
LOCATION_INFO	0.19% (1.10%)	0.10% (0.90%)
NETWORK_INFO	92.75% (22.01%)	93.70% (22.14%)
SYSTEM_SETTINGS	1.05% (2.47%)	0.24% (1.11%)

6.3.1 Usage of Sources and Sinks

Since sensitive data in Android is accessed via invocations of sensitive and critical SDK APIs, understandings the production and consumption of sensitive data requires examining the frequency of API calls that are data sources or sinks as a percentage of all method calls, as shown in Figure 9.

Overall, the benchmark apps tended to retrieve sensitive information often, containing on average one sensitive API call out of every ten unique method invocations during their executions. Half of the apps had an even larger proportion (up to 20%) of sensitive API calls. All the apps invoked such calls (at least 2%) to retrieve sensitive information, but there were apps that never invoked sinks which may consume and leak sensitive data. According to the unique view, sources were exercised much more extensively than were sinks in most individual apps.

The instance view shows that sources were invoked also much more frequently than sinks. The majority of the apps had total source API calls account for about 15% of total call instances, with the highest up to 50%. In comparison, 75% of the apps saw less than 5% sink calls among all their method calls, up to 45% in some outlier apps. In short, the apps had considerably intensive accesses to sensitive information, yet did not perform potentially data-leaking operations as often.

The exercised sources and sinks were run-time projections of the predefined lists [40] of (17,920 unique) sources and (7,229 unique) sinks used, respectively. Thus, the percentages of source and sink calls we reported are directly influenced by the sizes of these lists. The much longer list of predefined sources compared to the shorter list of predefined sinks partially explains the substantially higher ratios of source calls than sink calls.

6.3.2 Categorization of Sensitive Data Accesses

One way to further examine how Android apps use sensitive information is to look into the information itself accessed by the apps and operations that may leak such information. To that end, we categorized the source and sink API calls according to the kinds of data retrieved by the sources and operations performed by the sinks. Knowing which kinds of information Android apps tend to access most and which types of critical operations are most performed can inform end users about the potential risks of leaking security-sensitive data when using the apps, as well as help security analysts make right choices and/or configurations of security-inspection tools when using the tools for app vetting.

Table 4: Sink breakdown over significant categories

Category	Unique view	Instance view
ACCOUNT_SETTINGS	65.52% (32.04%)	66.67% (35.14%)
FILE	0.39% (1.44%)	0.31% (2.23%)
LOG	18.71% (25.12%)	15.95% (27.37%)
NETWORK	2.19% (4.93%)	0.63% (2.73%)
SMS_MMS	0.47% (2.29%)	0.28% (2.48%)
SYSTEM_SETTINGS	4.24% (6.65%)	7.68% (16.90%)

For each category Table 3 lists the percentage of unique sources and source instances accessing sensitive information in that category over corresponding totals, in the same format as Table 1. Categories having maximal percentage below 1% are omitted. There were only five categories of sensitive information noticeably accessed by the benchmark apps. Network information was dominant in both views. Such information was previously noted as widely accessed in Android apps [17], yet the overwhelming dominance (92–93% on average) of this category has not been reported. System settings, calender, and location related data were also among the most common categories [10, 14, 17].

A similar breakdown of sinks over six significant categories is summarized in Table 4. Interestingly, the dominant category was associated with account settings, suggesting that the Android apps deal with account management intensively relative to other kinds of critical operations. Applying *possibly* sensitive data in managing accounts does not seem to constitute a data-leak risk, yet such risks can occur when a user shares account settings across apps (e.g., user age and location data used in the settings for an account on one app may be disclosed to another app where the user logs in to the same account). The second most prevalent potential consumer of sensitive data was logging operations, which can disclose such data via external storage. Similarly to account-setting operations, API calls for system settings can lead to data leakage as well. Lastly, network and message-service operations are capable of leaking data through network connections to remote apps and devices. In fact, these categories of sinks were previously recognized as the major means of leaking data out of Android apps or the mobile device [8, 10].

