2017 IEEE International Conference on Software Maintenance and Evolution

Understanding Android Application Programming and Security: A Dynamic Study

Haipeng Cai
Washington State University, Pullman, USA
hcai@eecs.wsu.edu

Barbara G. Ryder
Virginia Tech, Blacksburg, USA
ryder@cs.vt.edu

***Abstract*—Most existing research for Android focuses on particular security issues, yet there is little broad understanding of Android application run-time characteristics and their implications. To mitigate this gap, we present the first systematic dynamic characterization study of Android apps that targets a broad understanding of application behaviors in Android. Through lightweight method-level profiling, we collected 59GB traces of method calls and Intent-based inter-component communication (ICC) from 125 popular Android apps and 62 pairs among them that enabled an intensive empirical investigation of their run-time behaviors. Our study revealed that, among other findings, (1) the application executions were overwhelmingly dominated by the Android framework, (2) *Activity* components dominated over other types of components and were responsible for most lifecycle callbacks (3) most event handlers dealt with user interactions as opposed to system events, (4) the majority of exercised ICCs did not carry any data payloads, and (5) sensitive data sources and sinks targeted only one/two dominant categories of information or operations. We also discuss the implications of our results for cost-effective program analysis and security defense for Android.**

I. Introduction

The Android platform and its user applications (referred to as *apps*) dominate the mobile computing arena, including smartphones, tablets, and other consumer electronics [1], [2]. Meanwhile, accompanying the rapid growth of Android apps is a surge of security threats and attacks of various forms [1], [2]. 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 *supposed to* rely on the Android SDK and other third-party libraries [3], [4]. In fact, many of the distinct characteristics of Android apps have led to unique challenges in developing sound and effective code-based app 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 [3]–[5] to achieve reasonable accuracy. Implicit invocation between components in Android apps through a mechanism called inter-component communication (ICC) requires special treatments (e.g., ICC resolution [6], [7]) for a soundy [8] 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 [3], [5], [9] and computing callback control flows [9], [10].

Existing research on Android has been predominantly aimed at security (as lately surveyed in [2]). Most solutions targeted specific security issues, with merely a few offering a broader view of application security related characteristics in general [11], [12]. Several recent studies on Android apps investigated aspects other than security [13]–[16] but targeted *static* characterizations by examining the source code rather than the run-time behaviors of the apps. Existing *dynamic* studies for Android address individual apps (e.g., [17]) or focus on malware (e.g., [18], [19]) rather than performing a general behavioral characterization.

We randomly chose 125 free Android apps from Google Play and 80 linked pairs among them, exercised each app and app pair with automatically generated inputs attaining a mean line coverage of 70% per app, and gathered 59GB traces of ordinary function calls and ICCs. From these traces, which reasonably represent how the chosen apps are executed, we characterize applications on the primary Android market from both security and program understanding perspectives. In particular, we stress metrics relevant to the 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.

These metrics and corresponding results constitute *the first systematic dynamic characterization of Android apps*, which will benefit customization and/or optimization of future Android app analyses. For the longer term, our findings will inform better design of techniques and methodologies for more effectively developing and securing Android apps. Further, changes in these behavioral characteristics over time will reveal how Android apps evolve.

The main contributions of this work include:

- A dynamic study of the layered composition and functionality distribution of Android apps, which sheds light on their run-time structure and its security implications. The study reveals that (1) Android apps are extremely framework-intensive, with around 90% of all callers and callees being SDK methods and (2) constantly the most (60–90% of all) exercised components are *Activities*, which receive most (about 60% of all) exercised lifecycle callbacks.
- An intensive investigation of Intent-based ICC, the main inter-component communication mechanism in Android, which suggests optimization strategies for ICC-involved program analyses of Android apps. The investigation reveals that (1) most ICCs (70–80%) do not carry any data payloads and (2) ICCs that carry data payloads favor

bundles over URIs, particularly in inter-app ICCs (>20% using bundles versus <5% using URIs).

- 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) up to 5–10% of method calls access sensitive data/operations (with respect to highly comprehensive source/sink lists) and (2) constantly most (90% of all) exercised sensitive calls target only one or two particular (out of over ten) types of data/operation.
- An open-source dynamic study toolkit including an Android app (line) coverage measurement tool that does not rely on source-code access and various predefined categorizations that can be reused for future studies and understanding of Android apps, along with a benchmark suite of dynamically communicating (via ICC Intent) app pairs that can support other Android studies and analyses, especially *dynamic inter-app* analyses.

II. 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 APIs as part of its SDK which implements 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 their executions via various callbacks, including *lifecycle methods* and *event handlers* [9].

Four types of components are defined in Android, *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*, as the top-level abstraction of user interface, background service, response to broadcasts, and data storage, respectively [20]. The SDK includes APIs for ICC by which components communicate primarily via ICC objects called *Intents*. We focus on ICCs based on Intents that 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 Android OS at runtime.

