A Longitudinal Study of Application Structure and Behaviors in Android

Haipeng Cai and Barbara Ryder

Abstract—With the rise of the mobile computing market, Android has received tremendous attention from both academia and industry. Application programming in Android is known to have unique characteristics, and Android apps be particularly vulnerable to various security attacks. In response, numerous solutions for particular security issues have been proposed. However, there is little broad understanding about Android app code structure and behaviors along with their implications for app analysis and security defense, especially in an evolutionary perspective. To mitigate this gap, we present a longitudinal characterization study of Android apps to systematically investigate how they are built and execute over time. Through lightweight static analysis and method-level tracing, we examined the code and execution of 17,664 apps sampled from the apps developed in each of eight past years, with respect to metrics in three complementary dimensions. Our study revealed that (1) apps functionalities heavily rely on the Android framework/SDK, and the reliance continues to grow, (2) *Activity* components constantly dominated over other types of components and were responsible for the invocation of most lifecycle callbacks, (3) event-handling callbacks consistently focused more on user-interface events than system events, (4) the overall use of callbacks has been slowly diminishing over time, (5) the majority of exercised inter-component communications (ICCs) did not carry any data payloads, and (6) sensitive data sources and sinks targeted only one/two dominant categories of information or operations, and the ranking of source/sink categories remained quite stable throughout the eight years. We discuss the implications of our empirical findings for cost-effective app analysis and security defense for Android, and make cost-effectiveness improvement recommendations accordingly.

Index Terms—Android, code structure, app behavior, longitudinal study, evolution, app analysis, security, ICC.

1 Introduction

The Android platform and its user applications (referred to as *apps*) have been dominating various mobile computing platforms, including smartphones, tablets, and other consumer electronics [1], [2]. Android developers are increasingly creating apps that cover a growing range of application domains. 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 researchers and tool developers to *understand* the particular software ecosystem of Android for developing cost-effective solutions to assuring the quality of Android apps.

Android apps have been primarily developed in two Java-based (JVM) languages, the canonical Java and Kotlin [3] (A Java-like language)—both are the current official development languages in Android. Yet these apps are different from traditional Java programs in how they are coded and executed. Android apps are *supposed to* rely on the Android SDK and various third-party libraries to realize their functionalities, according to existing (static) characterizations [4], [5]. In fact, many of the distinct characteristics of Android apps have led to unique challenges in developing sound and effective code-based app analyses [4], [6]. These challenges have resulted in specialization and customization, for the sake of Android apps, of analysis algorithms that were originally devised for traditional object-oriented programs.

- *Haipeng Cai is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA. E-mail: haipeng.cai@wsu.edu*
- *Barbara Ryder is with the Department of Computer Science, Virginia Tech, Blacksburg, VA. E-mail: ryder@cs.vt.edu*

Manuscript received April 1, 2018; revised August 26, 2015.

Specifically, the framework-based nature of Android apps requires substantial modeling of the platform and runtime for static analyses [4], [5], [7] to achieve reasonable accuracy. Implicit invocation between app components via a mechanism called inter-component communication (ICC) requires special treatments (e.g., ICC resolution [8], [9]) for a soundy [10] whole-program analysis. In addition, the event-driven paradigm in Android programming accounts for many challenges in app security analyses, such as determining component lifecycles [4], [6], [7] and computing callback control flows [6], [11].

Existing research on Android apps has been mainly aimed at *security* [12]. Further, most existing solutions targeted specific security issues, with merely a few offering a broader view of application security related characteristics in general [13], [14]. Intuitively, it is important to dissect app behaviors that commonly underlie varied security issues, so as to develop more capable and fundamental defense solutions that work across different kinds of those issues. Moreover, while a critical quality factor, security is not the only aspect of the holistic quality profile of apps. Knowledge about the underlying app behaviors is also essential for developing cost-effective app quality assurance solutions with respect to quality aspects other than security.

Studies do exist which aim to characterize Android apps beyond the security aspect, yet current studies are not sufficient in multiple ways. First, most of the prior work in this area (e.g., [15], [16], [17], [18], [19], [20], [21]) exclusively targeted *static* characterizations by examining the source code rather than the run-time behaviors of the apps. While they provide useful insights into app behaviors, these studies only offer a relatively rough approximation of those behaviors due to their overly conservative nature (i.e., considering all possible app executions). Dynamic characterizations would be necessary to complement

these studies by precisely profiling how apps actually behave at runtime. Second, existing *dynamic* characterization studies for Android either only address individual apps (e.g., [22]) or look at external exhibitions of apps behaviors (e.g., resource usage [14] and battery consumption [17]). Yet these dynamic studies do not offer a *code-based* (i.e., from the perspectives of app programming and code constructs) behavioral understanding of apps at large. Since it is the code of an app that holds the primary and original control of the app's behaviors, characterizing code-based (static and dynamic) app traits is a key to gaining a deep, fine-grained understanding of app behaviors that would commonly explain their diverse exhibitions. We refer to a code-based behavioral characterization of an app population as a *general* characterization. Third, other peer studies focus on malware (e.g., [23], [24], [25]) only or lack an evolutionary lens [26], thus they do not provide behavioral understandings about benign apps and the evolution of their behaviors. In all, there is a lack of *general characterizations of code-based behaviors with an evolutionary lens for Android apps* as a peculiar software domain.

To fill this gap, we conducted large-scale longitudinal studies of user applications in Android. We sampled a total of 17,664 apps developed throughout the past eight years (2010–2017) from varied sources, exercised each app with inputs that cover at least 60% of its code (74.25% on average), and gathered over 400GB execution traces of ordinary method calls and ICCs. From their code and executions that reasonably represent how they are developed and used, respectively, we characterize these apps, both statically and dynamically, while from both structural and behavioral perspectives.

In particular, we stress metrics relevant to notable challenges to app analysis (as mentioned earlier) including the interaction between user code and libraries, distribution of app components and ICCs, classification of callbacks, and categorization of security-sensitive data access. Accordingly, we defined characterization metrics that fall in three orthogonal dimensions, capturing the code and execution features of Android apps in multiple perspectives: *Structure* dimension, measuring the structure of app code and executions, *Communication* dimension, measuring component-level functionality invocation and data transmission, and *Sensitive Access* dimension, measuring the security aspect of apps in terms of their access to sensitive data. In each dimension, we further define complementary subcategories of measures. As a result, 122 metrics were derived and used in our characterization.

These multi-perspective metrics and corresponding measurement results constitute a *systematic* characterization of Android apps, which offers new understandings of app behaviors on the predominant mobile platform. Our study also demonstrates a practical methodology for obtaining such understandings of framework-based software applications in general. For the longer term, our results will benefit customization and/or optimization of future (code-based static/dynamic) Android app analyses, and will inform better design of techniques and tools for more cost-effectively securing the Android ecosystem. In particular, we apply an *evolutionary* lens to the characteristics of these apps to investigate how the *static* code traits and *run-time* behaviors of Android apps change *over time*. This lens reveals the *dynamics* of code structure and behavior traits in Android. Meanwhile, we put the dynamic behaviors of apps in context of their corresponding static characteristics to examine how apps *actually* behave versus how they are coded (i.e., how they are *attempted/supposed* to behave).

Our study has led to several noteworthy findings. First, the study revealed that the studied apps are generally featured with heavy reliance on the Android platform in terms of both API callsites referred in the code and API calls invoked at runtime. Beyond similar prior studies which all suggested the significant involvement of the platform/SDK in apps' functionalities, our study highlighted the *dominating* proportion of the platform functionalities in those of apps while also revealing similar dominance in apps' runtime behaviors. Also, the dominance kept growing over the studied period of eight years, as reflected by the continuously rising percentage of calls occurred within the SDK among all method calls in apps' code and executions. In contrary, we found that user code in Android apps kept shrinking in terms of the percentage of all callsites referred in user code and method calls initiated from that code layer, whether the calls being within user code or targeting the SDK. Regarding the involvement of third-party libraries in apps' code structure and execution composition, calls to these libraries tended to be rarely initiated from either the platform or user code, yet followed by noticeable and gradually increasing portions of calls within libraries.

Second, our study showed that only a small portion ($<$2%) of callsites and runtime method calls in the benchmarks targeted callback methods, albeit the presence of exercised callbacks in app executions was noticeably greater than that of callback callsites in app code. Among the callbacks used in the sample apps, either in code or at runtime, more aimed at lifecycle management of various types of app components by the Android platform than handling varied kinds of events (e.g., events on user interaction, system status, hardware management, etc.). In particular, consistently over time, the most dominating portion of lifecycle callbacks were used for managing the lifecycle of *Activity* components while the most dominating portion of event-handling callbacks were invoked for handling events associated with *Views* (as a type of user interface) and *System status*. Yet overall the use of callbacks in the apps dropped considerably, while the category ranking of used callbacks remained almost the same, over the studied years.

Third, our results indicated ICCs were a relatively minor part of the runtime behaviors of apps—among all method calls occurred during app executions, (API calls for) ICCs accounted for a tiny proportion ($<$1%) only. Among the exercised ICCs, *Activity* components (the type of components hosting GUIs) were the dominating (85%) initiators and receivers of ICCs, consistent with the dominance of this type of components over all other types of components that were exercised during app executions. While this is not surprising given the fact that Android apps are generally rich in GUIs, such an overwhelming degree of dominance has not been reported before. Historically, however, we found that this dominance has been declining over time. We also found that newer apps tended to use *Content Providers* (app components for data storage and management) with increasing frequency at runtime yet with less and less callsites in their code. In addition, between components within the same apps, the targets of exercised ICCs were rarely implicitly matched but almost always explicitly specified, while between components across apps, the targets were much more likely to be implicitly resolved. The majority, and over time increasing proportions, of these exercised ICCs carried no data at all, or they did so preferably in the *extras* field of the ICC messaging objects.

Fourth, we demonstrated through this study the extensive access to sensitive data and operations in the studied apps, albeit through a relatively proportion of callsites for such access. Over time, proportions of APIs for accessing sensitive data among all method calls in apps remained generally stable, yet those of APIs for accessing sensitive operations (which are potentially able to send the data out of the apps) have been declining. Among the categories of accessed sensitive data, network information was the most common category, while the most common category of accessed sensitive operations were related to account setting. Through an evolutionary lens, we found that both the top categories of sensitive access and the overall ranking of major categories have not changed much across the eight-year span studied. Also, with respect to the category ranking, the access in app code is overall proportional to that during app executions in terms of the percentage of the underlying APIs.

In sum, this paper makes the following contributions:

- We systematically characterized both the code traits and run-time behaviors of 17,664 apps in terms of a diverse set of metrics that lie in three complementary dimensions: functionality composition (*Structure*), component communication (*Communication*), and access to sensitive data and operations (*Sensitive Access*).
- We conducted a longitudinal study of apps developed in the past eight years (2010–2017), which revealed how Android apps (as a population) evolved over time with respect to the diverse metrics.
- We discussed the implications of our empirical findings and made actionable recommendations to both researchers and tool developers for improving (code-based static and dynamic) app analysis in general and security defense techniques in particular, for enhanced cost-effectiveness. We also examined how the quality of test inputs affected our conclusions.
- We released our study utilities and dataset to facilitate both reuse and reproduction, which is available online [27].

2 Background

Android is now the most popular operating system (OS) running on various 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* [6].

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 [28]. The SDK includes APIs for ICC by which components communicate primarily via messaging 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, and contacts [13], [28]. Exposure/leak of the sensitive information is a main cause of varied security threats in an app. A primary approach to the defense against these threats is to detect the existence of feasible program paths, called *sensitive information flow*, between predefined information *sources* and information *sinks* [6], [29]. In Android, *sources* are the APIs through which apps access sensitive information (i.e., *sensitive* APIs). Therefore, as per this definition, we consider *sensitive* the data retrieved by any source API. 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 *sinks*). We consider *sensitive* the operation performed the invocation of any sink API.

3 Methodology

To understand the *dynamic* features of applications in Android, we traced all ordinary method calls and Intent ICCs. 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. To contextualize these dynamic features, we further characterize the underlying *static* code features as well. Next, we elaborate on the design of our empirical study—benchmark apps, inputs used for the dynamic analysis, metrics calculated, and study process.