In all, despite the large variances of the means, our results reveal that security-sensitive accesses in Android apps are not targeted *broadly* in terms of the categories of information accessed and operations executed. Instead, only very few kinds of sensitive data and critical operations, which are indeed highly relevant to common mobile device functionalities, are most frequently involved. This finding suggests that one way of optimizing information-flow-analysis-based security defense solutions (e.g., [1, 44]) could be to narrow down the search space of vulnerable flows to those involving data and operations of the predominating categories. The analysis performance might be greatly boosted without increasing false negatives substantially by ignoring a marginal portion of vulnerable flows.

Alternatively, with the knowledge from a dynamic analysis, the long lists of predefined sources and sinks used by static analyses could be pruned to focus on the ones used most often. For example, based on our results, considering just one or two top categories of sources and sinks would allow static taint analyses to capture taint flows between over 85% of all sources and sinks, providing a slightly unsafe but rapid solution—both previous studies and our experience suggest that cutting the lists significantly may lead to substantial analysis performance gains [1, 44].

6.3.3 Occurrence of Sensitive Data Leaks

Sensitive operations (source API calls) are insecure only if the sensitive information they retrieve flows to some sinks. Critical

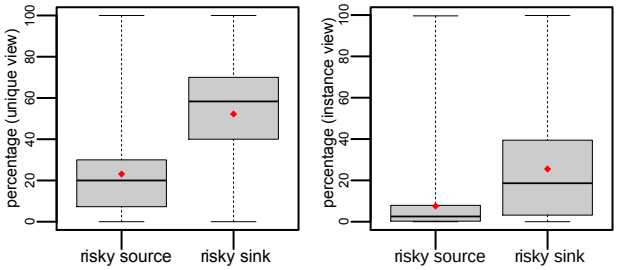


Figure 10: Percentage distribution (y axis) of *risky* sources and *risky* sinks over all invoked sources and sinks, respectively.

operations (sink API calls) are insecure only if the information they access is indeed private and/or security sensitive. Thus, the occurrence of sensitive data leaks is contingent on the existence of feasible information flow paths from sources to sinks. In fact, at the core of taint analysis is a reachability analysis discovering such flow paths. Traditionally, taint analysis examines sensitive information leaks through data flow analysis at statement [1, 44] or finer-grained levels [8]. In our study, we approximated feasible taint flows at method level by traversing the dynamic call graph of DROIDFAX (see Section 4): we considered a sink *reachable* from a source if their least common ancestor (on the graph) called the sink after calling the source during app execution. We identified sources reaching at least one sink (i.e. *risky sources*) and sinks reachable from at least one source (i.e., *risky sinks*) in the traces.

Figure 10 depicts the distribution of percentages of risky sources (risky sinks) over the total exercised sources (sinks) for each individual app. The unique view shows that on average 20% of exercised sources could leak the sensitive data they retrieved and 50% of exercised sinks could access (hence leak) sensitive data. Given our conservative approximation, it was expected that these results are imprecise. However, the large numbers may still indicate the prevalence of method-level control flows that *allow for* sensitive data leaks in the benchmarks. Since there were many more sources exercised than sinks (see Figure 9), a sink was more likely to be reachable from a source than a source to have a reachable sink, thus higher percentages of risky sinks were seen than were percentages of risky sources.

The instance view shows a similar contrast between risky sources and sinks, but much lower percentages of both (8% and 25% by average) over all instances of source and sink calls. This observation implies that although considerable percentages of exercised sources and sinks were risky, the risky sources and sinks were not as much frequently called as non-risky ones.

Further examining the types of information accessed by the risky sources and types of operations performed by the risky sinks revealed that the more prevalent types of sources and sinks were more likely to be risky as well, in both the unique and instance views (see Appendix B for the detailed categorization results).

6.3.4 *Summary and Discussion*

Our security metrics show that (1) sensitive information accesses and critical operations are commonly involved in Android app executions, though they are not heavily executed (accounting for less than 15% of all calls even with respect to the highly-comprehensive lists of predefined sources and sinks), (2) the target information of sources and target operations of sinks are both in narrow scopes: over 90% of sources focus on accessing network information, while about 85% of sinks focus on operations related to account setting and logging, (3) substantial portions of exercised sources and sinks can potentially channel data leaks thus be risky, (4) these risky sources and sinks tend to be executed less frequently than non-risky ones, and (5) the more commonly accessed sensitive data tends to be more likely to be leaked, and the more commonly invoked critical operations tend to be more likely to leak sensitive data.