Some information on mobile devices is security-sensitive, such as device ID, location data, and contacts [11], [20]. 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* [9], [21]. 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*).

III. Experimental Methodology

We traced method calls and Intent 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 behaviors 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 process.

A. Benchmarks and Test Inputs

We randomly downloaded 3,000 free apps from Google Play and statically analyzed the ICCs of each app using the most precise current ICC analysis [7] to find *potentially* communicating app pairs by matching the ICCs across apps [22]. 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 [23]. To ensure that we gathered useful traces and that our study reflected the use of current Android SDK features, we discarded pairs in which at least one app (1) was built on a version of Android SDK lower than 4.4 (API 19) or (2) failed to run on the emulator after the instrumentation, or (3) did not have at least 55% (as is higher than the average line coverage achieved by Monkey [24], [25]) user code covered by Monkey inputs. We repeated this random selection until we obtained 80 different potentially communicating app pairs which included 125 unique apps and covered *all* Google Play app categories (e.g., `Sports` and `Tools`).

Previous dynamic studies of Android apps, using much smaller benchmark suites, mostly resorted to manual (expert) inputs [21], [26]–[29], 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 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 [24] and they are scalable. The latest, most comprehensive comparison of such generators showed that the Monkey [30] tool outperformed over its peer approaches in terms of user code coverage [25] (another tool [31] achieved slightly higher average coverage of 55% yet it is not as applicable). Therefore, we utilized Monkey for our study. Although Monkey does not generate many system events directly, it triggers those events indirectly through UI events. The coverage of our per-app traces ranged from 55% to 91% (mean=70.1%, standard deviation=11.2%). The Monkey inputs we used also exercised *inter-app* communication for 62 out of the 80 app pairs. Eventually, the 125 single-app traces and 62 inter-app traces formed the basis of our empirical study.

B. Metrics

We characterize run-time behaviors of Android apps via 122 metrics in *three complementary perspectives/dimensions* each consisting of several supporting measures defined as

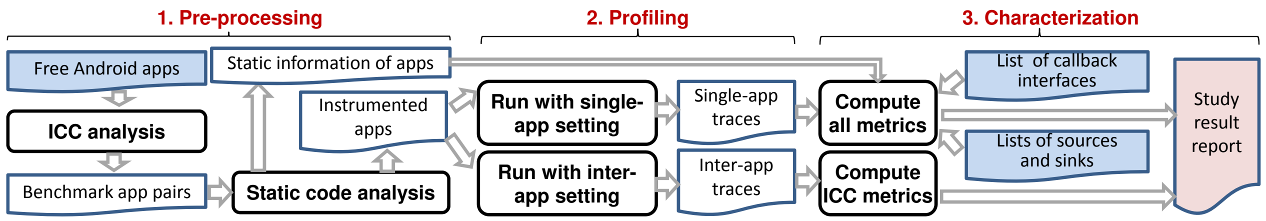

Fig. 1: The three phases of our characterization process, including the inputs and outputs.

follows. Metrics in the ICC dimension also cover *both intra- and inter-app communications*. Thus, our dynamic characterization is *systematic* in terms of the study scope.

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 (*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 Intent-based inter-component interaction within single apps and across multiple apps. ICC has been a major security attack surface in Android [6], [22], [32] 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 II) 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) and (2) the categories of information accessed by executed sources and operations performed by executed sinks.

For each of these measures, with each app, we examined the full execution traces of method calls and exercised Intent ICCs for the app. We considered all instances for each call and ICC (i.e., accounting for the frequency) to capture the run-behaviors of the app. For understanding inter-app ICCs, we also examined the traces for each pair of apps. We trace all calls including those via exceptional control flows [33] by reusing our previous relevant utilities [34].

C. 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 the single-app and 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 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. 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 in the statistics of our results.

IV. Characterization Process

Figure 1 depicts the process of our characterization study, including its three phases as well as inputs and outputs.

Pre-processing. After obtaining the benchmark app pairs as described in Section III-A, 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 [35].

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. We recorded method calls and ICC Intents using the Android logging utility and collects the traces using the `logcat` tool [36].

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, we calculated metrics of all three dimensions. From inter-app traces only the ICC metrics are computed. In order to categorize event handlers, we utilized a predefined categorization of callback interfaces, which we manually produced from the uncategorized list used by FlowDroid [9]. We did the categorization based on our understanding of each interface according to the official Android SDK documentation. Lifecycle callbacks were categorized 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 [37] hence producing a more precise categorization.

To facilitate reproduction and reuse, we released the open-source implementation of our study utilities as an open-source toolkit DroidFax [38], including a line coverage tracking tool directly working on an APK. Given a configured Android emulator or device and a set of apps and/or app pairs, the automated study workflow produces both metrics values and their visualization and tabulation. Also available are our study results, the categorization of event handlers we created, the improved source and sink categorization we generated, and other documentation including the detailed definition of the 122 metrics used.