3.1 Benchmarks and Test Inputs

TABLE 1: Numbers of benchmarks of per different year and in total used in our study

Year	2010	2011	2012	2013	2014	2015	2016	2017	Total
#apps	2,235	2,222	2,208	2,107	2,288	2,183	2,174	2,247	17,664

For our evolutionary characterization, we collected sample apps that were developed in each of the eight past years between 2010–2017. The apps from 2017 were downloaded from Google Play, while all other apps were selected from the AndroZoo database [30] (which crawls apps from a large variety of sources). The reason that we did not also download year-2017 apps from AndroZoo was that during our study period there were not sufficient samples in there.

Benchmark selection process and criterion. For each year, we started with a pool of 500 apps that were randomly picked from the respective source, and then applied the following three selection criteria to choose the benchmarks eventually used in our study; we then repeated this selection process with another 500 downloaded samples, until we had at least 2,000 samples per year. First, the app is not a repackaged version of any other app that has been already selected. Second, the app can be successfully processed by our study pipeline (as described in Section 4). Third, exercising the app with the inputs generated by Sapienz [31] for 10 minutes can cover at least 60% user code of the app. We tried consistently 2,500 apps in total per year—for each 500, we used all that met the three criteria. We chose 60% as the minimal coverage because

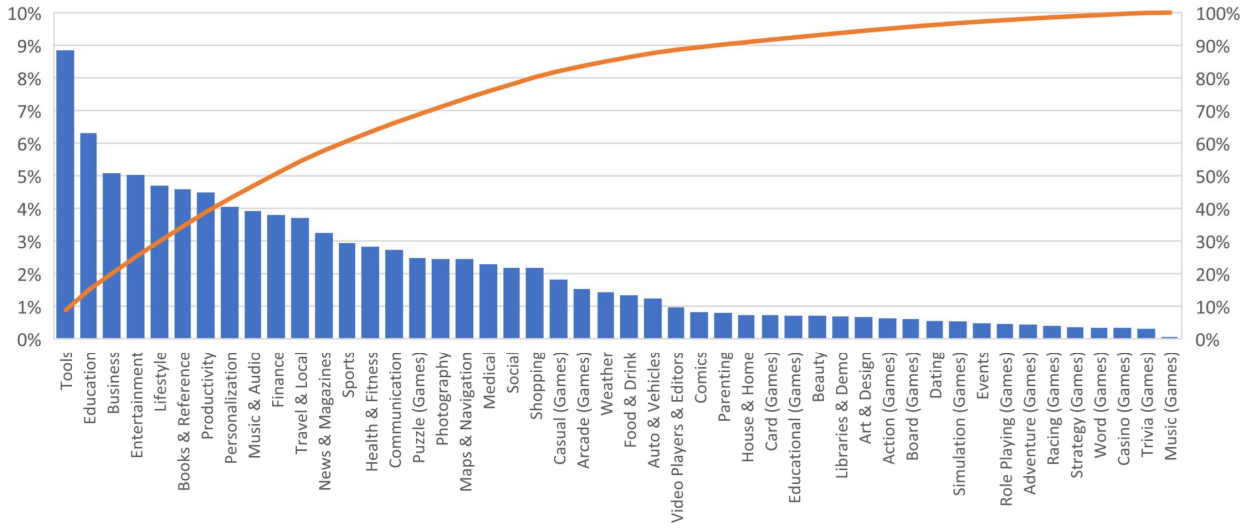

Fig. 1: A pareto chart showing the percentage distribution of our benchmarks over 48 distinct functionality categories.

(1) it enforces the *majority* of the code lines to be covered and (2) previous relevant studies showed that the coverage of human inputs is up to this level [32]. Table 1 lists the number of samples from each year that we actually chose as per the three selection criteria, totaling 17,664 apps.[1] This scale of our study is greater than or at least comparable to that of most prior Android app characterization studies (e.g., 27 apps [22], 125 [26], 1,260 [23], 3,568 [34], 4,323 [16], 12,466 [25]). Our study scale was mainly limited by the substantial run-time profiling cost (i.e., 10 minutes per app).

The selected apps covered 48 out of the total 49 distinct functionality categories (available by 2019) of apps [35] currently listed on Google Play (e.g., `Sports` and `Medical`). The only one that was not covered was `Wear OS by Google`, which is the category of apps developed for Android wearable devices—our study was not intended to cover these specialized apps. All the 17 game categories were represented by our benchmark suite. The percentage of our benchmarks in each of these 48 categories ranged from 8.85% (`Tools`) to 0.06% (`Music` games). Figure 1 depicts with a pareto chart the detailed percentage distribution of our 17,664 samples over these functionality categories. Separately for each year, our yearly benchmark set also covered all categories available on Google Play in that year except for those specialized for wearable devices—the number of app categories on the store has been changing (growing) over the years.

The reason we filtered out repackaged apps was because these apps would have substantial overlapping in terms of functionality and construction with other apps. Including these apps would confound our study results (e.g., two apps, with one app being a repackaged version of the other, would be counted as two distinct apps although they mainly only represent one unique app's behavior; also, this would falsely promote the representativeness of that kind of behavior). The reason we attended this issue was because repackaging has been a common practice for quick production of apps in Android. To detection repackaged apps, we utilized a state-of-the-art tool [36] dedicated for this purpose that is publicly available and highly efficient. It identifies repackaging relationship between two apps based on their APK content similarity. By default, the similarity threshold for recognizing the relationship is 0.7—a similarity score of 0 indicating entirely disparate apps while a score of 1 indicating two identical apps. To be conservative, we dismissed an app during the benchmark selection if it was detected to be similar to any other already-selected app with a score$\geq$ 0.5. Eventually, among the chosen apps, 94.5% of the app pairs across years and 84.9% of those of the same year had a similarity score of 0.

During the selection process, there were several reasons why a candidate app cannot be processed thus was discarded: (1) the app could not be unzipped (e.g., corrupted apps); (2) the app could not be statically-analyzed and instrumented (e.g., apps with damaged bytecode or missing resource files); and (3) the instrumented app could not be executed for successful tracing. The failed executions were mainly because of compatibility issues [37], [38] (e.g., an app relies on outdated framework APIs that are no longer supported on the our platform of API level 22). Another cause was that there were apps that self-check their integrity (e.g., signature) which was broken during the instrumentation. In our study, there were on average 4.2% (ranging from 2.1% to 6.6%) of candidate apps across the studied years that were discarded due to one or more of these three reasons.

Input generation. Previous dynamic studies of Android apps, using much smaller benchmark suites, mostly resorted to manual (expert) inputs [29], [39], [40], [41], [42], because the coverage of automatically generated Android inputs was considered 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 is an unscalable strategy for dynamic analysis, especially at our scale. Second, state-of-the-art automatic Android input generators can achieve practically as high code coverage as human experts [32] and they are scalable.

With these considerations, we chose Sapienz [31] because of two reasons. First, it is a state-of-the-art automated input generator that works for Android, having achieved a superior level of average code coverage than many peer tools including

1. Since our study focuses on benign apps in Android, not being malware is an implicit criterion: for each candidate benchmark, we used VirusTotal [33] to scan it; as long as any of the VirusTotal tools found it as malware of any type, we discarded it.

DynoDroid [32], PUMA [43], and Monkey (Android's standard/built-in UI automator) [44]. Second, it has been being maintained and used as an industry-strength tool for mobile app testing at Facebook [45]. We applied the 60% coverage criterion to make sure that the majority of user code in the studied apps is exercised, so that our dynamic features reflect a substantial portion of apps behaviors. The eventual line coverage of our per-app traces ranged from 60% to 100% (mean=74.25%, median=71.4%, standard deviation=10.73%)—if we include the apps discarded due to falling short of the minimal (60%) coverage, the range remains the same with mean=71.2%, median=70.1%, and standard deviation=11.1%.

3.2 Metrics and Measures

TABLE 2: Summary of characterization metrics and measures

Metric Dimension	Measure	Description
Structure	Class distribution	percentage distribution of all method calls over different code layers (in class and method granularity levels), app component types, and callback categories
	Method distribution	
	Inter-layer interaction	
	Component distribution	
	Callback invocation	
	Callback categorization	
Communication	ICC invocation	percentage of ICC calls and their percentage distribution over different ICC types and data transfer means
	Connection type	
	Data payload	
Sensitive Access	Source usage	percentage of source/sink calls and their percentage distribution over different source/sink categories
	Sink usage	
	Source categorization	
	Sink categorization	

We characterize run-time behaviors along with static characteristics of Android apps via 122 metrics in *three complementary categories/dimensions* each consisting of several supporting measures. Thus, our characterization is *systematic* relative to relevant prior works in terms of the study scope. Table 2 gives a brief summary of these metrics, as elaborated and justified below—we refer to the focus and scope of each metric/measure, rather than its name, for brevity.

- ***Structure metrics.*** Metrics of this dimension characterize an app with respect to its functionality composition and the structure of its execution, consisting of six measures.
  - Class distribution (i.e., percentage distribution of classes) and method distribution (i.e., percentage distribution of methods) measure, at class and method level respectively, how the exercised functionalities of an app are distributed over three layers of the app code: user code (*UserCode*), third-party libraries (*3rdLib*), and the SDK (*SDK*).
  - The communication among these layers is captured at method level through the inter-layer interaction (i.e., calling relationships across code layers) measure. Component distribution (i.e., percentage distribution of components) also measures the functionality distribution, but at component level.
  - We capture the usage of callbacks via the callback invocation (i.e., invocation frequency of callbacks) measure. With another, complementary measure, callback categorization (i.e., callback invocation percentage distribution over callback categories), we look further into that usage with respect to different categories. For lifecycle methods, the categories are the five top-level enclosing classes: *Activity, Service, BroadcastReceiver, ContentProvider, Application* (see Section 2). for event handlers, the categories are five types of user interfaces (UIs) that are associated with the handlers: *App bar*, *Media control*, *View*, *Widget*, *Dialog*, as well as five types of system events handled: *App management*, *System status*, *Location status*, *Hardware management*, and *Network management* [26].
- ***Communication metrics.*** The interaction between components is captured in this dimension by three measures.
  - The ICC invocation (i.e., invocation frequency of ICCs) measure captures the usage of ICCs.
  - The connection type (i.e., percentage distribution of ICCs over ICC types) measure quantifies how ICCs in the app are distributed over four types: the four pair-wise combinations of **scope** (i.e., *internal* and *external*) and **specificity** (i.e., *explicit* and *implicit*) of the inter-component linkage.
  - The data payload (i.e., ICC data transfer mechanism) measure captures the means by which data can be carried in ICCs—URI (the `data` field of Intent) or bundle (the `extras` field of Intent), or both—which informs the type of the data transferred between app components via ICCs hence is a key indicator of ICC-induced app behaviors.
- ***Sensitive Access metrics.*** We capture app characteristics relevant to sensitive data access (via sources and sinks) by four measures.
  - We examine the prevalence of sensitive access through the overall source usage (i.e., source call frequency) and sink usage (i.e., sink call frequency) measures.
  - The source categorization (i.e., source call percentage distribution over source categories) measure checks deeper into the semantics of exercised sources concerning the information they access. Similarly, the sink categorization (i.e., sink call percentage distribution over sink categories) measure checks deeper into the semantics of exercised sinks concerning the operation they access. Based on preliminary results in this regard [26], we consider five categories of information that sources mostly retrieve: *Account info*, *Calendar info*, *Location info*, *Network info*, and *System settings*, and six categories of operations sinks mostly perform: *Account setting*, *File operation*, *Logging*, *Network operation*, *SMS/MMS*, and *System setting*.

Two views: static and dynamic. As mentioned earlier, our study includes both static and dynamic characterizations. Accordingly, across the above three dimensions of metrics, we examine two views of each measure: *dynamic* and *static*. The dynamic view is associated with the full execution traces, while the static view is associated with the app code. The dynamic view captures the run-time behaviors of apps with call *frequency* while the static view considers specific callsites in the apps code. These two views are thus complementary, together conveying the code characteristics and run-time behaviors of Android apps.

Illustration. To illustrate, consider the *class distribution* and *method distribution* measures in the *Structure* dimension.

Suppose an app's code contains three classes `A`, `B`, `C`, defined in *UserCode*, *3rdLib*, and *SDK* while having 2, 3, 4 methods, respectively. Further suppose the app's execution trace has two calls `A::f→B::g` and `B::m→C::h`. Then, the *class distribution* is characterized by three numbers (1/3, 1/3, 1/3) in the static view and (1/4, 2/4, 1/4) in the dynamic view. The numbers for *method distribution* are (2/9, 3/9, 4/9) and (1/4, 2/4, 1/4) in the static and dynamic view, respectively.

