A Study of Run-time Behavioral Evolution of Benign versus Malicious Apps in Android

Haipeng Cai[a,*], Xiaoqin Fu[a], Abdelwahab Hamou-Lhadj[b]

[a] *Washington State University, United States*
[b] *Concordia University, Canada*

Abstract

Context: The constant evolution of the Android platform and its applications have imposed significant challenges both to understanding and securing the Android ecosystem. Yet, despite the growing body of relevant research, it remains unclear how Android apps evolve in terms of their run-time behaviors in ways that impede our gaining consistent empirical knowledge about the workings of the ecosystem and developing effective technical solutions to defending it against security threats. Intuitively, an essential step towards addressing these challenges is to first understand the evolution itself. Among others, one avenue to examining a program's run-time behavior is to dissect the program's execution in terms of its syntactic and semantic structure.

Objective: In this paper, we study how benign Android apps execute differently from malware over time, in terms of their execution structures measured by the distribution and interaction among functionality scopes, app components, and callbacks. In doing so, we attempt to reveal how relevant app execution structure is to app security orientation (i.e., benign or malicious).

Method: By tracing the method calls and inter-component communications (ICCs) of 15,451 benign apps and 15,183 malware developed during eight years (2010–2017), we systematically characterized the execution structure of malware

[*] Corresponding author
Email addresses: haipeng.cai@wsu.edu (Haipeng Cai), xiaoqin.fu@wsu.edu (Xiaoqin Fu), wahab.hamou-lhadj@concordia.ca (Abdelwahab Hamou-Lhadj)

versus benign apps and revealed similarities and disparities between them that are not previously known.

Results: Our results show, among other findings, that (1) despite their similarity in execution distribution over functionality scopes, malware accessed framework functionalities mainly through third-party libraries, while benign apps were dominated by calls within the framework; (2) use of *Activity* component had been rising in malware while benign apps saw continuous drop in such uses; (3) malware invoked significantly more *Services* but less *Content Providers* than benign apps during the evolution of both groups; (4) malware carried ICC data significantly less often via standard data fields than benign apps, albeit both groups did not carry any data in most ICCs; and (5) newer malware tended to have more even distribution of callbacks among event-handler categories, while the distribution remained constant in benign apps over time.

Conclusion: We discussed how these findings inform understanding app behaviors, optimizing static and dynamic code analysis of Android apps, and developing sustainable app security defense solutions.

Keywords: Android apps, security, execution, evolution, longitudinal study

1. Introduction

Android as the dominant mobile operating system continues to gain in its momentum both on the traditional smart device market [1] and beyond [2]. Accompanying this trend is the unceasing surge of mobile malware, a market that is also predominated by Android [3]. Numerous defense solutions have been developed for securing the Android ecosystem [4, 5], yet many of them quickly became *outdated* due to the fast evolution of both the Android platform and its user applications (known as *apps*), according to recent studies of our own [6, 7, 8] and others [9].

A peculiar evidence of Android evolution perplexing security defense is the relatively *short-span capability* of learning-based classification techniques [10, 11, 12, 13, 14]. Typically these techniques work by extracting certain sets of features from samples of benign and malicious apps based on, for instance, how apps use permissions [15, 16, 17] and/or APIs [18, 19, 20, 21, 22, 23]. A classifier can be then trained on these samples to classify unknown apps. The evolution of Android itself and that of app development paradigms, however, may largely impede or even render almost unusable the classifier in identifying new samples [24]. Note that retraining may not always be a solution there since new samples may not be as soon available as needed (e.g., for recognizing zero-day malware). As an example, a state-of-the-art learning-based app classifier, which focused on long-span classification, achieved highly competitive performance (over 96% accuracy) only for one year [6]. Its classification accuracy degenerated to merely a random prediction (50%) when working on apps over two years newer than training samples [7]. Since feature engineering is a key to learning-based app classification, *knowing how benign and malicious apps evolve would inform design of security solutions for Android that works effectively for a longer span.* For instance, the study on the evolution of run-time behavioral differences between benign apps and malware may reveal app characteristics that can consistently separate these two groups over time; metrics of such characteristics can then be used as features for training a machine learning classifier to develop a dynamic malware detector that sustains its classification performance for a longer time than would a classifier built on features that differentiate the two app groups for a short period of time (e.g., only differentiating malware from benign apps developed in a particular year).

The constant evolution of Android also poses significant challenges for app development and testing. The Android platform evolves to harness the full potential of new-generation hardware capabilities of the host device, while apps evolve to accommodate the platform evolution. This justifiable symbiosis has resulted in fragmentation and various other complicated app/device compatibility issues [25, 26, 27, 28, 29] which have become immediate barriers for app development, understanding, and testing. Developing *well-informed* *app development and testing strategies would also require an understanding of the evolution dynamics of benign apps and malware* (part of the Android ecosystem). For instance, leveraging the behavioral differences between benign apps and malware in terms of code-level execution structures, especially the evolutionary patterns of the differences, would facilitate more precise detection and even repair of run-time incompatibilities in Android apps [28]. For example, since many of these compatibility issues are closely relevant to callbacks, ICCs, and other APIs, patterns in the use of these APIs in incompatible apps may be used for incompatibility detection. Thus, the differences between malware and benign apps in the use of these APIs, as revealed from the evolution dynamics between these two app groups, would help develop detection techniques that are able to pinpoint compatibility issues differently in benign apps from those in malware, hence to achieve higher detection precision.