V. Research Questions

With respect to the metrics described above, our study seeks to answer 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 [4], [5]. RQ1 is answered using the first two measures (i.e., (1) and (2)) 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 [3], [9], [10]. 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 (i.e., (3)) of the general metrics.

RQ3: How do Android app components communicate using the ICC mechanism? Much prior research has targeted Android security concerning ICCs [6], [7], [22], [32], yet it remains unclear how often ICCs occur relative to regular function calls during app executions, how different types of ICCs are used, and whether all ICCs constitute security threats. The answers to each of these questions are subsumed by RQ3, and are 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 [3], [9], privilege escalation defense [5], [27], and data leakage detection [22], [39]. However, how often that usage is exercised or which kinds of sensitive information are mostly accessed has not been studied. RQ4 explores these questions using the security metrics.

VI. 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 report the number of instances of each executed callsite throughout all single-app traces using scatterplots. For callback and source/sink categorization, we rank the categories for each app and report for each category the mean rank across all benchmarks along with the standard deviation of the ranks. 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 report either the distribution of all these percentages using boxplots or their summary statistics (mean and its 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 middle bar 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).

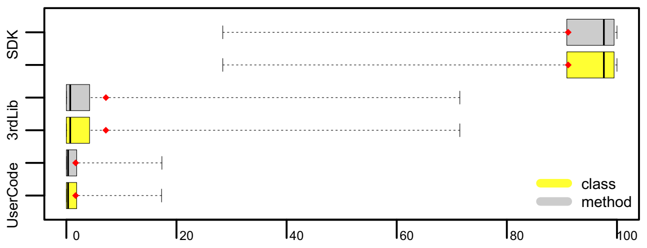

Fig. 2: Percentage distribution (x axis) of method calls to the three code layers (y axis) at class and method levels.

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 development and security defense of Android apps.

A. General Characteristics of Android Apps

To gain a general understanding of Android app behaviors, 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.

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 call instances accessing user code, third-party libraries, and the Android SDK. Figure 2 shows the distribution of these layers in all the single-app traces, with each group of boxplots depicting both class and method granularity.

The plots reveal that consistently all the subject apps employed library functionalities extensively, especially the SDK, in performing their tasks. On average, at both class and method levels, SDK code was executed the most frequently among the three layers, suggesting that run-time behaviors of the SDK dominate in the apps. The observation that over 90% (on average) of all calls were to the SDK code in almost all apps corroborates that Android apps are highly framework-intensive. In contrast, these apps tend to execute their user code relatively occasionally—in fact, only 25% of the apps had over 2% of their calls target user code and none had over 20%. Third-party library code was not called frequently either albeit more so than user code: the means are larger than the 75% quartile, implying that only *a few* outlier apps had over 10% calls to third-party libraries.

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*$\rightarrow$*callee layer*, among the three code layers. Each plot shows the call-frequency ranking for one of

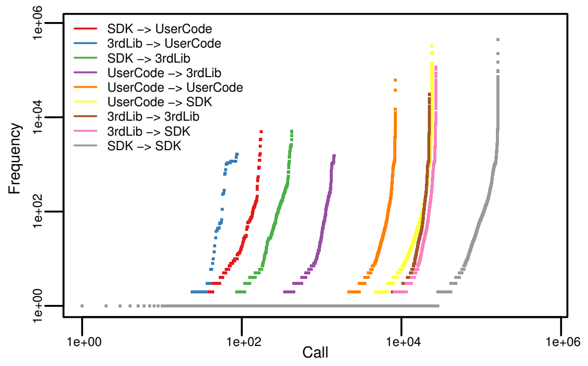

Fig. 3: Executed callsites of all apps (x axis) ordered (per pair of layers) non-descendingly by their frequencies (y axis).

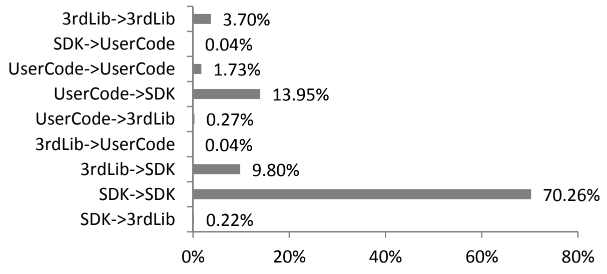

Fig. 4: Percentages of different categories of inter-layer call instances over all benchmark app executions.

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 overwhelmingly 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) followed by those 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 (75.7%) of call instances over all apps happened *within* the same layer—dominated by the SDK layer (70.3%)—rather than across layers. 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.

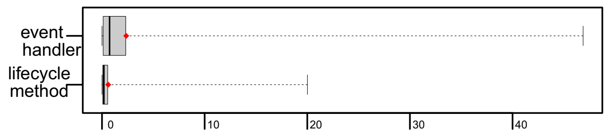

Fig. 5: Percentage distribution (x axis) of callbacks, including lifecycle methods and event handlers (y axis), over all calls.