3.3 Procedure

For each app, we characterize its static code traits through simple static analysis, which results in the 122 metric values in the *static* view for the app. To compute the *dynamic*-view values of these metrics for an app, we first collect the operational profile of the app in the form of a method-call and ICC-Intent trace. To that end, we instrumented each app for monitoring ICC Intents and all method calls. Next, we ran each instrumented app on a Samsung Galaxy S4 smartphone with the Android SDK 5.01 (API level 22), 2G RAM, and 4G SD storage.

To avoid possible side effects of inconsistent device settings, we started the installation and execution of each app in a clean environment of the device (with respect to built-in apps, user data, and system settings, etc.). For each individual app, test inputs generated by Sapienz were provided for 10 minutes of execution. We chose this execution-time length because (1) after 10 minutes, the code coverage gain seen by Sapienz was generally very small, and (2) given the scale of our study (tracing our entire benchmark suite for 10 minutes per app took over five months), we had to balance between the code coverage achieved and total time cost of the study.

With this procedure, our studies took over eight months, the majority of which was spent on collecting the app traces for the dynamic characterization. The total storage cost of the studies was over 400GB, mainly due to the space of execution traces.

4 Study Pipeline and Implementation

This section elaborates important details about the implementation of our study pipeline. We focus on the key aspects of the study toolkit that are essential for understanding and interpreting the characterization results presented next. Further engineering details can be found in [46].

Figure 2 depicts the workflow of our study with respect to one app, including its three inputs (the APK of the app, the predefined list of sources and sinks, and the predefined list of callback interfaces), and a single ultimate output (the study result report). There are three major steps that compute this output using the inputs, as detailed below.

4.1 Pre-processing

This step aims to compute the information needed for both the static and dynamic characterizations afterwards. For static characterization, the *static code analysis* module identifies the three code layers and all callbacks methods. Identifying the latter requires sophisticated static analysis, due to the event-driven nature of Android apps. In our study, the callback methods were iteratively found through a state-of-the-art callback control-flow analysis for Android as offered by the GATOR toolkit [47]. Our static metrics also require knowledge about the component type of each class (i.e., the top component class it inherits), for which we performed a class hierarchy analysis (CHA). The same CHA was also used for identifying the interfaces each class implements, which later supports the categorization of callbacks. Computing these kinds of *static app information* was enabled also by Flowdroid [6] for its various analysis utilities for Android apps. Both GATOR and Flowdroid use Soot [48] for lower-level bytecode analysis and manipulation.

For dynamic characterization, in addition to computing the static app information, we instrument the APK (specifically the Dalvik bytecode in it) of the given app at the Jimple intermediate representation (IR) provided by Soot. The instrumentation probes for method call profiling and ICC Intent tracing, and results in the *instrumented APK* to be executed in the next step. To instrument the bytecode in a given APK, we used our own home-made bytecode instrumentation and manipulation framework [46] developed based on the Soot framework—Soot itself provides basic bytecode instrumentation capabilities, based on which our framework provides more usable APIs through encapsulation and/or wrappers.

4.2 Profiling

The second step installs an instrumented app to the device (i.e., the Samsung Galaxy S4 smartphone). Once installation is completed, our toolkit launches the Sapienz input generator to feed the app with test inputs for 10 minutes. Our toolkit (including Sapienz) ran on a Ubuntu 16.04 desktop with a 16GB RAM and a 3.4GHZ i7 processor. The inputs generated by Sapienz were sent to the device through the Android Debugging Bridge (`adb` [49]), which is a toolkit as part of the Android SDK. When the app executes, the logging statements (invoking the Android Log APIs [50]) probed for each method call and (each Intent field of) each ICC will be invoked to produce the method-call and ICC Intent traces for the app. The logs (execution traces) were transimitted back to the desktop also via `adb` (using its `logcat` functionality [51]).

4.3 Characterization

As the final step, we characterized each app by computing static and dynamic metrics. Static metrics were computed using the results of the static code analysis, along with the lists of sources and sinks as well as the list of callback interface. The source/sink lists were immediately used for identifying sources/sinks in the code. In particular, we used the lists of sources and sinks produced by SuSi [52]—these lists have been widely used by other researchers. Moreover, we manually improved the training set of SuSi hence produced a more precise source/sink categorization, used for computing the corresponding metrics (e.g., percentage distribution of sources invoked in app code over the five source categories). 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 [6]. We did the categorization based on our understanding of each interface according to the official Android SDK documentation. Lifecycle callbacks were categorized using the same CHA as used in Step 1 (i.e., *Pre-processing*).

To compute the dynamic metrics, we analyzed the trace of each app by first building a *dynamic call graph*. Each node of the graph is the signature of an executed method, 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.,

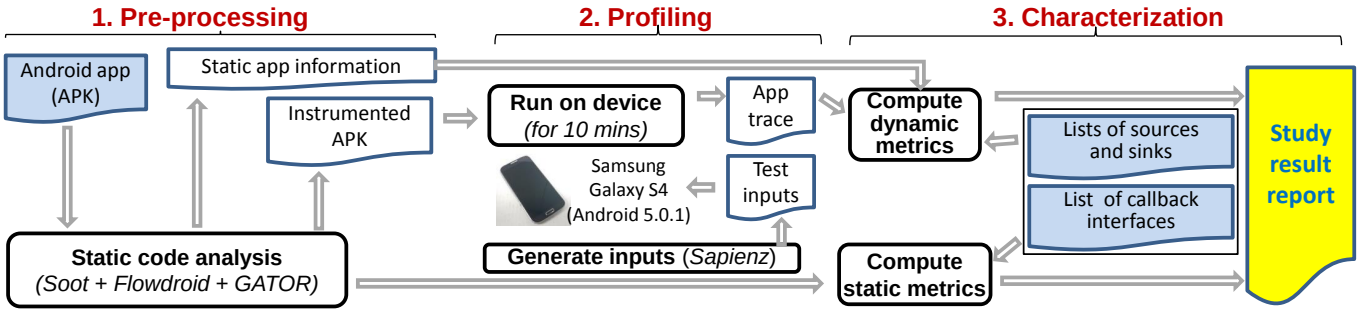

Fig. 2: The overall workflow of our characterization study, including the inputs and outputs.

`startActivity`) to the receiving API (e.g., `getIntent`) of that ICC, established according to the fields of the Intents at the sending and receiving sites. More specifically, for explicit ICCs, the linking is based on the explicit target specified in the Intent of the sending site; for implicit ICCs, we follow the standard approach of matching the *action*, *category*, and *data* fields of the Intents with respect to the Intent filters specified in the sending app's manifest file. Particularly for external ICCs, since we do not trace the external apps, we use virtual nodes to represent the external ICC API sites hence establish the ICC edge to add to the call graph. For an explicit external ICC, only one such virtual node is needed, while for an implicit sending Intent, one or more virtual external nodes may be needed, depending on how many external (built-in apps') components can be implicitly matched using the same matching rules used for internal implicit ICC linking—we first try to match internal Intents, and external ICC edges are established only when internal matching fails. Once the call graph construction is completed, this phase computes metric values in the dynamic view using the call graph and the static information computed in the first phase.

To facilitate reproduction and reuse, we released the open-source implementation of our study utilities as an open-source toolkit DROIDFAX [46], including a line coverage tracking tool directly working on an APK. Given a configured Android device (either an emulator or real device) and a set of apps, the automated study workflow produces both metrics values and their visualizations and tabulations. 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.

5 RESEARCH QUESTIONS

The primary goal of our study is to understand how applications in the Android ecosystem are coded and behave so as to inform of more cost-effective app analysis (static or dynamic) and security defense strategies. With respect to the metrics we define for the study, we seek to answer the following research questions. Each question is explored with an evolutionary perspective in mind.

RQ1: *How are code and executions of Android apps structured with respect to user code, SDK, and other libraries?*

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 execution of Android apps—previous works only *suggested* the *existence* of that nature through static analysis [5], [7]. RQ1 is answered using the first three measures the *class/method distribution* and *inter-layer interaction* measures of the general metrics.

RQ2: *How are lifecycle and event-handling callbacks used in Android apps?*

It is well known that callbacks, including lifecycle methods and event handlers, are widely *defined or registered* in Android app code [4], [6], [11]. This question additionally 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 *callback invocation* and *callback categorization* measures of the general metrics.

RQ3: *How are different types of components used and communicating in Android apps?*

ICC has been a major security attack surface in Android [8], [53], [54] as well as a feature of Android application programming that sets it apart from ordinary Java programming. Much prior research has targeted Android security concerning ICCs [8], [9], [53], [54], 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 are sensitive information and critical operations accessed in the code and executions of Android apps?*

Addressing the secure usage of sensitive information has been the focus of various prior works on app security, including taint analysis [4], [6], privilege escalation defense [7], [40], and data leakage detection [53], [55]. However, how often that usage is attempted in code and exercised at runtime, and which kinds of sensitive information/operations are mostly accessed, have not been studied. RQ4 explores these questions with *Security* metrics. We note that although our study does not immediately address specific security problems (e.g., data leaks), the security-relevant characteristics of apps we examine (e.g., sensitive data access through source/sink API calls) represent app behaviors that are commonly relevant to a range of those specific problems.

6 RESULTS

This section presents the results of our study with respect to the relevant research questions. We focus on explaining the results themselves here, and discuss their implications in Section 7.

For callback and source/sink categorization, we rank the categories for each app and report for each category the mean rank across all benchmarks and the standard deviation of the

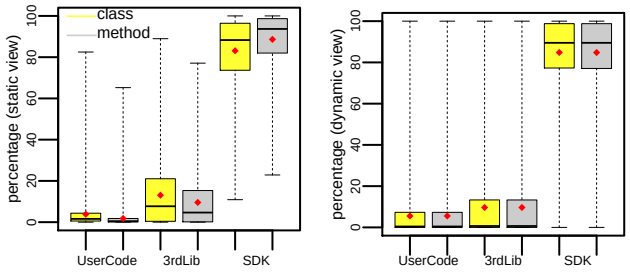

Fig. 3: Percentage distribution (y axis) of the three code layers in terms of callsites (left) and call instances (right) of all apps.

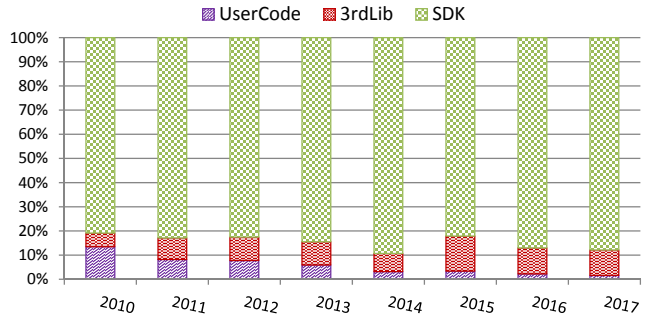

Fig. 4: The evolution of apps execution composition (y axis) in terms of the three code layers over years (x axis).

ranks. For each of the other metrics, consistently expressed as a percentage, we first calculated the percentage for each app separately. Then we report 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 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). In addition, we present the evolutionary pattern with respect to a metric based on the average metric values of apps from each year.

6.1 RQ1: Functional Structure of Code and Executions

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

6.1.1 Composition of Code and Execution

The composition of an app's code and execution is characterized through the percentage distribution of callsites (i.e., in the *static* view) and that of call instances (i.e., in the *dynamic* view) over the three code layers, respectively. Figure 3 shows the distribution of these layers in all the 17,664 apps, with each group of boxplots depicting both class and method granularity.

The *static view* reveals that consistently all the apps referred to library functionalities extensively, especially the SDK, for completing their tasks. On average, at both class and method levels, only about 5% or less of code in these apps was actually written by the developers (i.e., *UserCode* layer). In comparison, 14% of all classes and 10% of all methods were in various non-SDK libraries. The rest (over 80% of classes and methods) were defined in the Android SDK/framework (and invoked by the apps). Clearly, the results show that the SDK dominated the code of these apps, and suggest that an average Android app tended to be heavily dependent on a *diverse*[2] set of SDK APIs. A main reason for the dominance and dependence is that Android applications are framework-based while the framework provides rich functionalities that developers can easily reuse.