In light of these findings, Android app security analysis may *prioritize on the few predominant categories of sources and sinks* to avoid being overly conservative in discovering taint flows to gain better overall tradeoffs between precision and efficiency. End users and security experts should also pay *more attention to these highly accessed categories* to make better decisions on permission management and app vetting. The large room that method-level control flows left for potential data leaks calls for effective app protection against such threats which would *need more fine-grained (than method level) and comprehensive (checking data flows also) analysis*. Meanwhile, the rapid call-graph-based source-sink reachability computation may *quickly and largely reduce the inspection scope of precise* dynamic *taint analysis and data-leak detection* to achieve better scalability (by focusing on the risky sources and sinks).

6.4 Inter-App Benchmark Suite

Several Android app benchmark suites have been shared by researchers [1, 44, 49] but all target a single-app static analysis. For an inter-app dynamic analysis, a suite of apps with known communicating peers in the same suite is more desirable. At the early stage of our study, we faced the challenge of finding such a suite. Now we have created one and released it for public reuse.

Our study results have confirmed that out of the 61 potentially communicating app pairs (based on static ICC resolution and matching) 45 have ICCs between the pair that have been exercised readily by random inputs. Of these 45 pairs, 23 have ICCs going in both directions between the app pair. The package names and exercised ICC statistics of each pair are available for download from our project website (see Section 4), where we have hosted corresponding APKs for free, handy downloads too.

6.5 Threats to Validity

One threat to internal validity of our study results lies in possible errors in the implementation of our toolkit DROIDFAX and various experimentation scripts. To reduce this threat, we have conducted careful code review of the toolkit and scripts in addition to manual verification of their functional correctness against our experimental design. The maturity of the Soot framework supporting our toolkit also helps increase the credibility of our tool implementation. Another internal threat comes from the possibility of missing or incorrectly placing profiling probes during the instrumentation due to code obfuscation, which has been a significant block for static analysis of Android apps [12, 31]. However, our manual inspection of Soot Jimple [28] code and call traces for the benchmark apps revealed that for the few obfuscated apps this did not affect our simple static analysis.

Primary threats to external validity concern our choice of benchmark apps and the test inputs we used for dynamic analysis. First, we studied a limited number of Android apps, which may not be representative of all Android apps on the market or in use. To mitigate this threat, we started with a much larger pool of apps that were ranked as top popular apps recently, and picked each benchmark app from that pool randomly. In addition, our study targeted relatively new apps which were built on Android SDK 5.0 or above. Thus, our results may not generalize to older apps. However, with the Android app market migrating to newer platforms [22], we believe that studying Android apps with respect

to more recent platforms gives more valuable information for the development and protection of future Android applications.

As any other dynamic analyses, our empirical results are subject to the coverage of the test inputs we used—some behaviours of the benchmarks might not have been exercised. To reduce this threat, we used the tool that gives the highest coverage among peer tools [4] (and only 5% lower than the highest reported so far which was achieved by manual inputs in a much smaller-scale study [31]). Nonetheless, our conclusions and insights are limited to the covered app behaviours, and inputs of considerably higher coverage may give stronger/different results (by reusing our methodology and toolkit). Possible non-determinism in the chosen apps could also be a threat, so we repeated our experiments three times and took the average metric values for each app and app pair, and found only marginal variances for most apps and pairs.

The main threat to conclusion validity concerns the distribution of underlying data points, in light of the large variances of metric values, for a few measures (e.g., categorization of data-carrying ICCs and callbacks). In those cases, the non-uniformity of the data distribution prevented stronger claims, but it did not affect the current observations. As none of the benchmarks were known as malicious, our results may not generalize to malware.