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.

3) Usage of Callbacks: We examined the extent of callback usage (over all code layers) in the benchmark apps through the distribution of percentages of callback method invocations. As shown in Figure 5, callbacks in these apps were not invoked very frequently. The average percentage of either type of callback invocations was under 3%, indicating that callbacks, while prevalently defined and registered in Android apps [9], [10], 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 reveals that (1) event handlers were called more frequently than lifecycle callbacks (2.7% versus 0.6%) and (2) there were apps executing event handlers substantially (up to 47% of total call instances) yet none of the apps had more than 20% of calls targeting lifecycle methods. Note that the medians were all below 2%, for both lifecycle and event-handling callbacks, implying the generally light invocations of callbacks overall in these apps.

To look further into the callback usage, we categorized lifecycle callbacks by their enclosing classes with respect to the four types of application components and the *Application* type corresponding to the `android.app.Application` class defined in the SDK. The rank (by the number of call instances) of each category is listed in Table I, including the means (of ranks) and corresponding standard deviations over all 125 benchmark apps. From the second to the sixth row of the table, the component types are listed in a non-ascending order by the corresponding rank averages.

As shown, *Activity* lifecycle methods were invoked most frequently in comparison to such methods in other categories. In fact, our results also show that these methods dominated the targets of all executed lifecycle-method calls. In the vast majority of the apps, *Activity* was consistently ranked the first (i.e., rank=1) among the five classes of lifecycle-method call receivers, with a few apps having this component type ranked the second (i.e., rank=2). As a result, the average rank was 1.28 for *Activity*. The second most handled lifecycle events were associated with the application as a whole, with an average rank of 1.76. Events handled by the other three types of components were close in their mean ranks, and were all considerably lower than the two dominant categories (i.e., *Activity* and *Application*).

In addition, on average the apps had over 75% of all lifecycle callbacks associated with *Activity* components. These numbers suggest that the vast majority of lifecycle

TABLE I: Lifecycle methods breakdown over all categories

Category	rank average	standard deviation
Activity	1.28	0.54
Application	1.76	0.61
ContentProvider	2.49	0.79
BroadcastReceiver	2.48	0.77
Service	2.54	0.77

TABLE II: Significant categories of invoked event handlers

Category		rank average	standard deviation
UI	View	1.81	0.69
System	System mgmt.	1.89	1.09
UI	App bar	2.07	1.21
UI	Dialog	2.46	0.93
UI	Widget	2.56	0.99
System	App mgmt.	2.63	1.22
System	Media control	2.69	1.15
System	Hardware mgmt.	2.72	1.21

method calls were dealing with *Activities*, hinting at the pattern that Android apps typically have abundant user interfaces (UI) and rely on frequent user interaction. Further, the small standard deviations (all less than half of the associated means) suggest that the pattern was pretty consistent across the benchmark apps. 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 [10], to reduce analysis complexity and/or to achieve better performance.

Exploring the data further, Table II presents a two-level breakdown of invoked event handlers according to our manual categorization of those callbacks (see Section IV). When at least one of the benchmark apps had over 1% of all call instances falling in a (second-level) category, we regarded that category as *significant*. We only report significant categories.

In the table, all the significant second-level categories are listed in a non-ascending order by the corresponding rank averages. Overall, in most of these apps, there were more callbacks triggered by UI events than those handling system events. The top three categories were pretty consistently ranked at top in most apps, while the other categories were close in the average ranks implying the ranking of these categories had more variations across different apps.

A more detailed look reveals that the majority of UI event handlers dealt with two particular kinds of user interfaces, *View* (generally ranked the first with an average rank of 1.81) 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 system management (*mgmt.*), with a few others dealing with app and hardware management and media control. The relatively small standard deviations of the average ranks imply that the sample means capture the general traits of these apps. Given these results, Android app analyses of event handlers [3], [9], [10] could be customized or optimized for better cost-effectiveness while remaining soundy [8] by prioritizing analysis of those in the most commonly exercised categories.

4) Summary and Discussion: The general metrics show that at runtime Android apps (1) depend heavily on the SDK—over 80% of methods executed are defined in the SDK and (2) are highly framework-intensive—90% of all call instances are those of SDK methods, and the largest

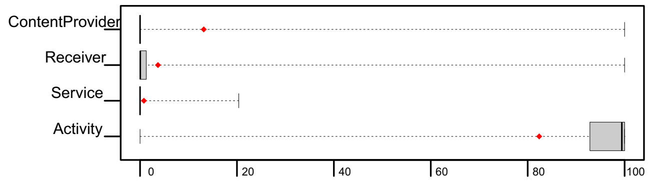

Fig. 6: Percentage distribution (x axis) of the four types components (y axis) in app executions.