Counting all call instances, the *dynamic 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 behaviors of the SDK dominate in those of the apps. The observation that over 82% (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 5% of their calls target user code. These observations were plausibly due to the fact that the Android platform offers immediate support for apps to fulfill most of the common tasks targeted by the apps, thus developers do not have to write much of their own code.

Figure 4 depicts how the composition of apps execution evolved during 2010–2017, at the class level—the method-level results were similar. Each bar shows the average percentage of classes from each code layer over all benchmarks of a particular year. The (proportion) contrast of the three layers for each year was close to that of all benchmarks as shown in Figure 3: the majority (at least 80%) of functionalities were realized through the Android framework, followed by other libraries, while user code was constantly the smallest portion (at most 13%). Our results show that the portion of SDK and third-party libraries increased, while that of user code decreased, over the time period. These trends may be explained by the growing capabilities of the Android framework during its evolution and the rising number and diversity of third-party libraries available.

The evolution of code composition (i.e., in the static view, not depicted here) was similar to Figure 4, except it revealed (1) *decrease* in the apps's use of SDK, (2) much slower decrease in the portion of user code, and (3) slightly faster growth in the use of third-party libraries, over the past eight years. Put in contrast to the dynamic view (Figure 4), our results imply that, on average in newer apps, each SDK callsite was executed more frequently and each *UserCode* callsite was executed less frequently.

> **Finding 1**: The Android apps relied heavily on the SDK, but little on user code, to realize their functionalities. Over time, the apps have been using slightly more third-party libraries, but *significantly less user-defined functionalities*, both in their code and executions; they also tended to *increasingly use SDK code*, although these apps contained fewer SDK callsites.

6.1.2 Inter-layer Code Interaction

Figure 5 shows the percentage of call instances in each inter-layer interaction category over the total call instances in all the benchmark executions. The data points are categorized by calling relationships, denoted in the format of *caller layer→callee layer*, among the three code layers. Noticeably, the majority (27.5%) of

2. Recall that the static view only counts unique methods invoked in app code, thus the SDK dominance (as measured in terms of method calls) in the static view suggested the diversity. According to our further analysis on inter-layer interaction as discussed next and shown in Figure 5, these diverse SDK APIs were invoked primarily *within* the SDK code layer with a few, and increasingly fewer (as per Figure 6), being called from the *UserCode* layer.

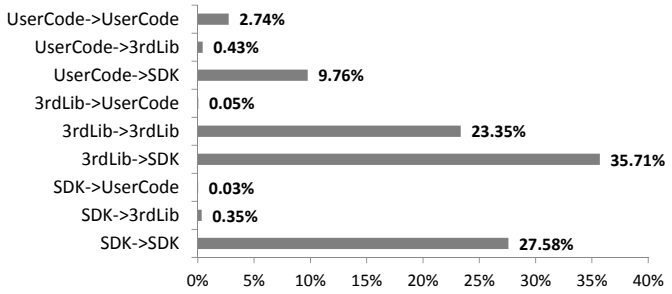

Fig. 5: Percentages of different categories of inter-layer call instances over all benchmark app executions.

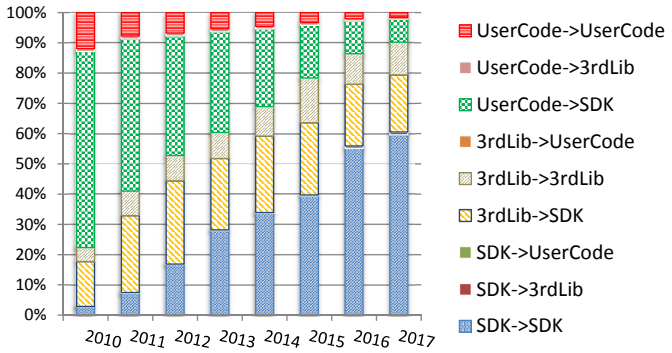

Fig. 6: The evolution of inter-layer interaction in terms of call instances (y axis) over the past eight years (x axis).

call instances happened *within* the same layers, dominated by SDK (41.5%). User functions were called very rarely by any callers (<3% in total), reconfirming our previous observation from Figure 3. The results reveal that the vast majority of calls to third-party library functions were from the same code layer.

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. As expected, calls (from all the three layers) to *SDK* dominated the nine categories of calling relationships (73.05% in total). These results 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.

The evolutionary pattern of run-time app behaviors with respect to inter-layer interaction is illustrated in Figure 6. Each bar shows the average percentage of call instances in each of the nine inter-layer call categories over all apps in a specific year. Overall, the most substantial categories among apps in each year were the same as those in the apps across all the eight years (Figure 5)—$3rdlib{\rightarrow}SDK$, $SDK{\rightarrow}SDK$, $3rdlib{\rightarrow}3rdlib$, $UserCode{\rightarrow}SDK$, and $UserCode{\rightarrow}UserCode$—albeit the rankings by percentage differ. In particular, the number of call instances within *UserCode* and those from *UserCode* to *SDK* have been monotonically decreasing in the eight-year span. It is probable that this phenomenon was caused by the fewer *UserCode* call instances during executions of newer apps (see Figure 4).

Calls within *SDK*, however, have seen steady growth over these years. This indicates that newer apps tend to have even more of their tasks completed by the Android framework alone (than by collaborating with user code or third-party libraries). Plausible

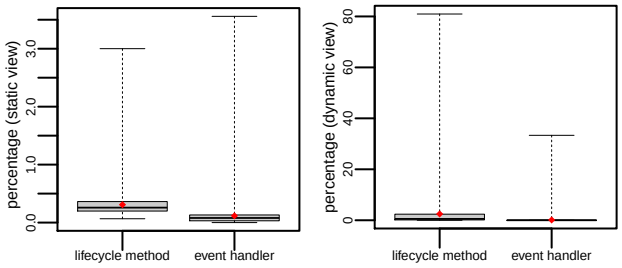

Fig. 7: Percentage distribution (y axis) of callbacks (x axis) over all callsites (left) and all call instances (right) in all benchmarks.

reasons behind this trend include the platform's capabilities being enriched over time and developers getting increasing familiarity with the SDK so that they know gradually better about how to reuse the framework's functionalities. Another possible reason is that the SDK had increasing encapsulation (i.e., on average each SDK method tended to do more), and/or encapsulation in the user code grew over time (e.g., developers encapsulate calls to the same SDK APIs in single *UserCode* methods/classes). In general, there also has been a slight increase in the calls within various libraries, while library calls targeting SDK did not change much.

Finding 2: The majority of calls in apps occurred *within* the SDK, and most of the rest targeted either SDK or other libraries. There were constantly very few calls targeting user code. Over the eight years, apps have been performing tasks increasingly via the *SDK alone* with decreasing portions of calls from user code.

6.2 RQ2: Usage of Callbacks

Due to their event-driven nature, Android apps typically contain callbacks. We specifically examine the overall extent of use and distribution of callbacks, both as invoked in code and as exercised during app executions, over the major callback categories (as described in Section 3.2).

6.2.1 Extent of Use

We first looked into the extent of callback usage (over all code layers) in the benchmark apps through the percentage distribution of callback invocations. As shown by Figure 7, on average no more than 0.25% of all callsites (in the *static view*) targeted either lifecycle callbacks or event handlers. Overall, there were more (albeit very slightly) lifecycle-method callsites than event-handler callsites defined in these apps. In a few apps, up to 3.5% of all callsites were for event handling and there were no apps containing more than 3% callsites targeting lifecycle methods.

The *dynamic view* indicates that callbacks were not invoked very frequently either. The average percentage of either type of callback invocations was under 2%, indicating that callbacks, while prevalently registered in Android apps [6], [11], tend to be called only quite lightly at runtime. For no more than a quarter of the apps, up to 80% of callback instances were for managing lifecycle of app components while 38% were for handling all other events. Yet in the majority (75%) of the apps, these numbers were never above the means. This observation is consistent with our results on inter-layer interaction of Figure 5, where we have seen relatively small numbers of calls invoking user code from the SDK or other libraries—such calls correspond to callbacks. In contrast, there were more lifecycle callbacks than event handlers exercised out of all method call instances in these

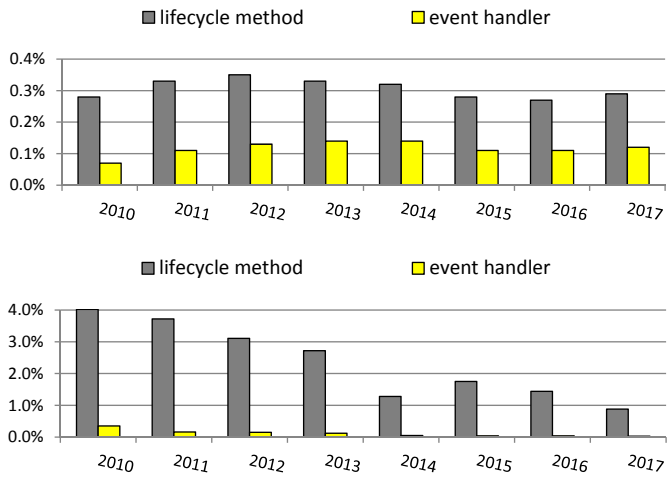

Fig. 8: The evolution of callback usage (y axis) in the static (top) and dynamic (bottom) views over years (x axis).

benchmarks, in accordance with the greater portion of lifecycle callbacks invoked in the apps' code (see the *static view*).

Examining the extent of callback usage in apps by year reveals the evolutionary pattern, as depicted in Figure 8. First of all, lifecycle methods were consistently used much more heavily than were event handlers, in apps from any of these years. This is consistent with the aggregate contrast of Figure 7. Across these years, the proportions of both kinds of callbacks defined in apps code remained generally steady. A possible reason is that, despite the evolution of the Android framework, the callback methods defined in the SDK did not change much. Yet the call frequency tended to gradually drop overall. In all, callback usage appeared to decrease over time within the studied span of years.

Finding 3: Among the very-small percentage of method calls that were callbacks, lifecycle methods were used more often than event handlers both in app code and execution. Callback usage in apps has been dropping significantly in recent years.

6.2.2 Callback Distribution over Major Categories

To look further into the callback usage, we categorized lifecycle callbacks by their enclosing classes with respect to the four types of app components and the *Application* type corresponding to the `android.app.Application` class defined in the SDK. The rank (by the number of callsites and that of call instances) of each category is listed in Table 3, including the means and associated standard deviations over all benchmarks (in parentheses).

As shown, *Activity* lifecycle methods dominated the targets of all lifecycle-method callsites in code and also were invoked most frequently among all lifecycle-method call instances. The second most handled lifecycle events were associated with the application as a whole. Events handled by the other three types of components were close in their ranks, and were all considerably lower than the two dominant categories (i.e., *Activity* and *Application*). The ranks suggest that the vast majority of lifecycle 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 standard deviations suggest that the pattern was pretty consistent across the benchmark apps. Also, the pattern did not vary much from the static view (i.e., callsite diversity) to the dynamic view (i.e., call frequencies).

TABLE 3: Breakdown of lifecycle methods by categories with rank averages and standard deviations (in parentheses)

Category	Static view	Dynamic view
Activity	1.11 (0.37)	1.12 (0.38)
Application	1.77 (0.52)	1.78 (0.53)
ContentProvider	2.06 (0.72)	2.07 (0.72)
Service	2.09 (0.70)	2.09 (0.70)
BroadcastReceiver	2.11 (0.73)	2.11 (0.73)

TABLE 4: Breakdown of event handlers by categories with rank averages and standard deviations (in parentheses)

Category		Static view	Dynamic view
UI	View	1.07 (0.28)	1.08 (0.28)
UI	Widget	1.20 (0.44)	1.20 (0.45)
System	System status	1.22 (0.47)	1.21 (0.46)
UI	Dialog	1.23 (0.47)	1.23 (0.47)
UI	Media control	1.23 (0.49)	1.23 (0.49)
System	Hardware mgmt.	1.24 (0.49)	1.24 (0.49)
UI	App bar	1.24 (0.50)	1.24 (0.50)
System	Location status	1.24 (0.50)	1.24 (0.50)
System	App mgmt.	1.24 (0.50)	1.24 (0.50)
System	Network mgmt.	1.25 (0.51)	1.25 (0.50)

Table 4 presents a two-level breakdown of event handlers according to our manual categorization of those callbacks (see Section 3.2). Overall, in most of these apps, there were more callbacks triggered by UI events than those handling system events. A more detailed look reveals that the majority of UI event handlers dealt with two particular kinds of user interfaces, *View* and *Widget*, while user events on *Dialog* or *Media control* were less frequent. On average, most system event handlers responded to events that serve system management (*mgmt.*), with a few others dealing with hardware, location, and app management. The standard deviations of the average ranks imply that the sample means stably capture the general traits of the benchmark apps. Also, the ranking was quite consistent between the two views.