7. RELATED WORK

7.1 Dynamic Analysis for Understanding

Various dynamic analysis techniques have been employed for program understanding in general [6], of which the most relevant to us is execution trace analysis (e.g., [35, 46]). Yet, many existing works of this kind concerned techniques aiming at effectively exploring the traces themselves, such as trace reduction [24] and visualization [5]; and others serve different purposes other than directly for understanding the behaviour of programs, including evolution tracking [18] and architecture extraction [27].

We also employed trace analysis for program-understanding purposes. However, instead of exploring the traces directly or improving trace analysis techniques, we focused on studying the functionality structure and runtime behaviour of programs by utilizing execution traces as a means. Also, we target mobile applications, unlike the majority of prior approaches which addressed other domains such as traditional desktop software.

A few dynamic analyses focusing on Android involved tracing method calls as well, for malware detection [46], app profile generation [45], and access control policy extraction [13]. Yet, their main goal was to serve individual apps thus different from ours of characterizing Android application programming as a whole. In addition, their call tracing aimed at Android APIs only, whereas our execution traces covered all method calls including methods defined in user code and third-party libraries.

7.2 Android Security Analysis

Recent years have seen a growing body of research on securing Android apps against a wide range of security issues [12]. Among the rich set of proposed solutions [43], modeling the Android SDK and other libraries [1, 44], approximating the Android runtime environment [23, 30], and analyzing lifecycle callbacks and event handlers [1, 48] are the main underlying techniques for a variety of *specific* vulnerability and malware defense approaches [43]. Examples of such specific approaches include information flow analysis [8, 42] in general and taint analysis [1, 36] in particular.

In comparison, we are concerned about similar aspects of the Android platform and its applications, such as different layers of app code and interactions among them, as well as lifecycle methods and event handlers. We also performed a coarse (i.e., method level) and approximated (i.e., control-flow based) taint analysis. However, rather than proposing or enhancing these techniques themselves, we empirically characterized sample Android apps with respect to relevant app features and investigated how such features discovered from our study would affect the design of those techniques. Also, different from many of them that are purely static analyses, our work is dynamic. On the other hand, compared to existing dynamic approaches to security analysis for Android which were mostly platform extensions (i.e., modifying the SDK itself), our work did not change the Android framework but only instrumented in Android apps directly as in [26].

7.3 Characterization of Android Apps

Empirical works targeting Android also have emerged lately. Previous studies have covered a broad scope of topics ranging from general security vulnerabilities [10], resource usage [17], and battery consumption [16] to permission management [15], code reuse [41], ICC robustness [32], SDK stability [34] and user perception of security and privacy [14]. In contrast, our work aims at the general characterization of Android applications from the point of view of programming paradigms and language constructs. Also, different from these prior works which leveraged static analysis and/or user studies (e.g., survey and interview), we utilize comprehensive method call profiling and message-passing tracing to investigate the dynamic semantics of Android apps.

For categorizing data accessed by sensitive and critical API calls, we created lists of predefined sources and sinks from those used in [40], where frequencies of source and sink usage in malware samples were studied. Several other works on Android app characterization also targeted Android malware [47, 49]. These studies either utilized static analysis [40, 47] or relied on manual investigation [49]. In contrast, we targeted understanding the run-time behaviours of benign Android apps via dynamic analysis, which potentially complements those previous studies.

8. CONCLUSION

We presented the first systematic dynamic study of Android apps that targets a general understanding of application security in Android from the perspective of run-time app behaviours. To that end, we traced method calls and ICC Intents from one-hour continuous executions of 114 top popular apps randomly selected from Google Play and 61 communicating pairs among them. We developed an open-source toolkit DROIDFAX and applied it to characterize the execution structure, usage of lifecycle callbacks and event handlers, ICC calls and data payloads, and sensitive data accesses of Android apps at runtime. We also have produced an app-pair benchmark suite and its exercised ICC statistics that are particularly useful for dynamic Android inter-app analysis.

Our results reveal that (1) Android apps are heavily dependent on various libraries, especially the Android SDK, to perform their tasks, (2) only a small portion of method calls target lifecycle callbacks or event handlers, (3) most ICCs during Android app executions do not carry any data payloads and the rest pass data mainly via bundles instead of URIs, (4) while sensitive and critical APIs are substantially invoked, they mostly focus on only a few kinds of information and operations, and (5) the more commonly accessed sensitive information is more likely to be leaked, and the more commonly performed critical operations are more likely to leak sensitive data. We have offered considerable insights into the security implications of our empirical findings that potentially inform both secure development of future Android apps and the design of cost-effective security defense solutions for Android.