numbers (over 10K) of calls with the highest frequencies (over 100K) targeted methods in the SDK. Thus, a clear, deep *understanding of the SDK and its interface with apps is essential* for secure Android app development. Meanwhile, *the security of the Android framework itself deserves foremost attention* in securing the whole Android software ecosystem. The overwhelming dominance of SDK in app executions uncovers promising *benefits of SDK optimizations.*

In addition, *Activity* was the predominant (above 70%) target of lifecycle method calls, which indicates that Android apps are generally rich in user interfaces. Therefore, Android app analyses should pay considerable *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 would 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-ranked categories of lifecycle methods and event handlers would render lifecycle modeling, taint analysis, and callback control flow analysis *more cost-effective (e.g., sacrificing safety for higher scalability)*.

B. ICC Characterization

ICCs constitute the primary communication channel between the four types of components as well as a major attack surface in Android. We first look at component distribution in the app executions before examining the interaction between them through Intent ICCs. We then characterize whether data payloads are carried (i.e., *data carriage*) in the ICCs. We report the measures based on exercised ICC calls with respect to single- and inter-app traces separately and compare findings in these two settings.

1) Component Distribution: Figure 6 shows the distribution of method calls over different component types. 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 (almost 90% on average) among all components executed. *Receivers* were used noticeably (about 5% on average), related to the previous observation that these apps had considerable percentages of callbacks handling system events over all invoked callbacks (see Table II). Except for in one outlier app where 20% executed components provide background service, *Service* components were generally invoked very sparingly.

2) ICC Categorization: Having an understanding of the usage of different types of components, we now break down all exercised ICCs (i.e., links between components), as shown in Figure 7, over four possible categories (on the x axis) in single- and inter-app traces separately. Each data

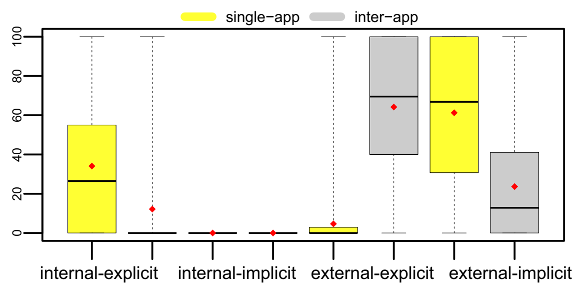

Fig. 7: Percentage distribution (y axis) of all ICCs over four categories (x axis) in single-app versus inter-app executions.

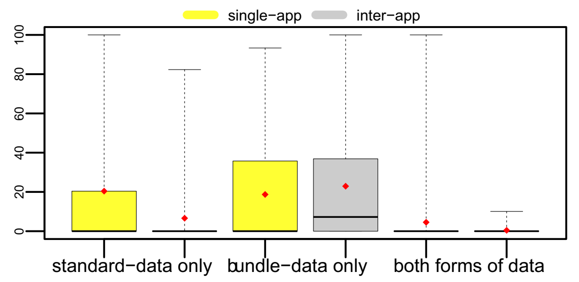

Fig. 8: Percentage (y axis) of ICCs that carried data payloads in different forms (x axis) over all ICCs exercised.

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 the dominance of internal-explicit and external-implicit ICCs. In contrast, there were much smaller percentages of external-explicit ICCs while internal-implicit ones were even fewer. These observations suggest that components within the same app tend to connect explicitly—in fact, they rarely communicate via implicit ICCs. There were substantial percentages of external ICCs even in the single-app traces because system and built-in apps (e.g., photo viewer, camera, web browser, maps, etc.) often communicated with our benchmark apps (almost always via implicit ICCs according to our results).

In the inter-app setting, external-explicit and external-implicit ICCs combined (over 80% on average) dominated over 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 (60%) than external-implicit ones (25%). 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 (external) ICCs were not successfully exercised thus ignored in our study. Nonetheless, these apps more preferably used explicit ICCs when communicating with peer user apps than when communicating with system/built-in apps (as observed in the single-app setting).

We also found that less than 0.5% 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; in comparison, components communicated with other components only occasionally (e.g., when an intra-component computation completed and results were ready to deliver). More than half of all ICCs were between two *Activity* components; among the other ICCs, over 60% were either initiated by or aimed at *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 tends to serve the communication between various user interfaces.

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 the percentage distribution (y axis) of data-carrying ICCs with different forms of data (x axis) over all ICC instances. The single-app traces saw similar usage of standard as that of bundle data, although the latter was favored in more apps. In the inter-app setting, however, on average no more than 5% of ICCs transferred standard data only, while bundle-only ICCs were over 20% of the same total. Very few ICCs carried both standard and bundle data at the same time, though, in either communication setting. Despite 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 (especially between user app pairs) 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 would miss the majority of potential data leaks. Instead, security analyses of ICC-based data leaks should carefully examine the bundles contained in ICC Intents [29].