Figure 9 depicts how lifecycle method distribution evolved in terms of call instances over the past eight years. Results in the static view were very similar. Overall, the ranking of the lifecycle callback categories in both views was steady: the ordering was nearly constant albeit the ranks varied slightly. The top two categories were consistently the callbacks for managing the lifecycle of *Activity* and *Application* components. Also, for any year, the ranking in the static view was almost the same as the ranking in this dynamic view, suggesting that (1) the more dominant categories defined in the apps code generally led to their accordingly greater dominance in the apps execution, and (2) all the categories were exercised almost proportionally.

The evolutionary trend of event-handler category ranking in terms of call instances is illustrated in Figure 10 (ranking in the static view was similar). Like the lifecycle callback category ranking, the category distribution of event handlers appeared to be generally stable over the years (even more so than the category distribution of lifecycle methods). The top UI categories were constantly the callbacks that handle *View* and *Widget* events, while the top system callbacks were always those that handle *System status*. The usage of different categories of event handlers in code was also always consistent with the usage of corresponding categories at runtime. In contrast to lifecycle-method categorization, most of the event-handler categories always had very small differences in their ranks, which contributed to the ties in overall (i.e., across all years) ranking seen in Table 4.

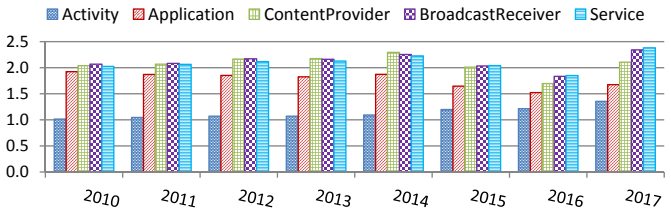

Fig. 9: The evolution of lifecycle callback category rank (y axis) in the dynamic view over years (x axis).

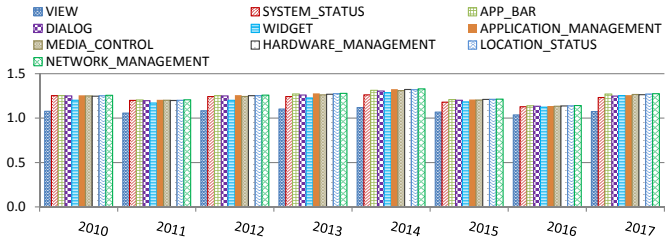

Fig. 10: The evolution of event-handler category rank (y axis) in the dynamic view over years (x axis).

Finding 4: The most often used callbacks were constantly those managing the lifecycle of *Activity* components or handling *View* and *System status* events. The ranking of callbacks by categories remained generally stable over the past eight years.

6.3 RQ3: Component Distribution and Communication

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 apps code and executions before examining the interaction between them through Intent ICCs. We then characterize whether data payloads are carried in the (Intents of the) ICCs.

6.3.1 Component Distribution

Figure 11 shows the distribution of callsites defined in code and call instances at runtime over four different component types. According to the *static view* (left chart of the figure), in the majority of the benchmarks at least 50% (54.1% on average) of components used in the code were *Activity*. *Broadcast Receivers* were used substantially too (over 26.5% on average), consistent with the previous observation that these apps had significant percentages of callbacks handling system events over all invoked callbacks (Table 4). *Service* and *Content Provider* components accounted for on average 10.1% and 9.4% of all components referred to in apps code, respectively. In a few outlier apps, almost all components were *Activity*, and there were apps in which up to 70–90% of components were one of the other three types. Meanwhile, there were also apps without including one of these four types of components—in particular, 25% of the 17,664 apps did not include any *Service* component.

The *dynamic* view (right chart), however, revealed that component distribution was even more skewed. In particular, the *Activity* components were extremely heavily exercised: on average, among all components executed in the benchmarks, 84.9% were *Activity* components. More specifically, in over 75% of these benchmarks, the percentage of *Activity* components was above the average. In contrast, other three types of components were executed very lightly—in the majority of our benchmarks, no more than 7% of run-time component instances were of any of

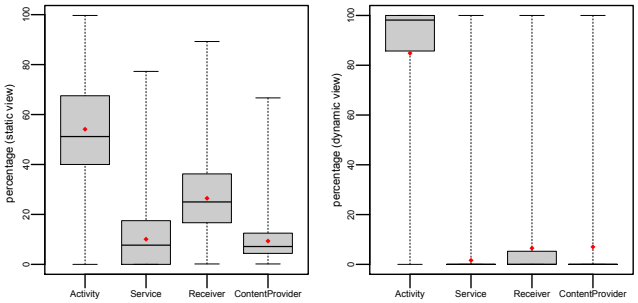

Fig. 11: Distribution (y axis) of callsites (left) and call instances (right) over four component types (x axis) in all benchmarks.

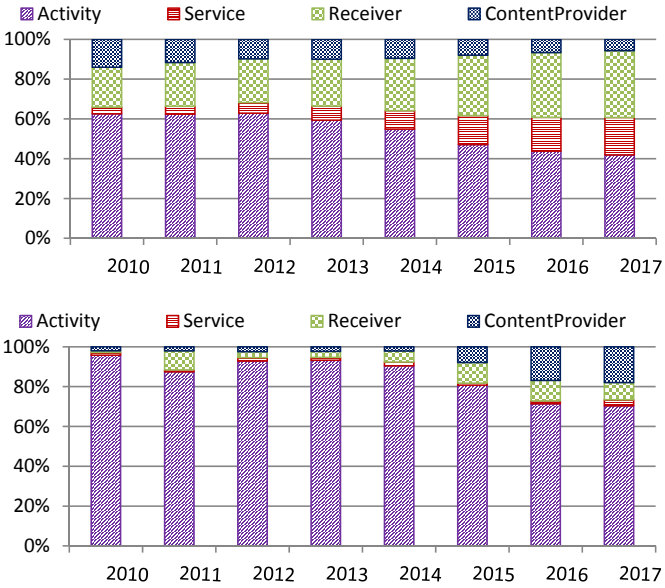

Fig. 12: The evolution of component type distribution (y axis) in the static (left) and dynamic (right) views over years (x axis).

the other types. Notably, despite their considerable portions in the apps code, *Services* and *Content Providers* that were exercised were almost negligible in terms of their portions among all exercised components. For the component types other than *Activity*, the outlying means indicated that only a few outlier apps had noticeable percentages of components of such types exercised, while in the executions of the majority of the apps the percentages were negligible.

Figure 12 shows how the component distribution in the apps code and execution evolved over the years. As expected, *Activity* components always dominated (accounting for 60% or above) in the apps components in each year, with the dominance being even greater at runtime (accounting for 70% or above). This is consistent with the observation from the entire set of apps across years (Figure 11). Notably, during the eight-year span, the slow drop in the usage (both in code and execution) of *Activity* components suggests a slight reduction in UI elements and activities in apps. A possible reason for the reduction is that newer apps have been becoming more diverse in terms of their structural composition, involving more non-GUI functionalities, which caused the drop of the *Activity* components' dominance.

Newer apps tend to include and exercise more *Broadcast Receiver* components, reflected in an increasing presence of these

components in the apps' code and execution. This trend might be the result of growing functionalities of newer Android devices and platforms (hence more needs for receiving system events associated with the richer hardware and system software features). Inclusion of *Service* components has been growing too. In contrast, increased inclusion of *Service* components has not resulted in an increased percentage of *Service* call instances. A possible reason is that app developers started adopting more service-oriented design by which more (non-GUI) components are structured as services to run in the background, yet app users still focus on exercising GUI components.

Notably, while newer apps tend to include fewer *Content Providers* in code, the execution frequency of these components has been growing substantially, especially in the last three years. A plausible explanation for these sharply contrasted patterns is that the apps increasingly dealt with data storage and management tasks at runtime, but the newer Android platforms have been making these tasks easier for developers thus fewer relevant calls are required in the code.

Finding 5: Over the past eight years, *Activity* has been constantly the most dominating component type, but with a slowly decreasing percentage of component executions. Newer apps defined more *Broadcast Receivers* and *Services*, using the former with increasing (but latter with decreasing) frequency. Newer apps defined fewer *Content Providers*, but exercised them more frequently.

6.3.2 Component Communication

The component-level communication in apps is realized through ICC APIs (e.g., `startActivity`, `getIntent`). On average, there were only 0.34% callsites and 0.58% call instances in all the studied benchmarks dedicated to ICC. In the majority (75%) of these apps, calls to ICC APIs accounted for only a marginal portion of all method calls. This result suggests that the overwhelming majority of calls were between methods *within* individual components. With respect to all call instances, components communicated with other components only occasionally (e.g., when the computation within a component completed and results were ready to deliver). In addition, more than half of all ICCs were between two *Activity* components; among the other ICCs, over 60% were either initiated by or aimed at *Activity* components. These observations are not surprising, given the dominance of *Activity* among all component types (see Figure 11). One implication of these observations is that the predominant use of ICCs in Android apps is to serve the communication among various user interfaces.

After understanding the usage of different types of components (Section 6.3.1), we break down all exercised ICCs (i.e., links between components) over the four categories (connection types, Section 3.2). Since static ICC linking would be highly conservative (hence very imprecise), we only examine ICC links that we created from app execution traces (by matching monitored Intent objects). Among all dynamic ICCs, the dominating type (37.92%) was *internal explicit*, followed by *external explicit* (36.77%) and *external implicit* (25.2%). There were only a negligible portion of ICCs being *internal implicit* (0.12%). Thus, components within an app almost always communicated *explicitly*, yet components across apps could communicate either implicitly or explicitly (albeit the connection was slightly more often explicit). Note that although we executed each individual app separately, they still often communicated with external apps (i.e., built-in/system apps such as `photo viewer`, `web browser`, and `camera`).

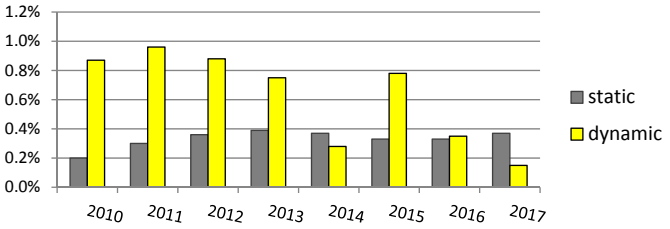

Fig. 13: The evolution of ICC use extent (y axis) in the static and dynamic views over years (x axis).

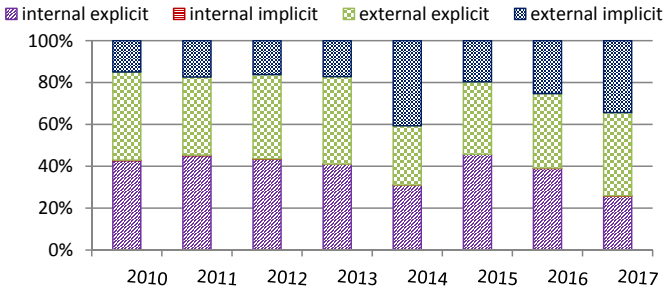

Fig. 14: The evolution of ICC category distribution (y axis, dynamic view) over years (x axis).

As depicted in Figure 13, the presence of ICC callsites in apps code fluctuated but very slightly, yet run-time invocation of ICCs has dropped noticeably since 2011. This implies that apps components have become more and more cohesive and the inter-component coupling has been gradually reduced in general. The absolute decrease was relatively small, though, given the tiny portion (<1.3%) of method calls devoted for ICCs overall.

Figure 14 shows how the ICC distribution by connection type evolved during the eight-year span. Due to their marginal presence in apps execution, *internal implicit* ICCs are not always visibly represented. Overall, the proportion of *external implicit* ICCs has increased, possibly due to the growing set of built-in/system apps on more recent platforms which the newer benchmark apps targeted: recall that user apps often communicate *implicitly* with built-in/system apps. Explicit ICCs, whether they were internal or external ones, tended to decrease (albeit slightly) over time, which implies that component-level decoupling (both with and across apps) was increasingly promoted in newer apps (e.g., for greater flexibility and easier upgrading of individual apps).