The need for those evolution-wise understandings has been recognized in prior works, with varying focuses on the particular aspect of the evolution dynamics (e.g., permission [30] and API [31]). Researchers also have studied the evolution of Android apps in terms of antipatterns in their code to assess app quality changes over time. These prior studies exclusively target a *static* characterization by examining the code or assets in app packages (i.e., APKs), not concerning the *actual run-time behaviors* of these apps. Their results, with respect to all possible app executions, can be overly conservative hence potentially highly imprecise. Results based on static code analysis may further suffer unsoundness due to common code traits in apps that impede the analysis (e.g., obfuscation and dynamic language constructs) [32, 4]. Also, previous evolution studies for Android were conducted either on a few samples only (a few hundreds) while focusing on the evolution of particular apps (i.e., the multiple versions of an app) without differentiating benign and malicious apps [33, 34, 35] or on benign apps [36] or malware only [37, 38]. Most of these studies looked at the high-level external behaviors (e.g., data leaks, permission uses, network traffic, etc.) [39, 34, 35], instead of looking into the internal code structure from a software engineering perspective, and/or spanned a relatively short period of time (e.g., one year only [39]).

As it stands, it remains unclear how Android apps evolve in their *dynamic* behaviors *as a population* over a long period of time. In particular, understanding how malware evolves differently from benign apps is essential for long-term security defense of the Android ecosystem (e.g., for developing a long-span, sustainable malware detector [40]). Our recent effort on evaluating the sustainability (i.e., the ability to *sustain* high accuracy) of state-of-the-art malware detectors [7, 6, 8] preliminarily showed that more sustainable approaches tended to have used features that differentiated benign and malicious apps more resiliently against the evolution of both groups. Yet, a principled discovery of such features has yet to be explored.

In this context, we investigate the run-time behavioral evolution of benign apps versus malware in Android with a focus on app execution structure. While dynamic characterization is generally subject to limited code coverage, it reveals the real behaviors actually *observed* hence complements static characterization. Specifically, we characterize the execution structure in terms of various app functionality scopes (user code, Android framework, and third-party libraries) and their interaction, distribution of component types and communication between components, and callback use extent and categorization. All these dynamic measures are computed via the full scope of method calls and inter-component communication (ICC) exercised during app executions. We chose these specific measures as motivated by our previous study [36] that found them to well capture behavioral differences between benign apps and malware of a particular year. Also, concerning that capturing *explicit* malicious behaviors (e.g., sensitive data flows [41]) is not always feasible because sophisticated malware may hide their true behaviors at runtime [42] to evade detection, we chose the structural measures which are not *explicitly* associated with obvious, *known* malicious behaviors.

By examining the execution traces of 30,634 apps, including 15,451 benign apps and 15,183 malware as group samples (rather than evolved versions of the same apps), throughout the past eight years, we performed in-depth investigations of how malware evolves differently from benign apps in terms of various measures of execution structure. Through this extensive dynamic evolution study, we address the following research questions, with corresponding major findings summarized below. These questions are integral parts of an umbrella question of *how does malware behave differently from benign apps in terms of code-level execution structures?*, consistent with our goal with this study of understanding the run-time behavioral differences between the two app groups in terms of such structural traits.

- **How does malware exercise its functionalities in varied functionality scopes differently from benign apps?** We found that generally the execution of benign apps is similarly distributed over the three high-level functionality scopes (user code, third-party libraries, and the Android SDK), with similarly little yet still shrinking portion of user code exercised, to malware. However, malware made calls to SDK mainly via third-party libraries, as opposed to benign apps making such calls from within the SDK.
- **How does malware execute different types of components differently from benign apps?** Our results reveal that while the execution of both benign apps and malware was dominated by user-interface (*Activity*) components, the use of such components was steadily shrinking in benign apps whereas in malware the use was on rise after two periods of declination. Compared to benign apps, malware executed significantly larger portions of *Services*, but smaller portions of *Content Providers*. Also, malware made larger portions of ICCs connecting to external (built-in) apps. Although both groups did not carry any data in most ICCs, data-carrying ICCs in malware transferred data via standard-data fields significantly less often than benign apps.
- **How does malware use callbacks of various categories differently from benign apps?** We found opposite trends in overall callback usage

between benign apps and malware during their evolution. In particular, compared to the gradual reduction in their run-time invocation of callback in benign apps, malware has seen drastic growth in executing callbacks, as mainly attributed to the rise in using lifecycle callbacks. Meanwhile, rankings of callback categories were similar and generally stable over time between benign apps and malware, yet rank differences were shrinking in malware.