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 (over 35% and below 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 [3], [6]) 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 across apps tend to do so preferably via bundles (over 20%) instead of URIs (below 5%). Thus, security analyses involving 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.

The characteristics of ICCs in single-app executions are different from those in inter-app settings: (1) percentages of

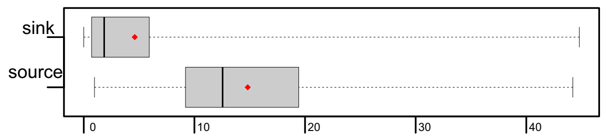

Fig. 9: Percentage distribution (x axis) of sources and sinks (y axis) over all calls.

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 (around 20%) larger percentages of all external ICCs than the former; and (2) components across apps connected more often via implicit ICCs in single-app traces (about 60%) than in inter-app ones (below 5%) while external explicit ICCs were much (over 60%) more often seen in inter-app traces. 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* (potential communicating peer apps).

C. Security-Sensitive Data Accesses

Android security analysis has been largely concerned with inappropriate accesses to security-sensitive data. To understand the implications of those accesses, we investigated in our benchmarks (1) the usage of sensitive and critical APIs and (2) the categories of sensitive data the sensitive APIs (sources) accessed and categories of critical operations the critical APIs (sinks) performed.

1) Usage of Sources and Sinks: Since sensitive data in Android is accessed via invocations of sensitive and critical SDK APIs, understanding 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 apps tended to retrieve sensitive information often, containing on average one sensitive API call out of every ten callsites exercised during their executions. Half of the apps had an even larger proportion (up to 22%) of sensitive API calls.

The exercised sources and sinks were run-time projections of the predefined lists [37] of (17,920 unique) sources and (7,229 unique) sinks we used, respectively. Thus, the percentages of source and sink calls we reported were directly influenced by the sizes of these lists.

Our results show that sources were invoked much more frequently than the 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, with up to 45% in a couple of outlier apps. In short, the apps had considerably intensive accesses to sensitive information, yet did not perform potentially data-leaking operations as much often. This difference can be partially ascribed to the much longer list of predefined sources compared to the shorter list of predefined sinks.

2) Categorization of Sensitive Data Accesses: One way to further examine how Android apps use sensitive and critical API calls is to look into the information itself accessed by

TABLE III: Source breakdown over significant categories

Category	rank average	standard deviation
Network information	1.0	0.0
System settings	2.13	0.49
Calendar information	2.27	0.60
Location information	2.38	0.65
Account information	2.40	0.68

TABLE IV: Sink breakdown over significant categories

Category	rank average	standard deviation
Account setting	1.18	0.41
Logging	2.21	0.91
System setting	2.36	0.81
Network operation	3.10	1.18
File operation	3.25	1.24
SMS/MMS	3.33	1.30

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 often 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.

Table III lists the rank (by the numbers of source call instances accessing sensitive information) of each source category, in the same format as Table I. Categories having maximal percentage below 1% are omitted. There were only five categories noticeably accessed by the benchmark apps. Network information was dominant, constantly ranked first in any benchmark app. Additionally, the average percentage of all source calls that retrieved network information was as high as 98%. Network information was previously noted as widely accessed in Android apps [12], yet such overwhelming dominance of this category has not been reported. System settings, calender, location and account related data were also among the most commonly used categories of accessed data [11], [12], [40].

A similar breakdown of sinks over six significant categories is summarized in Table IV. 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 data via external storage. Similarly to account-setting operations, API calls for system settings can lead to data leakage as well. Lastly, network, file, and message-service operations are capable of leaking data through network connections or file-system I/Os 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 [11], [21].

In all, 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 involved. The generally small standard deviations corroborated the consistency of the ranking across the apps. This finding suggests that one approach to optimizing security defense solutions based on information flow analysis (e.g., [3], [9]) could be to prioritize the search for vulnerable flows in those that involve 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, drawing on the results from a dynamic analysis, the long lists of predefined sources and sinks used by static analyses may be prioritized 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 an unsafe but rapid solution—both previous studies and our experience suggest that cutting the lists significantly may lead to substantial analysis performance gains [3], [9].

3) Summary and Discussion: We found that (1) sensitive information accesses and critical operations are commonly exercised during Android app executions, though they are not heavily invoked (accounting for on average less than 15% of all calls even with respect to the highly-comprehensive lists of predefined sources and sinks), and (2) the target information of sources and target operations of sinks are both in narrow scopes: the vast majority (stably ranked top, and over 90% of all) sources focus on accessing network information, while the constantly most accessed (ranked among the top, and more than 80% in total out of all) sinks focus on operations related to account setting and logging.

In light of these findings, Android 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. It is of particular interest to investigate whether this pattern remains over time in the context of the evolving Android ecosystem through a longitudinal study, part of our future work.