9. REFERENCES

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

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

[3] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.

[4] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015.

[5] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[6] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[7] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[8] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[10] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, pages 21–21, 2011.

[11] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE security & privacy*, (1):50–57, 2009.

[12] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *Communications Surveys & Tutorials, IEEE*, 17(2):998–1022, 2015.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[14] A. P. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 33–44. ACM, 2012.

[15] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[16] D. Ferreira, A. K. Dey, and V. Kostakos. Understanding human-smartphone concerns: a study of battery life. In *Pervasive computing*, pages 19–33. Springer, 2011.

[17] D. Ferreira, V. Kostakos, A. R. Beresford, J. Lindqvist, and A. K. Dey. Securacy: an empirical investigation of android applications' network usage, privacy and security. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 11. ACM, 2015.

[18] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind. System evolution tracking through execution trace analysis. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 237–246. IEEE, 2005.

[19] Google. Android emulator. http://developer.android.com/tools/help/emulator.html, 2015.

[20] Google. Android logcat. http://developer.android.com/tools/help/logcat.html, 2015.

[21] Google. Android Monkey. http://developer.android.com/tools/help/monkey.html, 2015.

[22] Google. Android Developer Dashboard. http://developer.android.com/about/dashboards/index.html, 2016.

[23] M. Gordon, D. Kim, J. Perkins, L. Gilhamy, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in DroidSafe. In *NDSS*, 2015.

[24] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 181–190. IEEE, 2006.

[25] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92. IEEE, 2012.

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

[27] T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474–492, 2007.

[28] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. Soot - a Java bytecode optimization framework. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[29] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

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

[31] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[32] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

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

28th Annual Computer Security Applications Conference, pages 51–60. ACM, 2012.

[34] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[35] J. Moc and D. A. Carr. Understanding distributed systems via execution trace data. In *Proceedings of International Workshop on Program Comprehension*, pages 60–67, 2001.

[36] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium*. The Internet Society, 2005.

[37] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering*, pages 77–88, 2015.

[38] 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 *Proceedings of USENIX Security Symposium*, 2013.

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

[40] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[41] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122. IEEE, 2012.

[42] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek. Information flows as a permission mechanism. In *ASE*, pages 515–526, 2014.

[43] D. J. Tan, T.-W. Chua, V. L. Thing, et al. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):58, 2015.

[44] 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*, pages 1329–1341, 2014.

[45] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[46] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and API calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[47] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Computer Security-ESORICS 2014*, pages 163–182. Springer, 2014.

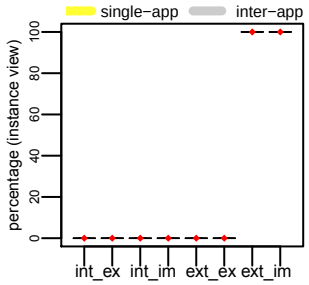
[48] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 89–99. IEEE Press, 2015.

[49] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

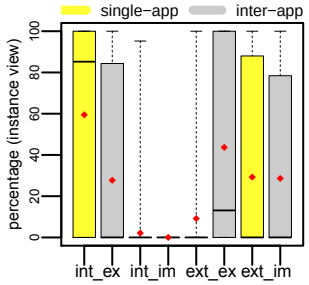
[50] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, pages 93–107. Springer, 2011.

APPENDIX

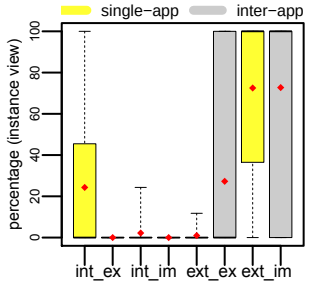
A. CATEGORIZATION OF ICC INTENTS CONTAINING DATA PAYLOADS



(a) standard data only



(b) bundle data only



(c) both forms of data

Figure 11: The distribution of ICCs carrying standard data only (a), bundle data only (b), and both forms of data (c) over the four ICC categories, in both single- and inter-app settings.