We also discussed implications of our findings to app testing and security analysis. To the best of our knowledge, this is the first, longest-spanning study focusing on the evolution of *dynamic* behaviors of benign apps versus malware in terms of code-level execution structures. Our study dataset and utilities have been made publicly available (as found here [43]) to the community to facilitate replication and support further studies. The toolkit for the study has been made publicly earlier [44], with detailed usage documentation available as well [45]. The dataset comes with necessary details for obtaining the apps themselves, and the toolkit is accompanied by informative usage documentation.

2. Background

This section gives background on the Android framework and its applications to facilitate understanding our study.

Android framework and callbacks. The middle layer between the Android OS (a customized Linux kernel) and its user applications constitutes the Android framework. This framework provides the implementation of application programming interface (API) methods through which user apps can receive system services and invoke common functionalities associated with mobile devices. The API is typically part of the Android software development kit (i.e., *SDK*) which also includes tools to support app development. The user apps are event-driven and interact with the framework often via *callbacks* implemented in their user code, including those dealing with various events (i.e., *event handlers*) and those for the framework to manager app lifecycles (i.e., *lifecycle methods*).

App components and ICC. Under the framework-based development paradigm, Android apps usually comprise building blocks called *components* of four types: *Activity* forming the basis of user interface, *Service* performing background tasks, *Broadcast Receiver* responding to system-wide broadcasts, and *Content Provider* offering database capabilities. The Android framework defines a set of lifecycle methods for each top-level class corresponding to each of these component types and additionally for the class `android.app.Application`. Inter-component communication (i.e., *ICC*) is the primary means for components within (i.e., *internal* ICC) the same app and across apps (i.e., *external* ICC) to exchange messages via message objects called *Intents*. The target of ICC may be explicitly specified (i.e., *explicit* ICC) or left unspecified for the framework to resolve at runtime (i.e., *implicit* ICC).

3. Scope and Research Questions

Our overarching aim is to understand *how the execution structure of malware evolves differently from that of benign apps.* Importantly, we focus on the differences that have security relevance and implications. Thus, we compare benign apps and malware as two large groups, disregarding (1) functional differences (e.g., between game apps and musical apps) and (2) differences within each group (e.g., among benign apps of different kinds/categories or among malware of varied families). In particular, we examine the execution structure of apps by profiling all method calls during their executions. Further, we express the structure in terms of the following three groups of dynamic measures.

- **Functionality scopes (code layers).** In general, an Android application package (APK) may contain three high-level scopes (layers) of app functionalities at runtime: user code (*userCode*, i.e., any code the app developer actually wrote), Android libraries (*SDK*, i.e., framework APIs), and third-party libraries (*3rdLib*, i.e., any libraries other than the SDK used by the app). We measure execution structure through the distribution of executed methods over these three code layers according to which layer each of these methods is defined in. We further measure calls across layers.

- **Components.** Android apps follow a general modular design, implementing four high-level categories of functionalities (user interface, background service, system communication, and data management) in the four different types of components (Activity, Service, BroadcastReceiver, and ContentProvider, respectively). In addition to code layer distribution and interaction, we examine app execution structure also through how apps execute code in each of these types of components and how app components communicate through ICC of varied types (*implicit* versus *explicit* and *internal* versus *external*). We further looked into the data fields in the Intent of exercised ICCs, concerning the three ways an ICC may carry data: only via *standard* URI (the `data` field of the Intent), only via *bundle* (the `extras` field of the Intent), or both (two Intent fields carrying data).

- **Callbacks.** Due to their event-driven programming paradigm, Android apps feature common use of callbacks through which apps leverage the SDK capabilities and the Android platform communicates with apps. We thus measure the overall callback usage and distribution of such usage over the two major kinds of callbacks: *lifecycle methods* and *event handlers*. We further examine the use of each kind over its major categories. Specifically, we categorize lifecycle methods according to the five top-level enclosing classes defined in the SDK: *Activity, Service, BroadcastReceiver, ContentProvider, Application.* A lifecycle method is categorized into one of these categories based on the rationale that the method is one of those that the framework uses to manage the lifecycle of the corresponding type of component or the entire app. For event handler callbacks, we consider five types of user interfaces (UIs) associated with (*App bar*, *Media control*, *View*, *Widget*, *Dialog*), and five types of system events handled (*App management*, *System status*, *Location status*, *Hardware management*, and *network management*). An event handler is categorized into one of these categories as it is the callback that the framework invokes when the UI or system event occurs. Our choice of these types of events is informed by a previous study that found them as the top five *most-frequently* exercised UI/system events [36].

The above measures *define* the execution structure that we characterize in our evolution study of Android app behaviors. With this definition, we approach our overarching question through the following three specific research questions.

- **RQ1:** How does malware exercise its functionalities in varied functionality scopes differently from benign apps?
- **RQ2:** How does malware execute different types of components differently from benign apps?
- **RQ3:** How does malware use callbacks of various categories differently from benign apps?