D. Inter-App Benchmark Suite

Several Android app benchmark suites have been shared by researchers [3], [9], [18] yet they all targeted 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 an 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 80 potentially communicating app pairs (based on static ICC resolution and matching) 62 have ICCs between the pair that have been exercised readily by random Monkey inputs. Of these pairs, 34 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 IV), where we have hosted corresponding APKs for free, handy downloads too.

E. 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 threat comes from the possibility of missing or incorrectly placing profiling probes (for the instrumentation) due to code obfuscation, which has been a significant block for static analysis of Android apps [1], [24]. However, our manual inspection of Soot Jimple [35] 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 popular apps and picked each app from that pool randomly. Our benchmark suite is reasonably large for a systematic *dynamic* study that requires long profiling time. In addition, our study targeted relatively new apps which were built on Android SDK 4.4 or above. Thus, our results may not generalize to older apps. However, with the Android app market migrating to newer platforms [41], 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.

Like any other dynamic analyses, our empirical results are subject to the coverage of the test inputs used—some behaviors of the benchmarks might not have been exercised. To reduce this threat, we used the tool applicable to us that gives the highest coverage among peer tools [25] (and only 5% lower than the highest reported so far which was achieved by manual inputs in a much smaller-scale study [24]). Moreover, we carefully chose benchmarks for which the randomly generated inputs did achieve a reasonable coverage (55% at least and 70% on average)—our toolkit and methodology are not limited by the coverage, but we aimed at results more representative of app behaviors by using inputs of higher coverage. Nonetheless, the presented conclusions and insights are limited to the chosen apps and covered app behaviors—the coverage threshold applied during benchmark selection potentially affected the representativeness of the apps we chose. 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 marginal variances between the repetitions.

The main threat to conclusion validity concerns the distribution of underlying data points. Thus we reported standard deviations of means for metrics with which we did not show the entire distribution. As none of the benchmarks were identified as malicious by VirusTotal [42], our results may not generalize to malware. Our results and observations regarding callback and source/sink categorizations are subject to the validity of the corresponding predefined lists (e.g., callback interfaces and source/sink APIs) (see Section IV). We have used the best such resources we are aware of to reduce this additional threat.

It is important to note that the numbers we reported in our results are not intended to serve as absolute and exact metric values that precisely quantify Android app characteristics. With different benchmarks and test inputs, those numbers are expected to shift. However, comparing these results to those from our previous, pilot study on different benchmarks reveals that the deviations are mostly small. More importantly, the major observations remain almost the same. Nonetheless, rather than making strong claims about the numbers in absolute terms, we emphasize on the general trends (e.g., overwhelming dominance of calls to SDK code and callbacks to *Activity* components), contrasts (e.g., more external-explicit ICCs in inter-app executions versus single-app ones), and distributions (e.g., the majority of sources accessing network information versus other categories of sources executed) that constitute an overall understanding of app behaviors. The numbers should be taken as estimates that assist with the understanding.

VII. Related Work

A. Dynamic Analysis for Understanding

Various dynamic analysis techniques have been employed for program understanding in general [43], of which the most relevant to us is execution trace analysis (e.g., [44], [45]). Yet, many existing works of this kind concerned techniques aiming at effectively exploring the traces themselves, such as trace reduction [46] and visualization [47]; and others serve different purposes other than directly for understanding the behavior of programs, including evolution tracking [48] and architecture extraction [49]. 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 behavior 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 [45], app profile generation [17], and access control policy extraction [50]. Yet, their main goal was to serve individual apps thus different from ours of characterizing Android application programming in general. 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.

B. Android Security Analysis

There has been a growing body of research on securing Android apps against a wide range of security issues [1]. Among the rich set of proposed solutions [2], modeling the Android SDK and other libraries [3], [9], approximating the Android runtime environment [4], [5], and analyzing lifecycle callbacks and event handlers [9], [10] are the main underlying techniques for a variety of *specific* vulnerability and malware defense approaches [2]. Examples of such specific approaches include information flow analysis [21], [51] in general and taint analysis [9], [52] 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. 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 could help with the design of those techniques. Also, different from many of them that are purely static analyses, our work is dynamic. 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 [29].

C. Characterization of Android Apps

Empirical works targeting Android also have emerged lately, covering a broad scope of topics ranging from resource usage [12], battery consumption [15] permission management [53], code reuse [14] to ICC robustness [32], SDK stability [16] and user perception of security and privacy [40]. In contrast, our work aims at the general characterization of Android applications from the point of view of language constructs and run-time behaviors.

For categorizing information accessed by sensitive and critical API calls, we utilized predefined sources and sinks based on those used in [37], where frequencies of source and sink usage in malware samples were studied. Several other works on Android app characterization also targeted Android malware [18], [19], [54]. These studies either utilized static analysis [37], [54] or relied on manual investigation [18]. In contrast, we focused on understanding the run-time behaviors of benign Android apps via dynamic analysis, which potentially complements those previous studies.