Figure 11 depicts the results of our deeper look at the data carriage of ICCs. The three charts, 11a, 11b, and 11c, break down the ICCs that carried standard data only, bundle data only, and both forms of data, respectively, showing the same categorization as in Figure 7 for these ICCs—the four possible categories as

listed on the x axis: internal explicit (*int_ex*), internal implicit (*int_im*), external explicit (*ext_ex*), and external implicit (*ext_im*). Each data point (i.e., percentage) was calculated the same way as for Figure 7 except that the population was not the total ICC Intents but those carrying data of either or both forms.

Chart 11a reveals that the few standard-data-only ICCs (around 5% of all) were uniformly all (100%) external implicit; that is, the apps tended to exchange data with peers by specifying URIs and using implicit ICCs if they did not pass anything through the `extras` field of the ICC Intents. This finding was consistent between the single- and inter-app settings.

Chart 11b shows that, in the single-app setting, over 80% of the ICCs that only used bundles to pass data were employed for explicitly exchanging the data between *intra-app* components, as seen by most of the individual apps. By comparison, external ICCs, especially explicit ones, very rarely contained data only in their `extras` field despite a few outlier apps having ICCs doing that considerably. In the inter-app setting, the largest mean and median for external explicit ICCs suggest that there were more app pairs exchanging bundle-only data explicitly than pairs where the apps pass such data between components with the other three categories of ICCs. In contrast, there were almost no app pairs where the apps pass bundle-only data using internal implicit ICCs.

Finally, chart 11c shows how the ICCs carrying both standard and bundle data, only found in a few outlier apps (see chart 8), are distributed over the four ICC categories. In both settings, external implicit ICCs (with median percentage close to 100%) dominated the other three categories (with median percentage of almost zero) of such both-data ICCs in most apps or app pairs.

B. CATEGORIZATION OF RISKY SOURCES AND SINKS

Table 5: Risky source breakdown over significant categories

Category	Unique view	Instance view
ACCOUNT_INFO	0.08% (0.75%)	0.47% (5.20%)
CALENDAR_INFO	0.74% (3.70%)	0.60% (4.48%)
LOCATION_INFO	0.06% (0.67%)	0.02% (0.19%)
NETWORK_INFO	84.22% (35.37%)	84.06% (35.74%)
SYSTEM_SETTINGS	0.11% (1.51%)	0.06% (0.79%)

Table 6: Risky sink breakdown over significant categories

Category	Unique view	Instance view
ACCOUNT_SETTINGS	67.33% (37.12%)	66.05% (40.09%)
FILE	0.24% (1.55%)	0.28% (2.51%)
LOG	13.47% (23.81%)	12.35% (27.46%)
NETWORK	0.67% (3.78%)	0.47% (3.74%)
SMS_MMS	0.13% (0.85%)	0.03% (0.46%)
SYSTEM_SETTINGS	3.37% (6.34%)	6.02% (15.27%)

Tables 5 and 6 present the results on the categorization of *risky* sources and *risky* sinks according to the types of sensitive information accessed and kinds of critical operations performed, in the same format as Tables 3 and 4, respectively.

The results show that (1) the top (significant) categories were consistent with those of all exercised sources and sinks (see Tables 3 and 4), (2) the distributions of risky sources and sinks were also consistent with those for the total sources and sinks invoked during app executions. Meanwhile, the most dominant category of exercised sources (i.e., network information) was less dominant among the risky categories, and account information tended to be relatively more commonly accessed by risky sources. As to the sink categorization, though, the category of account settings appeared to be even more dominant, while the category of logging operations appeared to be considerably less, among risky sinks than it was among all exercised sinks.

Nevertheless, the general observations from this detailed categorization are that (1) the more prevalently accessed types of information (e.g., network information followed by system settings) were also the more dominant types accessed by *risky* sources, and (2) the more commonly invoked types of critical operations (e.g., account settings followed by logging) were the more dominant types performed by *risky* sinks as well.