Note that these three questions are essentially three integral parts of our central (umbrella) question that guides the study in this paper—how does malware behave differently from benign apps in terms of code-level execution structures? We look into run-time behaviors of malware and benign apps as embodied by the execution structures of apps from three aspects: functionality scope, communication among app components, and callbacks. Accordingly, each of the three questions focuses on one of these closely related aspects of app behaviors. Thus, the three questions are tightly connected, addressing the umbrella question consistently.

4. Methodology

This section describes our experimental methodology, including datasets used and study procedure followed.

Table 1: Number of benchmarks used in our study

Year	2010	2011	2012	2013	2014	2015	2016	2017
# benign apps	1,530	2,019	2,053	1,748	3,127	1,333	1,548	2,093
# malware	2,029	2,431	2,225	1,230	1,493	1,667	2,171	1,937

4.1. Benchmarks

To investigate how execution structure in benign and malicious Android apps evolve, we collected real-world app samples from varied sources throughout the past eight years (2010–2017). Table 1 gives an overview of the app samples used in our study. Most of the samples were obtained from AndroZoo [46], except for that malware of years 2013 through 2016 was from VirusShare [47] and benign apps of year 2017 were from Google Play [48] to diversify the data sources for possibly better sample representativeness. The year of each sample was determined based on the DEX date and *versionCode* extracted from the app's APK [49]. For each year, we begun with a larger pool of samples and discarded those that did not meet our two selection criteria: (1) the app is dynamically analyzable—apps corrupted or with missing assets were dismissed, so were those that cannot be instrumented or launched, (2) exercising the app with random inputs generated by the Monkey tool [50] for 10 minutes did not cover at least 60% user code of the app (in terms of line coverage).

For our dynamic study, applying a coverage criteria is necessary. We set the 60% threshold in an attempt to cover a majority of the app such that the app executions we analyzed in the study represent a reasonable portion of the common operational profile of the app. It is important to note that this representativeness is essential: since we aim to understand the behavioral differences of malware from benign apps, exercising the benign and malicious behaviours is a key to the validity of our study results. On the other hand, we could not automatically check if these behaviors (especially those of malware) are sufficiently exercised—automatically capturing and validating the exhibition of malicious behaviors is still an open research problem in general [42]. Nevertheless, a relatively high code line coverage as our threshold enforces should give enough confidence about the behaviors we need to characterize being covered.

A few research prototypes of automated input generation techniques exist, which may reach the threshold coverage faster or higher coverage in 10 minutes. Our choice of using Monkey over research prototypes [51, 52] was made primarily due to the much greater robustness and usability of this industry-strength tool and its relative small shortage of coverage compared to the prototypes. Also, main relevant research prototypes only support relatively old Android (e.g., Dynodroid [51] and Sapienz [52] work for Android 4.4 or earlier versions), with which many of our samples cannot be installed/executed. Two recent studies further justified the use of Monkey [53, 54].

The numbers of Table 1 were the actual numbers of apps used in our study, after applying the two selection criteria. In total, we analyzed the execution of 30,634 apps, including 15,451 benign apps and 15,183 malware. We did not intentionally select an equal number of apps for all the eight years to respect the uneven distribution of the total app populations over the years. We also had removed redundant apps within and across years whenever applicable, such that only unique samples were considered and any two of our yearly benign/malware datasets are disjoint. Each sample was confirmed as malware or benign using the VirusTotal [55] service—the app was considered malware if at least ten of the anti-virus tools on VirusTotal identified it as so, otherwise it was considered benign. For all the samples eventually used in our study, the profiling with random inputs had line coverage ranging from 60% to 100% (mean 74.85%, standard deviation 11.97%).

4.2. Execution Profiling

To profile each benign and malware sample, we performed purely application-level instrumentation to trace method calls and ICC Intents. Our scope of tracing includes all method and ICC calls, including those made through exceptional control flows and reflection. The 10-minute per-app trace was produced by running the instrumented APK on a Google Nexus One emulator with the Android SDK 6.0 (API level 23), 2G RAM, and 1G SD storage, with inputs fed by Monkey. The emulator itself ran on a Ubuntu 15.04 host with 8G memory and a dual-core 2.6GHz processor. We utilized our Android app characterization toolkit [44] for these profiling tasks. In the end, 30,634 traces, each for a sample, formed the basis of our evolution study. To avoid possible side effects of inconsistent emulator settings, we started the installation and execution of each app in a fresh clean environment of the emulator (with respect to built-in apps, user data, and system settings, etc.).

4.3. Characterization

To characterize the execution structure of apps and their evolution over the eight-year span, we computed the dynamic measures (Section 3) for each app separately from its trace. More specifically, we first build a *dynamic call graph* from the trace, where each node represents a method/ICC call and each edge represents a dynamic call which is annotated with the frequency (i.e., number of instances) of that call. For an ICC call, the corresponding node contains additionally the Intent field values. Based on this call graph, our dynamic measures were mostly computed as a percentage (of certain kind of calls over the calls in a larger class). The only exception was for callback categorization, for which we rank the categories for each app according to the percentage of callback invocations belonging to each category and report for each category the mean rank across all benchmarks in a dataset. The categories of lifecycle callbacks were decided through a class hierarchy analysis, and those of the event handlers were recognized in reference to the callback interface categorization we developed earlier. More implementation details can be found in [44].