Finding 6: ICC calls were a marginal percentage of all call instances, which dropped over time. Components within an app rarely communicated implicitly, and components across apps communicated more often explicitly than implicitly. Over the years, the percentage of explicit ICCs tended to slowly shrink in both the internal and external categories, while the percentage of implicit ICCs across apps tended to increase.

6.3.3 ICC Data Payloads

Part of the reason that ICCs have been a major security attack surface is that they can carry, hence possibly leak, sensitive and/or private data. There are two ways in Android in which ICCs can carry data in an Intent: via the `data` field of the ICC

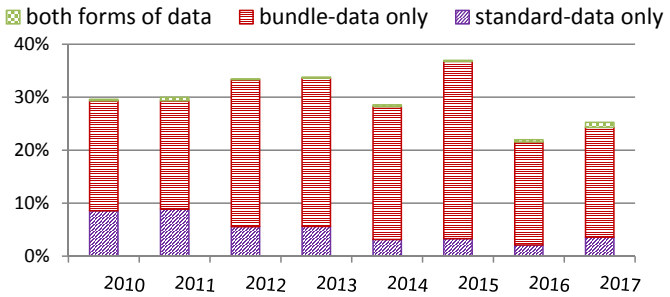

Fig. 15: The evolution of ICC data payload distribution (y axis, dynamic view) over years (x axis).

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.

Over all of our benchmarks, on average 4.83% of ICC calls executed carried standard data only, 24.72% carried bundle data only, and 0.4% carried both forms of data. Thus, most (70.1%) of the exercised ICCs did not carry any form of data in the associated Intent object. Very few ICCs carried both standard and bundle data at the same time. Bundles were clearly favored over URIs to transfer data at runtime. Thus, for data-leak detection, 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 [42].

From an evolutionary perspective, Figure 15 shows that the percentage of data-carrying ICCs as a whole first rose until 2013 and then started dropping (albeit not monotonically). ICCs carrying both forms of data were always marginal and did not change much over time. In general, apps have had continuous decrease in exercised ICCs that carried standard data only throughout the years. Our results further show that standard data was mostly carried in external implicit ICCs (accounting for 92.7% of all standard-data-carrying ICCs), while bundle data was typically (50.4%) carried in internal explicit ICCs. Among the 30% of ICCs that carried data by any means, 55.67% were within apps and the others were across apps. This indicates that ICCs across apps carry data generally as likely as ICCs within apps doing so.

Finding 7: Most (70%) ICCs did not carry any data at runtime, while in others bundle data was clearly preferred over standard data. Apps rarely carried both forms of data. Over the entire time interval, ICCs with standard data only have decreased as a percentage of all ICCs executed, and the total percentage of all data-carrying ICCs has dropped slowly from a high of 30% in 2010 to 25% in 2017, a noticeable difference of 5%.

6.4 RQ4: Sensitive Data Access

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

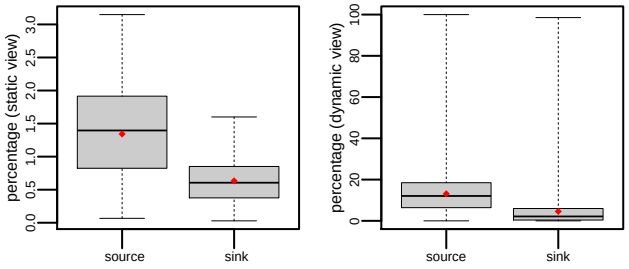

Fig. 16: Percentage (y axis) of sources and sinks (x axis) over all callsites (left) and all call instances (right).

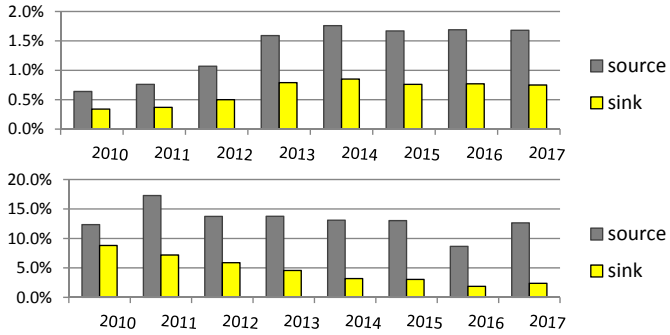

Fig. 17: The evolution of source/sink usage (y axis) in the static (top) and dynamic (bottom) views over years (x axis).

6.4.1 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 existence and frequency of API calls that are data sources or sinks as a percentage of all method calls, as shown in Figure 16. On overall average, the apps only had a small portion of callsites targeting sensitive access (1.34% for sources and 0.63% for sinks). However, these sensitive callsites were exercised quite frequently: on average in each app, over 13.1% of call instances retrieved sensitive data, and almost 4.6% of call instances performed critical operations. Note that the exercised sources and sinks were run-time projections of the predefined lists [52] of (17,920) sources and (7,229) sinks, respectively. Thus, the percentages of source and sink calls we reported were relative to these lists.

Figure 17 plots the evolutionary pattern of source/sink usage in terms of percentage of callsites (top) and call instances (bottom) for sensitive access. As shown, for both sources and sinks, the percentage of callsites (i.e., static view) experienced continuous growth in the first five years and then remained stable afterwards. In contrast, the dynamic view revealed that the run-time invocation of sources remained roughly stable in terms of their percentages over total method call instances despite fluctuations, whereas the calls to sinks have been steadily dropping throughout the eight years. Thus, newer apps tend to execute every sensitive-access callsite, especially sink callsite, less frequently over time.

Finding 8: Sources and sinks together represented only a tiny percentage (<2%) of all callsites in apps, although sensitive access was performed noticeably (up to 18%) at runtime. Over time, percentages of sources invoked in code and at runtime have remained generally stable, whereas sinks were executed decreasingly often despite their growing presence in code.

TABLE 5: Breakdown of sources over the top five categories with rank averages and standard deviations (in parentheses)

Category	static view	dynamic view
Network information	1.0 (0.02)	1.0 (0.04)
System settings	2.08 (0.61)	2.06 (0.60)
Location information	2.11 (0.66)	2.14 (0.68)
Calendar information	2.12 (0.70)	2.13 (0.71)
Account information	2.25 (0.83)	2.26 (0.83)

TABLE 6: Breakdown of sinks over the top six categories with rank averages and standard deviations (in parentheses)

Category	static view	dynamic view
Account setting	1.05 (0.24)	1.11 (0.35)
Logging	2.00 (0.74)	2.02 (0.80)
System setting	2.28 (0.86)	2.33 (0.89)
File operation	2.71 (1.11)	2.76 (1.14)
Network operation	2.73 (1.14)	2.77 (1.17)
SMS/MMS	2.79 (1.18)	2.82 (1.19)

6.4.2 Categorization of Sensitive Data Access

One way to further examine how Android apps use sensitive and critical API calls 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 the kinds of operations performed by the sinks. Knowing which of these kinds are most often accessed 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 5 lists the rank (by the numbers of source callsites and source call instances accessing sensitive information) of each source category, in the same format as Table 3. Only the top five categories are presented. Network information was dominant, constantly ranked first in any benchmark app, in both views. Network information was previously noted as widely accessed in Android apps [14], yet such consistent dominance of this category has not been reported. System settings, location info, calendar and account related data were also among the most commonly used categories of sensitive data [13], [14], [56].

A similar breakdown of sinks over six significant categories is shown in Table 6. The dominant category was *account setting*, suggesting that the 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, file, network, 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 [13], [29].

Next, we examine the evolution of source/sink distribution over the top categories. It is important to note that, despite variations in the ranking, the top five (six) source (sink) categories of apps in both views remained the same throughout

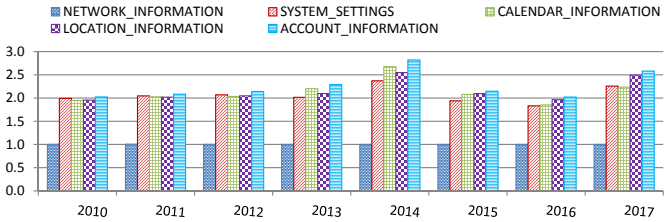

Fig. 18: The evolution of source category rank (y axis) in the dynamic view over years (x axis).

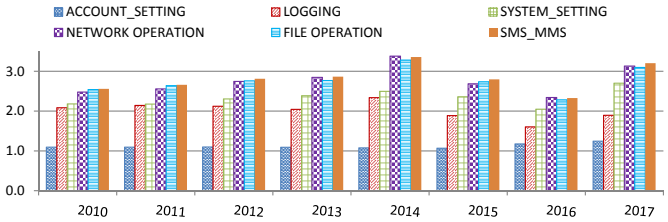

Fig. 19: The evolution of sink category rank (y axis) in the dynamic view over years (x axis).

the years, which are also the same as the top categories with respect to all benchmarks as a whole (Tables 5 and 6). Figure 18 depicts how source categorization evolved, showing that the ranking of source categories did not change much over time in the dynamic view (the results were similar in the static view). For any specific year and category, the ranks in both views were almost the same, suggesting that source callsites of varied categories were exercised quite proportionally in these apps.

The evolution of sink category distribution in terms of call instances is illustrated in Figure 19 (the results in terms of callsites were similar). The highest ranked category (`ACCOUNT_SETTING`) was also the most stably ranked one. Logging operations have been increasingly targeted by more sinks (despite the slow growth), while the rank of system-setting operations has slightly dropped—higher ranks are associated with lower rank values. Overall, the relative ranks of these six sink categories have remained stable over the time interval studied.

Finding 9: Network information and account setting was the most accessed data and operation by sources and sinks, respectively. The ranking of the top source and sink categories remained generally stable over time, in code and at runtime.

7 Lessons Learned and Recommendations

In this section, we discuss the implications of our major empirical findings and make recommendations on improving code-based static/dynamic app analysis and security defense in Android.

7.1 On Functional Structure

Lessons learned. The *structure* metrics show that Android apps depend heavily on the platform in construction and also are highly framework-intensive at runtime. While it is intuitive and straightforward that framework-based applications generally rely on the functionalities of the underlying framework (e.g., through SDK APIs), the peculiar dominance of the capabilities and behaviors of the framework in those of user apps as revealed in this study has not been reported before. In accordance with this degree of dominance, the little proportion of user code in apps partly explains the prosperity of the Android ecosystem powered by the prominent richness, tremendous number, and great

diversity of user applications—developing a functionality-rich user app in Android is much facilitated, hence the effort needed for app development largely reduced, by what the platform can offer. Another implication of the framework dominance in app construction and execution is that a clear, deep understanding of the Android platform and its interface with user apps is essential for analyzing and securing Android apps.

Moreover, from an evolutionary perspective, the increasing dominance of calls *within* the SDK (Finding 2) indicates growing support of the Android platform for continuously improving app development productivity. Meanwhile, the fact that the proportion of user code in apps and their executions shrunk steadily signifies the trend that the code that Android developers needed to write continually decreased. Given the relatively long (compared to most prior Android app evolution studies [18], [19], [23], [34]) span of our study, there is a reasonable likelihood that these trends would continue (albeit probably for only a limited number of years—and then the trend will stabilize).

Recommendations. Our results indicate that the security of the Android framework itself deserves continuous attention in securing the whole Android software ecosystem. Thus, to secure the Android ecosystem, researchers need to pay more attention to analyzing the security of the Android platform itself when developing an analysis technique for the security of Android apps—current app security defense solutions tend not to analyze the framework itself (e.g., they mostly focus on the security of user apps themselves; although in some cases framework related security features were utilized yet still the framework's security was rarely assessed directly) [2], [12]. Also, the overwhelming involvement of SDK in app executions (Finding 1) uncovers promising benefits of SDK optimizations for app performance at runtime—the majority of method calls in an app targeted SDK APIs. Thus, researchers and developers on performance optimization techniques and tools for Android apps should first and foremost address potential performance issues with the platform itself. These opportunities are worthy of even more attention and the recommendations more relevant as the dominance of SDK continues to increase over time. In addition, given the already substantial and historically growing third-party code inclusion and execution in apps, analyses for app security would be suggested to be preceded with an accurate third-party library detection technique for problem isolation—e.g., these libraries are often a source of vulnerabilities in the host apps [57].