VIII. Conclusion

We presented the first systematic dynamic study of Android programming that targets a general understanding of application behaviors and their implications in Android. To that end, we traced method calls and ICC Intents from one-hour continuous executions of 125 popular apps randomly selected from Google Play and 80 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 for future 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 (mostly for *Activities*) or event handlers (mostly for user interactions), (3) most executed ICCs do not carry any data payloads and inter-app ICCs pass data mainly via bundles instead of URIs, and (4) sensitive and critical APIs mostly focus on only a couple kinds of information and operations. In addition, we offered insights into the implications of our empirical findings that potentially inform future code analysis and security defense of Android apps towards better cost-effectiveness tradeoffs.

References

[1] 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," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[2] D. J. Tan, T.-W. Chua, V. L. Thing *et al.*, "Securing Android: a survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–45, 2015.

[3] 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 *Proceedings of ACM Conference on Computer and Communications Security*, 2014, pp. 1329–1341.

[4] M. Gordon, D. Kim, J. Perkins, L. Gilhamy, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proceedings of Network and Distributed System Security Symposium*, 2015.

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

[6] 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 *Proceedings of USENIX Security Symposium*, 2013, pp. 543–558.

[7] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2015, pp. 77–88.

[8] 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." *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[9] 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 *Proceedings of ACM Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.

[10] 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*, 2015, pp. 89–99.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proceedings of USENIX Security Symposium*, 2011, pp. 21–21.

[12] 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 ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015, pp. 1–11.

[13] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding Android fragmentation with topic analysis of vendor-specific bugs," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2012, pp. 83–92.

[14] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the Android market," in *Proceedings of IEEE International Conference on Program Comprehension*, 2012, pp. 113–122.

[15] D. Ferreira, A. K. Dey, and V. Kostakos, "Understanding human-smartphone concerns: a study of battery life," in *Pervasive Computing*, 2011, pp. 19–33.

[16] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *Proceedings of IEEE International Conference on Software Maintenance*, 2013, pp. 70–79.

[17] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: multi-layer profiling of Android applications," in *Proceedings of ACM International Conference on Mobile Computing and Networking*, 2012, pp. 137–148.

[18] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proceedings of IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[19] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 apps later: A view on current Android malware behaviors," in *Proceedings of International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014, pp. 3–17.

[20] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE security & privacy*, no. 1, pp. 50–57, 2009.

[21] 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 *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 393–407.

[22] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239–252.

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

[24] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proceedings of joint European Software Engineering Conference and ACM International Symposium on the Foundations of Software Engineering*, 2013, pp. 224–234.

[25] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 429–440.

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

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

[28] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of Annual Computer Security Applications Conference*, 2012, pp. 51–60.

[29] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in Android," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, 2015, pp. 118–128.

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

[31] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[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 *Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012, pp. 1–12.

[33] H. Cai and R. Santelices, "A comprehensive study of the predictive accuracy of dynamic change-impact analysis," *Journal of Systems and Software*, vol. 103, pp. 248–265, 2015.

[34] ——, "TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging," in *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 489–493.

[35] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a Java bytecode optimization framework," in *Cetus Users and Compiler Infrastructure Workshop*, 2011, pp. 1–11.

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

[37] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks." in *Proceedings of Network and Distributed System Security Symposium*, 2014.

[38] H. Cai and B. Ryder, "DroidFax: A toolkit for systematic characterization of Android applications," in *Proceedings of International Conference on Software Maintenance and Evolution*, 2017.

[39] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications on Android," in *Trust and Trustworthy Computing*, 2011, pp. 93–107.

[40] 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 ACM workshop on Security and privacy in smartphones and mobile devices*, 2012, pp. 33–44.

[41] Google, "Android Developer Dashboard," http://developer.android.com/about/dashboards/index.html, 2016, accessed online 09/20/2016.

[42] "VirusTotal," https://www.virustotal.com/.

[43] 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*, vol. 35, no. 5, pp. 684–702, 2009.

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

[45] 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 *Proceedings of Asia Joint Conference on Information Security*, 2012, pp. 62–69.

[46] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *Proceedings of IEEE International Conference on Program Comprehension*, 2006, pp. 181–190.

[47] 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*, vol. 81, no. 12, pp. 2252–2268, 2008.

[48] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind, "System evolution tracking through execution trace analysis," in *Proceedings of IEEE International Workshop on Program Comprehension*, 2005, pp. 237–246.

[49] 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*, vol. 80, no. 4, pp. 474–492, 2007.

[50] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of ACM Conference on Computer and Communications Security*, 2011, pp. 627–638.

[51] 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 *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2014, pp. 515–526.

[52] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of Network and Distributed System Security Symposium*, 2005.

[53] 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 Symposium on Usable Privacy and Security*, 2012, pp. 1–14.

[54] 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 *Proceedings of European Symposium on Research in Computer Security*, 2014, pp. 163–182.