4.4. Metrics and Measurements

It is important to note that, given the goal of our study, we focus on evolutionary trends of apps' execution structure rather than the absolute values of our measures—these absolute values would vary with different samples studied. To compare malware with benign apps, we adopted an average-case analysis. Thus, with respect to each measure, we typically compute the average over all apps in the benign or malware group of a particular year, and then compare the two groups in terms of the averages.

Beyond these average-case comparisons, we also computed the statistical significance of differences between benign-app and malware groups with respect to each structural trait of apps (e.g., percentage of external implicit ICC) involved in our studies. We used the paired Wilcoxon signed-rank tests [56] to assess the significance at the 0.95 confidence level (i.e., $\alpha = .05$). To understand the magnitude of those differences, we further computed corresponding effect sizes in terms of Cliff's Delta [57] (in a paired setting with $\alpha = .05$). In both analyses, the two groups compared were the average-case metric values of the two app groups across the eight years (i.e., each group has eight values). Both analyses are nonparametric, making no assumption about the normality of the distribution of underlying data points. For each app trait (i.e., dynamic measure used in the characterization), we typically compare benign apps against malware regarding that trait first in a plot, followed by the Wilcoxon p values (noted as p) and Cliff's Delta (noted as Δ) presented in a table. Given a Cliff's Delta value d, we interpret the effect size as follows [58]: effect size is *negligible* if $|d| \leq 0.147$, *small* if $0.147 < |d| \leq 0.33$, *medium* if $0.33 < |d| \leq 0.474$, and *large* if $|d| > 0.474$.

5. Major Findings

In this section, we present and discuss our study results, as guided by the three main research questions.

5.1. RQ1: Functionality Scope Distribution and Interaction

This research question examines how benign and malicious apps exercise their behaviors in the three layers of functionality (i.e., *UserCode*, *SDK*, and *3rdLib*), as well as how these layers collaborate (interact) in order to fulfill the app functionalities.

5.1.1. Execution Composition

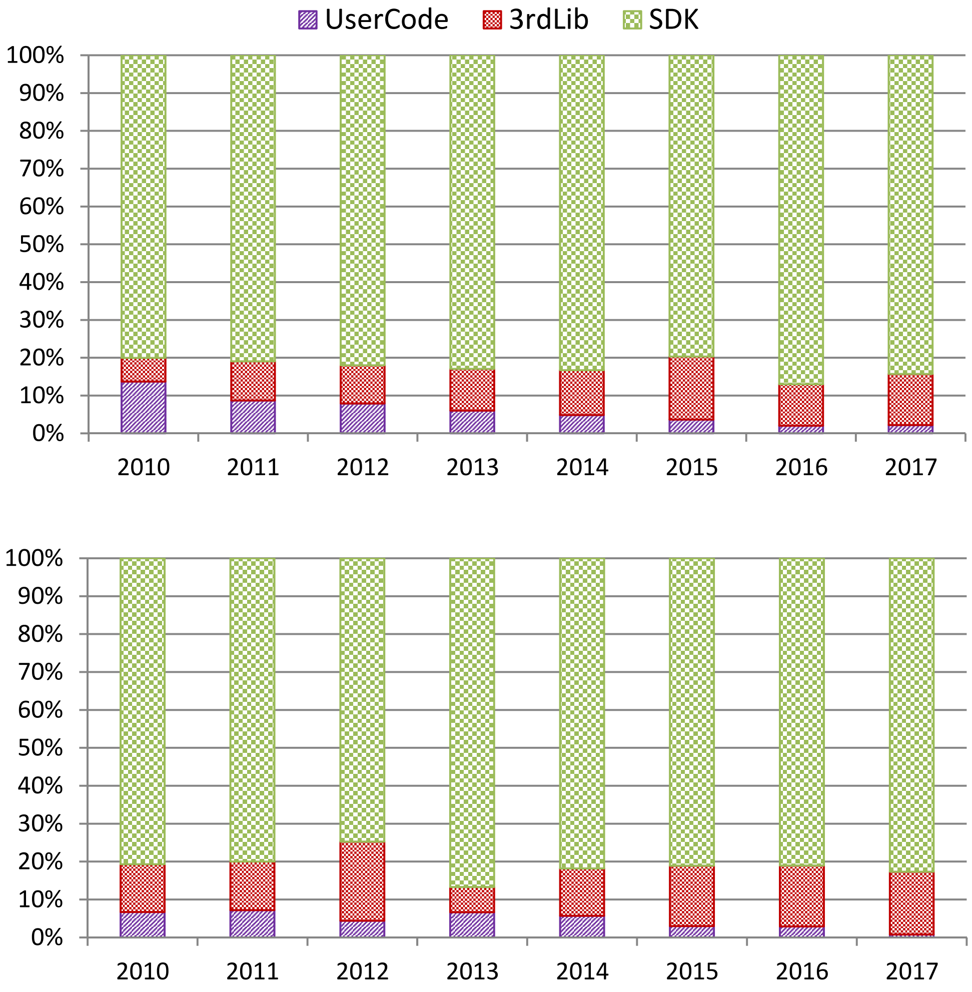
The composition of an app's execution is characterized through the percentage distribution of all method call instances over the three code layers. This is essentially done by uplifting each node of the app's dynamic call graph to its enclosing code layer and then computing the code-layer distribution by counting nodes in each layer.