7.2 On Callbacks

Lessons learned. *Activity* was the predominant target of lifecycle method calls, which indicates that Android apps are generally rich in user interfaces (UIs). Moreover, this predominance has not changed much during the evolution of Android apps (Findings 4 and 5). While the indication seems to be self-evident and has been implied in prior peer studies [16], [20]—after all, *Activity* components are the standard building blocks used to develop UIs in Android, the predominance has not been quantified before, nor has the extent to which various types of components are *exercised* at runtime been studied. This dominance also suggested that the Android platform is most heavily involved in managing *Activities*, compared to managing other types of components, and that the majority of the rich and diverse base of Android apps are interactive apps, as opposed to apps without UIs (e.g., background services).

Since the callbacks our study was concerned with are the methods generically defined in the Android platform that are overwritten by developers, the small proportions of callback references in code and callback invocations at runtime implied that app developers did not have to customize much of the platform's common capabilities. This also suggests that the Android framework is designed in a way that can accommodate most user applications' needs immediately. Meanwhile, given these callbacks are part of the user code, the small percentages of user code in apps and their executions are a straightforward reason why callbacks are neither extensively referenced in code nor much called during executions of apps. The evolutionary trend of callback usage continuously dropping further pointed to the direction of Android framework moving toward greater versatility. In addition, with respect to the dedicated functionality domain of each callback category, both the stability of category distribution of statically used callbacks and that of dynamically invoked ones in apps corroborated that the overall functionality distribution (e.g., how much of an app deals with UI versus how much of the app deals with data storage and management) of Android apps has sustained against the ecosystem's evolution.

Recommendations. Given the largely UI-centered design of Android apps, Android app analyses should carefully model app behaviors that are relevant to UI elements (e.g., UI-induced data and control flows). Since invocations of various callbacks account for only small percentages of all method calls (Finding 3), it would be practical (concerning efficiency and scalability) for dynamic app analyses that desire to perform precise callback data/control flow analysis to fully track callback data/control flows at runtime, so as to produce fine-grained analysis results (instead of substantially sacrificing effectiveness, for instance by adopting a coarse granularity). Such a strategy would be rewarding as regards to analysis accuracy (especially, for higher precision). This is particularly relevant as callback usage in apps code and execution has been dropping over the years.

Also, given the skewed distribution of callback categories (i.e., prominent dominance by one or two categories), app analyses like lifecycle modeling, taint checking, and callback control flow analysis should consider prioritized analysis strategies (i.e., giving priority to the very few top-ranked categories of lifecycle methods and event handlers) in order to achieve a greater level of cost-effectiveness (e.g., trading accuracy for scalability with the gains considerably overweighing the losses). For example, Flowdroid [6] remains a state-of-the-art static taint analysis for Android, yet it can be highly computation- and memory-intensive hence generally suffers scalability barriers [58]; the later approach in [58] achieved substantial improvement in efficiency and scalability, yet faces usability challenges with the results it produces. With the recommended prioritization strategy, we may configure Flowdroid to only consider callbacks of the top two categories—in our case studies with a dozen of large/complex apps, the prioritization led to over 30% efficiency gains at the cost of no more than 4% precision loss, resulting in a significant cost-effectiveness improvement. This can be explained by the fact that the iterative callback analysis in Flowdroid accounts for a major portion of its overall time cost according to the nature of its core algorithm. For another example, the dominance of *Activities* 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 [11], to reduce analysis complexity and/or to achieve better efficiency. Given that the ranking of these top categories tended to be quite stable (Finding 4), such prioritization strategies would also be a long-term methodology.

7.3 On Components and Their Interactions

Lessons learned. Our ICC characterization reveals that components of the same apps communicate rarely through implicit ICCs as opposed to through explicit ICCs. Between components across apps, the proportions of explicit ICCs were even more substantial than those of implicit ICCs—in contrast to the general intuition that implicit ICCs would dominate external ICCs given that connecting to external apps through explicit Intents is less convenient/handy than explicitly targeting internal components. Nevertheless, on average, over a quarter of ICCs were implicit across apps. The implication of the presence of implicit external ICCs during app executions concerns the precision of static analyses of ICC-induced information flows, which typically link components via implicit ICCs in a conservative manner (i.e., through the `action`, `category`, `data` matching tests [4], [8]). Apparently the conservative linking leads to imprecision of the static app analyses. Moreover, given that the presence of implicit external ICCs is pervasive and substantial, this imprecision can be excessive.

Our study on ICC data payloads reveals that most ICCs did not carry any data and those carrying data tend to do so preferably via bundles instead of URIs. Moreover, ICCs carrying data solely via valid URIs have been in decline over time (Finding 7). An immediate indication of this contrast and trend is that ICC-based data communications among components in and across apps tend to be structured (i.e., the key-value map structure of *bundles*) as opposed to scalar data (e.g., a string holding the URI data). A plausible reason for the growing preference of bundles over URIs lies in the former providing more flexibility, capacity, and capabilities.

Recommendations. Since implicit ICCs are common, a static app analysis needs to account for the effects of conservative implicit ICC linking on the analysis's effectiveness as a major source of potentially considerable imprecision. On the other hand, despite the imprecision, a *sound* app analysis, within or across apps, should completely model the app's (both implicit and explicit) communication with external components—in particular, an intra-app analysis would suffer from loss of recall if it dismisses external ICCs, because individual apps may still communicate with built-in apps via explicit/implicit ICCs. This is peculiarly necessary as implicit ICCs across apps are on the rise while explicit external ICCs have accounted for a stable, significant percentage of all ICCs (Finding 6).

There is a need to carefully address `bundles` in ICCs for ICC-involved security testing [42]—examining URI data transfers is not sufficient. Especially, given the preference of apps using bundles over URIs for data communication via ICCs, app testing and security diagnosis techniques should perform deeper analysis of the `extras` fields in ICC Intents. Meanwhile, security analyses involving ICCs may benefit from *prioritizing examination of ICCs that carry data, especially those using bundles, to obtain more effective results* within a time budget, because ICCs that do not carry any data would be less likely to facilitate malicious behaviors or serve as attack surfaces than data-carrying ICCs would do, according to our recent study on ICC-induced communication risks in Android apps [59]. In fact, in prior work [60], [61], we have demonstrated that the prioritization strategy can substantially enhance the ICC analysis performance for Android.

7.4 On Sensitive Accesses

Lessons learned. Our results on sensitive access in Android apps (Finding 8) revealed that such access was not quite prevalent in apps code (i.e., only $<2\%$ of all callsites aimed at such access). However, a quite significant portion of method calls at runtime were for sensitive access, implying that each of the sensitive access callsites was exercised quite intensively. Thus, sensitive access represented a kind of *hotspots* in apps.

In terms of the categories of information/operations associated with it, the sensitive access generally did not target *broadly* but was rather focused—the majority of the sources/sinks accessed only very few kinds of sensitive data (e.g., network information) and critical operations (e.g., account setting). Notably, these categories are indeed highly relevant to the common functionalities (e.g., networking, maintaining account information) of mobile devices. Considering that the access in app code reflects more of developers' intentions while the access in app executions reflects more of users' intentions, the high consistency between the static and dynamic views regarding the categories of sensitive access implied the consistency between developer and user intentions (in other words, developers seemed to be able to capture user intentions in this regard).

Also, over time, these dominating categories and their relative rankings remained almost the same (Finding 9). The specific ranks were stable too, evidenced by the generally small variations of the ranks across sample apps (Tables 5 and 6). These trends indicate that the developer and user preferences on sensitive access have been consistent against the Android ecosystem evolution.

Recommendations. Given the strong dominance of a few (one or two) categories of sensitive access in terms of the underlying sensitive data/operations in Android apps, app security diagnosis and defense techniques based on information flow analysis (e.g., [4], [6]) may be optimized (for better cost-effectiveness tradeoffs) by prioritizing the search for sensitive information 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 sensitive flows, resulting in improved cost-effectiveness. End users and security analysts should also pay more attention to these highly accessed categories to make better decisions on permission management and app vetting—for example, requests for permissions for the app to access the top-category information can be regarded less suspicious (hence deserve greater scrutiny) than permission requests for much less common categories of sensitive access.

Further, the long lists of predefined sources and sinks used by static taint analyses may be prioritized to focus on the ones used most often: the taint analysis would just focus on computing taint flow paths between the most often used sources and sinks, instead of exhaustively checking against the long, full lists. 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 80% 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 [4], [6].

In addition, given the high consistency of sensitive access category ranking over the years, the above optimization strategies can be considered long-term approaches to cost-effective app information flow analysis and its client techniques. Further, given the consistency of the category ranking between static and dynamic views, both static and dynamic approaches (information flow analysis and its applications) can use similar prioritization strategies for the purpose of cost-effectiveness optimization.

8 Threats to Validity

The primary threats to the *external validity* of our study results concern our choice of benchmark apps and the test inputs we used for dynamic analysis. First, we studied a limited number of Android apps from each of the eight years (mainly because of the large overheads of the dynamic profiling needed), which may not be representative of all Android apps on the market or in use during that year. To mitigate this threat, we started with a much larger pool of apps for each year from the respective source, and picked each initial app from that pool randomly. In addition, our benchmarks were originally from a large variety of sources: AndroZoo [30] itself is a highly diverse collection of apps. Nevertheless, given the huge total number of apps in the wild, our sample apps only represent a small portion of the entire app population per year, thus the empirical findings and our conclusions based on the findings are best interpreted with respect to the studied apps—we do not claim that our results hold for all Android apps.

Second, like any other dynamic analyses, our empirical results on run-time behaviors 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 only used benchmarks for which the dynamic inputs did achieve a reasonable coverage (60% at least and 74% on average). Nonetheless, our conclusions and insights based on the metrics in the dynamic view are limited to the exercised app behaviors. Also, the coverage threshold applied during benchmark selection potentially affected the representativeness of the apps we chose.

Another threat to external validity is that our static code analysis of apps for computing metrics in the static view may not be sound due to undetected obfuscation. It is well known that Android apps widely adopt various obfuscation schemes [62]. Due to the obfuscation, the target of some callsites may not have been recognized correctly, leading to the incompleteness of our static analysis. Fortunately, obfuscation is more often used in malware for evading anti-malware defense. A recent study revealed that only a small portion ($<$25%) of benign apps are obfuscated [63]. Moreover, since our static study only concerns method calls and class hierarchy, only a few obfuscation schemes could affect our results.

Concerning the external validity threats related to benchmark selection, we also note that many of our findings were highly consistent with what we found in a preliminary study against a different, small, one-year dataset [26]. For example, both studies found that the functionalities of Android apps heavily and increasingly relies on those of the Android framework, lifecycle callbacks mostly target *Activity* components which also constantly dominated over other component types, user interface events (as opposed to system events) were the main triggers of event handlers, and sensitive data sources primarily retrieved *network information* and *system settings* while sensitive data sinks performed *account setting* and *logging*. These consistencies hold despite small differences in the values of respective metrics, which are expected given that different benchmarks were used. In all, none of our main conclusions changed between that preliminary study and the current study. Thus, this work further confirmed the earlier results, corroborating the credibility of our findings on the characteristics of Android apps (hence that of our lessons learned and recommendations made). The result comparisons between the two studies suggested that the test generation tools chosen did not have much impact, at least for the benchmarks studied in terms of the characterization metrics used.

The main threat to *construct validity* concerns the metrics and measurement procedures we used to quantify the code structure and behavior of the benchmarks. To mitigate this threat, we considered a diverse set of metrics in three orthogonal dimensions, which together constitute a sensible profile of app characteristics. In addition, we adopted varied ways of measurement including the full data distribution and summary statistics (mean and standard deviations).

As none of the benchmarks were identified as malicious by VirusTotal [33], a threat to *conclusion validity* is that 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). 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 study on different benchmarks [26] reveals that the deviations were mostly small. More importantly, the major observations remained 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 among callbacks those to *Activity* components), contrasts (e.g., ranking of callback categories), 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 and their evolutionary patterns. The numbers should be taken as estimates that assist understanding.

9 Related Work

Three main categories of previous work are related to ours: general characterizations of Android apps, security of the Android ecosystem, and analysis of code for Android.

9.1 Characterization of Android Apps

Empirical works targeting Android have emerged lately, covering a broad scope of topics ranging from resource usage [14], battery consumption [17], permission management [64], code reuse [16] to ICC robustness [54], SDK stability [18], and user perception of security and privacy [56]. In contrast, our work aims at the general characterization of Android applications from the point of view of language constructs and run-time behaviors.