Figure 1 shows the distribution of these layers between benign apps and malware across the eight-year span. In both benign and malware apps, the vast majority (over 80%) of methods called were defined in the SDK, and this dominance remained almost constant in the past eight years. It is known that most of methods invoked by benign apps are Android SDK APIs [36]. Our results here revealed largely the same role of the Android framework in malware executions.

A closer look reveals that the average percentage of benign-app execution in SDK had an overall slight growth while the percentage slightly declined in malware (especially since year 2013). As a result of this trend, third-party libraries were increasingly (albeit also in small increments) used in malware and decreasingly in benign apps. On the other hand, the fluctuations were at best unsubstantial, resulting in the generally stable dominance of the Android framework during the executions of both app groups.

A relatively clear trend was that user code kept shrinking in app executions, despite the security of the apps, although the magnitude of decrease over the entire eight-year span was slightly greater in benign apps. This trend implies that developers of Android apps, benign or malicious, tend to write less and less code for building an app, possibly because of the continuous enrichment of features offered by the Android platform.

Overall, functionality scope distribution in malware was not significantly different from that in benign apps, as our statistics (*St*) showed (bottom of Figure 1).



St	UserCode	3rdLib	SDK
p	0.1953	0.1484	0.3125
Δ	-0.2500	0.5000	-0.2500

Figure 1: Functionality scope distribution (y axis) of benign apps (top plot) and malware (bottom plot).

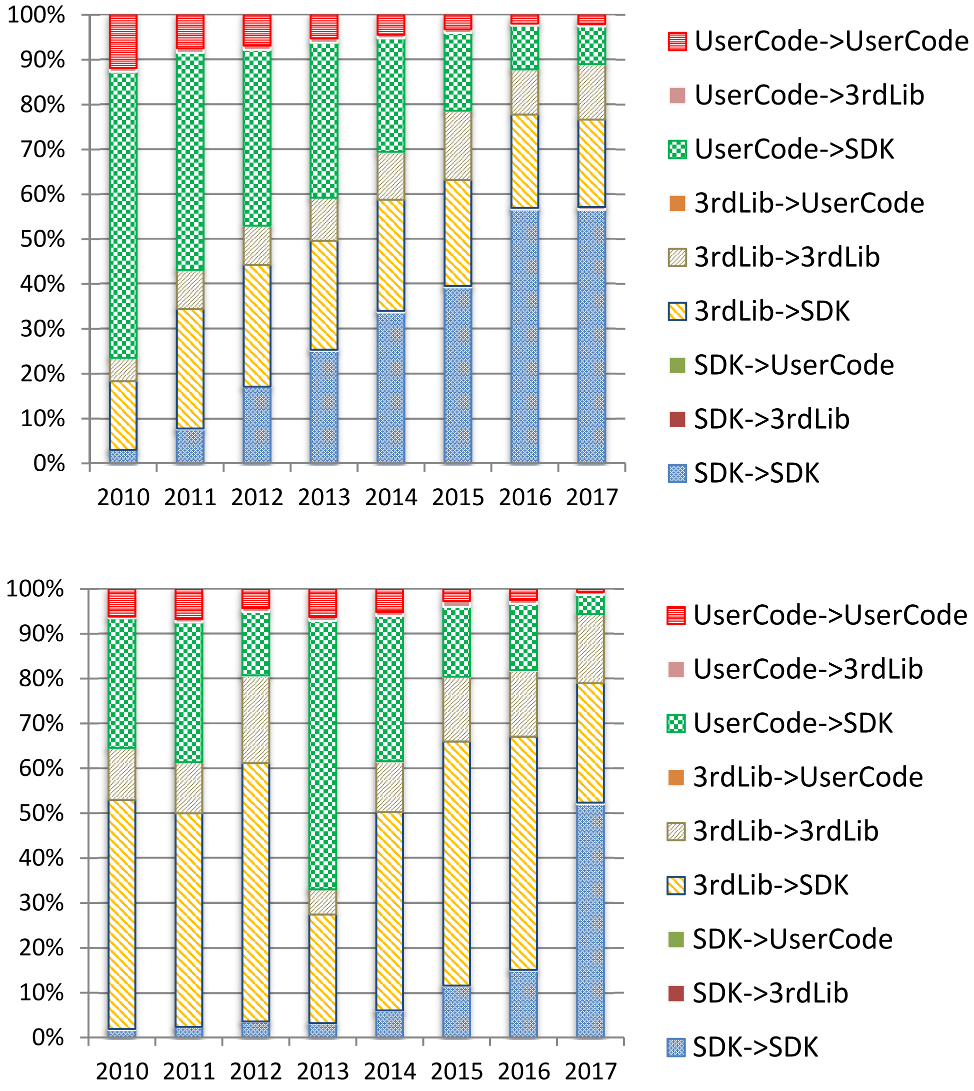
Finding 1: *Benign apps and malware had very similar functionality distribution over user code, third-party libraries, and the Android SDK, which was also consistent over time. In both app groups, user code accounted decreasingly, and SDK stably, for exercised app functionalities.*

5.1.2. Cross-Scope Collaboration

The method-level calling relationships we profiled reveal collaboration among the three functionality layers/scopes, when we uplift each node of the dynamic call graph to its enclosing scope/layer and then count the edges. This collaboration within each app indicates the internal dynamics of the app with respect to its use of library functionalities through the user code. We characterize the cross-scope collaboration via the distribution of the dynamic calls (i.e., edges) on the graph over the nine possible calling relationships among the three scopes. Note that we focus on characterizing call pairs (i.e., a caller and a direct callee), rather than call chains (e.g., a user-code method calls a third-party library method which then calls back to user code)—with respect to *direct* calling relationships, the call chains are subsumed by the call pairs we characterize.

Figure 2 depicts the percentage distribution of all instances of calls over the nine cross-scope calling relationships in benign apps versus malware. In terms of general evolutionary trends, both benign apps and malware share the overall decrease in calls *within* user code and the overall increase in calls within the SDK. These increases and decreases essentially changed the dominating cross-scope calls from user code to SDK (in year 2010) to within the SDK (in year 2017) in benign apps, and dominance of *3rdlib→SDK* calls to calls within the SDK in malware. In both groups, calls within the SDK gradually gained dominance, but even more abruptly in the malware group. Some minor calling relationships were not quite visible on the chart because of their negligible portion in the distribution (e.g., *SDK→UserCode* calls in both benign apps and malware).

Moreover, benign apps saw steady decrease in calls to the SDK launched from user code, yet this change experienced much greater variations in malware.



St	SS	S3	SU	3S	33	3U	US	U3	UU
p	**0.039**	0.195	**0.039**	**0.008**	0.148	0.547	0.641	0.742	0.461
Δ	**-0.750**	-0.750	**-0.750**	**1.000**	0.500	0.000	-0.250	0.000	-0.250

Figure 2: Cross-scope calling relationship distribution (y axis) of benign apps (top plot) and malware (bottom plot).

Also, there were always much greater portions of *3rdlib→SDK* calls, but always much smaller portions of *SDK→SDK* in malware than in benign apps. This observation was consolidated by our p and Δ statistics that showed *significant* and *large* differences (highlighted in boldface) between benign and malware apps with respect to these two types of cross-scope interaction. Along with the results on functionality scope distribution, these differences imply that, although both groups were similarly dominated by calls targeting SDK APIs, malware tended to make such calls more via third-party libraries while benign apps tended to do so more within the Android framework. Our statistical analysis revealed that percentages of calls from SDK to user code were significantly and largely higher in benign apps than in malware.

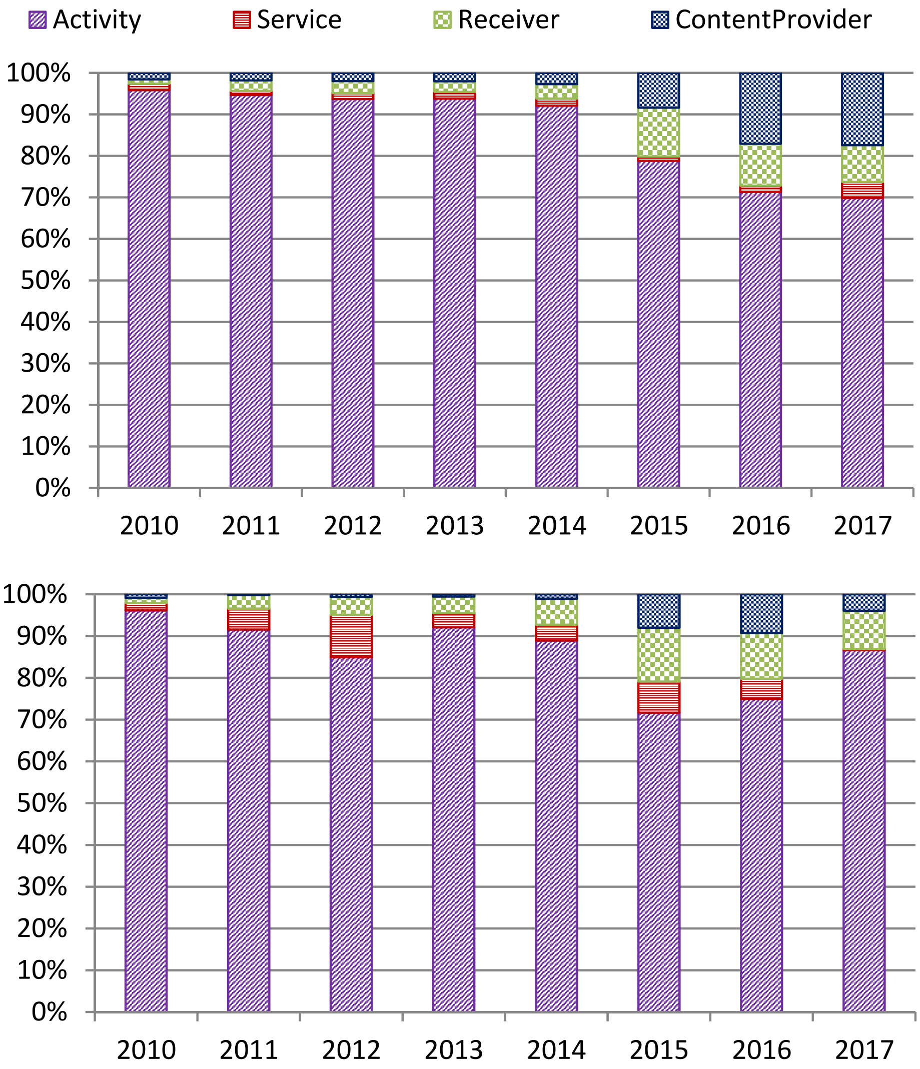
Finding 2: *Over time, both benign apps and malware had decreasing calls within user code and increasing calls within the SDK, yet malware had significantly less calls to user code from SDK. Constantly, malware had more calls to SDK from third-party libraries than did benign apps, yet benign apps had more such calls within the Android framework.*