A few prior studies also looked at the code structure of apps. For instance, with 20 sample subjects, Minelli and Lanza aimed

to understand how Android apps are different from traditional software applications in terms of source size (LOC), use of third-party libraries, and version history data [19]. They found that the dominant proportion of calls targeted third-party libraries, different from our finding that the majority of all method invocations are calls to the Android SDK—they did not particularly examine the extent of SDK calls. The study also suggested substantial percentages of user code (referred to as *core* elements) in apps, contrary to our results showing very small proportions of user code (in relation to the heavy use of SDK). While also applying an evolutionary view like our study, their study looked at the historical versions of the same apps over a period of time (according to the version history data available), as opposed to our study looking at the varied apps across years as representatives of the entire population of each year at a much larger scale and longer span. On one individual app, they found the number of user code entities grew over time, in contrast to our finding that the proportion of user code kept shrinking over years. These differences in findings can be attributed to the differences in sample size (our 17,664 versus their 20 apps) and methodologies (e.g., their looking at different versions of same apps versus ours looking at apps of different years).

Use of third-party libraries is a kind of code reuse. Mojica et al. conducted a large-scale study of Android apps concerning code reuse through class inheritance, class reuse, and (framework) reuse of whole apps [20]. The authors found that 18.75% of all classes in 208,601 apps studied inherited from an Android platform base class, and 8.4% of all particularly inherited from the *Activity* class. While our study shares these perspectives, our findings are quite different in that our results showed much greater extent of framework reuse (in terms of calls to SDK APIs) and much greater dominance of *Activity* components. A plausible reason for the differences is that their study looked only at apps of a particular (early) year (i.e., 2011) while we examined apps of eight different years and most of apps we studied are much newer than their benchmarks. Following up this study by Mojica et al., Linares-Vasquez et al. investigated the effects of code obfuscation and use of third-party libraries on the empirical studies of code reuse in Android [21], where the authors found that code-reuse study results can be significantly different between considering and dismissing third-party libraries or obfuscated apps. Neither of the studies in [20] and [21] investigated the evolutionary characteristics of apps over years as we did, and accordingly they did not report findings on how the studied app traits changed over time. Also, like the study by Minelli and Lanza and that by Mojica et al., this study is static in that no concrete app executions were analyzed, as opposed to our study being dynamic in addition to considering static views.

Several Android app characterization studies only targeted Android malware [23], [24], [25]. These studies either utilized static analysis [25], [52] or relied on manual investigation [23]. In contrast, we focused on understanding the run-time behaviors of benign Android apps, which potentially complements those previous studies. Like ours, the evaluation study in [34] also applied an evolutionary lens to Android apps, yet focusing on the code anti-patterns against general software quality. Also different from our work is the methodology of examining the (3,568) evolved versions of a few number of (106) same apps across two years, versus ours of looking at thousands of apps as samples of the yearly populations for eight consecutive years.

In preliminary study [26], we characterized a small set of (125) apps from year 2015 to understand their dynamic traits. In comparison, this paper largely expanded that study in multiple ways. First, the experimental scale is largely extended: the benchmark suite is enlarged by over 100 times. Second, the study scope is expanded to include static characteristics in addition to run-time behaviors, as well as the relationships between the two views with respect to the characterization metrics. Third, by examining apps from eight years (versus one particular year in the earlier study), we further investigated the evolutionary pattern of each metric to understand how Android apps evolve in their code structure and behavioral traits, and discussed the implications of new findings. Fourth, we ran the experiments on a real device with a state-of-the-art industry-maintained Android app input generator (Sapienz), versus the earlier study profiling apps an emulator against inputs generated from the Monkey tool [44].

9.2 Security of the Android Ecosystem

Regarding the analysis of sources and sinks in Android apps, our study is not the first. Recent studies on taint analyzers for Android [65], [66] investigated the effect of the lists of sources and sinks considered on the effectiveness results reported in the original evaluation study of respective analyzers. For a fair comparison among selected analyzers, these studies used the lists produced by SuSi [52] (independent of the study subjects) or those extracted from the particular study subjects. Avdiienko et al. [67] used FlowDroid [6] to compute taint flow paths between sources and sinks identified by SuSi and then uplifted the paths to the source/sink category level for building a one-class malware classifier. Beyond presenting the classification technique, the authors also characterized use of sources and sinks in the benchmark apps in terms of the percentage distribution of taint flow paths between various source/sink categories. These studies are static in nature and they looked at the data flows between sources and sinks. Our study is both static and dynamic while simply examining the individual calls to sources and sinks without computing the data flows. We also looked at the evolutionary patterns of source/sink usage which is not addressed in these prior peer studies.

Concerning the security of the Android platform, the study in [68] offers a dedicated view into the varied types of vulnerabilities in different layers (from the operating system kernel all the way up to the user applications) of the platform through a vulnerability taxonomy. The study also investigated the extent to which different layers are affected by the vulnerabilities as well as the timing of vulnerability fixes in Android. With a focus on the security of SDK APIs (application framework layer) of the platform, Thomas et al. studied the vulnerabilities of JavaScript-to-Java interfaces with respect to their origins and lifetime [69]. The authors found an exponential decay model that describes the fix rate of these vulnerabilities, which is useful for understanding their lifetime.

Similarly but focusing on a different perspective, Cao et al. [70] examined the system services in the Android framework particularly concerning input-validation vulnerabilities, measured attack surfaces for these vulnerabilities, and developed a fuzzing tool for detecting this kind of vulnerabilities in the Android system services. In [71], the authors proposed a set of three metrics (referred to as FUM) to quantify the critical vulnerability factors as regards to the device manufacturers and network operators in the Android ecosystem. Their study revealed that

vulnerability fixes were generally not timely delivered albeit the delivery varied across different parties, and that security updates offered by manufacturers were not sufficient due to little incentive. More recently, Jimenez et al. [72] manually inspected Android system vulnerabilities archived in the National Vulnerability Database (NVD) from 2008 to 2014 and characterized these vulnerabilities regarding their code locations, complexity, and fixing effort.

In sum, these studies called for more attention to securing the Android platform against varied kinds of code vulnerabilities, which our study echoes. Meanwhile, we raise the concern for the general security of the Android platform based on their dominating involvement in the functionalities of user applications, different from these prior studies looking at particular security threats (e.g., input-validation vulnerabilities) of the platform (or part of it such as system services) directly.

9.3 Code Analysis for Android Apps

While our study computes the proposed characterization metrics in the static view, our static code analysis of apps is minor and very straightforward. Thus, given the huge space of code analysis for Android apps in general, we address only two aspects that are most relevant to our study: dynamic analysis of Android apps and code analysis for Android app security. A great survey of static code analysis for Android apps can be found in [73].

Dynamic Analysis of Android Apps. A few dynamic analyses focusing on Android involved tracing method calls as well, for malware detection [74], app profile generation [22], and access control policy extraction [75]. 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.

Code Analysis for Android Security. 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 [4], [6], approximating the Android runtime environment [5], [7], and analyzing lifecycle callbacks and event handlers [6], [11] 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 [29], [76] in general and taint analysis [6], [77] 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 combines static and dynamic analyses. 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.

10 Conclusion

We presented a systematic, longitudinal characterization study of Android apps that targets a general understanding of application code structure and run-time behaviors, as well as their implications for improving Android app analysis and security defense. Our study investigated the functionality composition and distribution, component distribution and communication, ICC data payloads, use of lifecycle callbacks and event handlers, and sensitive data access, of apps developed during eight past years.

Our extensive study has enabled a series of notable findings about both the overall coding features (static profile) and run-time behavioral traits (dynamic profile) of Android apps with respect to the studied samples, and their evolutionary dynamics with respect to the same profiles over the studied time span. In particular, our study findings indicate that (1) Android apps are increasingly framework-intensive and platform-reliant, with growing use of various SDK APIs, (2) only a small portion of method calls targeted lifecycle callbacks (mostly for *Activities*) or event handlers (mostly for user interactions), and newer apps tended to have lesser overall use of callbacks, (3) the majority of executed ICCs did not carry any data payloads, others passed data mainly via bundles instead of URIs, (4) sensitive and critical APIs focused mainly on network information/operations, account setting, and logging, and (5) ranking of callback and source/sink categories remained stable during the studied span of years. In addition, we offered insights into the implications of our empirical findings that potentially inform of 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] Google, "Android and Kotlin," https://developer.android.com/kotlin, 2019.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, p. 76, 2017.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] R. Minelli and M. Lanza, "Software analytics for mobile applications–insights & lessons learned," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 144–153.

[20] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE software*, vol. 31, no. 2, pp. 78–86, 2013.

[21] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 242–251.

[22] 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.

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

[24] 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.

[25] 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.

[26] H. Cai and B. Ryder, "Understanding android application programming and security: A dynamic study," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 364–375.

[27] H. Cai, "Systematic dynamic characterization of Android apps," http://chapering.github.io/droidfax/, 2017.

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

[29] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for real-time privacy monitoring on smartphones," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 393–407.

[30] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[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. 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.

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

[34] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 236–247.

[35] Google, "App categories on Google Play," https://play.google.com/store/apps, 2019.

[36] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "Fsquadra: fast detection of repackaged applications," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2014, pp. 130–145.

[37] M. Dilhara, H. Cai, and J. Jinkens, "Automated detection and repair of incompatible uses of runtime permissions in android apps," in *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft)*, 2018.

[38] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 216–227.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 204–217.

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

[45] Facebook, "Sapienz: Intelligent automated software testing at scale," https://code.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/, 2018.

[46] H. Cai and B. Ryder, "DroidFax: A toolkit for systematic characterization of android applications," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 643–647.

[47] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for Android," in *IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 658–668.

[48] 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.

[49] Google, "Android Debugging Bridge," https://developer.android.com/studio/command-line/adb, 2015.

[50] ——, "Android Log," https://developer.android.com/reference/android/util/Log, 2015.

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of ACM Conference on Computer and Communications Security*, 2016, pp. 356–367.

[58] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 106–117.

[59] K. O. Elish, H. Cai, D. Barton, D. Yao, and B. G. Ryder, "Identifying mobile inter-app communication risks," *IEEE Transactions on Mobile Computing*, vol. 19, no. 1, pp. 90–102, 2018.

[60] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "Prioritized analysis of inter-app communication risks," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 159–161.

[61] ——, "Mr-droid: A scalable and prioritized analysis of inter-app communication risks," in *2017 IEEE Security and Privacy Workshops (SPW)*, 2017, pp. 189–198.

[62] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2018.

[63] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," *arXiv preprint arXiv:1801.02742*, 2018.

[64] 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.

[65] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 331–341.

[66] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 176–186.

[67] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2015, pp. 426–436.

[68] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on android-related vulnerabilities," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 2–13.

[69] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android api vulnerabilities: case study on the javascript-to-java interface," in *Cambridge International Workshop on Security Protocols*. Springer, 2015, pp. 126–138.

[70] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 361–370.

[71] D. R. Thomas, A. R. Beresford, and A. Rice, "Security metrics for the android ecosystem," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 87–98.

[72] M. Jimenez, M. Papadakis, T. F. Bissyandé, and J. Klein, "Profiling android vulnerabilities," in *2016 IEEE International conference on software quality, reliability and security (QRS)*. IEEE, 2016, pp. 222–229.

[73] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

Haipeng Cai is an assistant professor in the School of Electrical Engineering and Computer Science at Washington State University, Pullman. He obtained his PhD in computer science from University of Notre Dame in 2015. His current research interests are software engineering and software systems in general with a focus on program analysis and its applications to software reliability and security.

Barbara Ryder is an emerita faculty member in the Department of Computer Science at Virginia Tech, where she served as Department Head (2008-2015) and retired on September 1, 2016. Previously, Dr. Ryder served on the faculty of Rutgers and worked at AT&T Bell Laboratories in Murray Hill, NJ. Dr. Ryder became a Fellow of the ACM in 1998, and received 2018 IEEE-CS TCSE Distinguished Women in Science and Engineering (WISE) Leadership Award. the ACM SIGSOFT Influential Educator Award (2015), the Virginia AAUW Woman of Achievement Award (2014), and the ACM President's Award (2008). Dr. Ryder led the Department of Computer Science at Virginia Tech team that tied nationally for 2nd place in the 2016 NCWIT NEXT Awards. Dr. Ryder has advised 16 Ph.D. and 3 M.S. students to completion of their theses; she has supervised the research of 4 postdocs and more than 30 undergraduate researchers at Rutgers and Virginia Tech.