5.2. RQ2: Component Distribution and Communication

This research question concerns the distribution of app executions over the four types of components. In essence, these four component types represent separation of concerns in app design. Thus, this question investigates the composition of app executions from a semantic perspective (e.g., how much code dealing with user interface was executed versus code for data management), which is complementary to studying the execution composition with respect to the three functionality scopes/layers in RQ1.

5.2.1. Component Distribution

Figure 3 compares benign apps with malware concerning how an app's execution is structured with respect to the four component types. Both benign apps and malware had rich user interface exercised via *Activity* components, which constantly dominated over other types of components over the years. At



St	Activity	Service	(Broadcast) Receiver	ContentProvider
p	0.6406	**0.0078**	0.0781	**0.0078**
Δ	-0.2500	**1.0000**	0.7500	**-1.0000**

Figure 3: Distribution of execution over the four component types (y axis) in benign apps (top) versus malware (bottom).

least 70% of all exercised components were *Activities*. Also common to both groups was overall growing portions of *Content Provider* and *Broadcast Receiver* components, albeit not monotonically.

Compared to the steady decrease in the use of *Activity* components in benign apps, the trend in malware was more of a zigzag shape—the use of *Activities* in malware had continuous drop during the years 2010–2012 and years 2013–2015 periods, followed a steady growth since 2015. Meanwhile, use of *Content Providers* was constantly more prevalent, and also growing faster, in benign apps, whereas *Service* components were used more substantially in malware. The p and Δ values highlighted that these two differences were significant and large statistically. One plausible explanation for these differences is that *Service* components effectively provide a latent, collaborative environment for committing malicious behaviors as they perform long-running operations in the background (i.e., without user interface); thus they are preferably exploited by malware to launch attacks in an unnoticeable and reliable manner. *Content Provider* components are more prevalently used in benign apps than in malware, likely because legitimate functionalities tend to more rely on dedicated data management than do malicious behaviors.

Finding 3: *For the execution of* Activity *components, malware experienced two diminishing periods, followed by a recent rise, as opposed to a continuous reduction in benign apps. Over time, malware used significantly more* Services*, but less* Content Providers*, than benign apps.*

5.2.2. Component Communication

Communication between app components informs about the internal dynamics of apps with respect to how semantically different code regions interact. Since the component-level communication in apps is realized through ICC APIs, we characterize such communications through a particular kind of method calls—calls targeting ICC APIs. Given our focus on app execution structure, we study how components connect within individual apps and how components interact across apps. We limited our study focus not to include

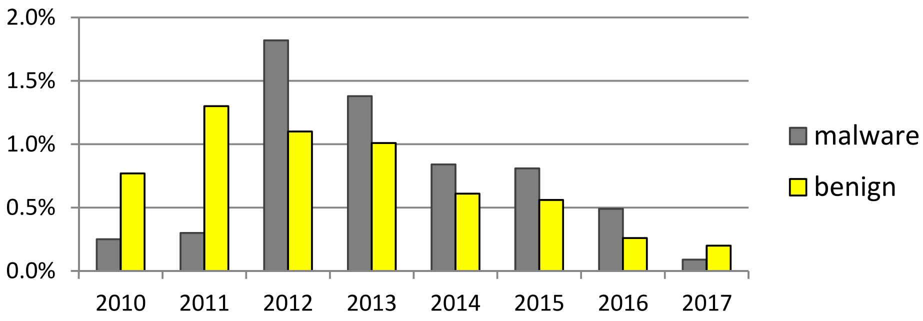
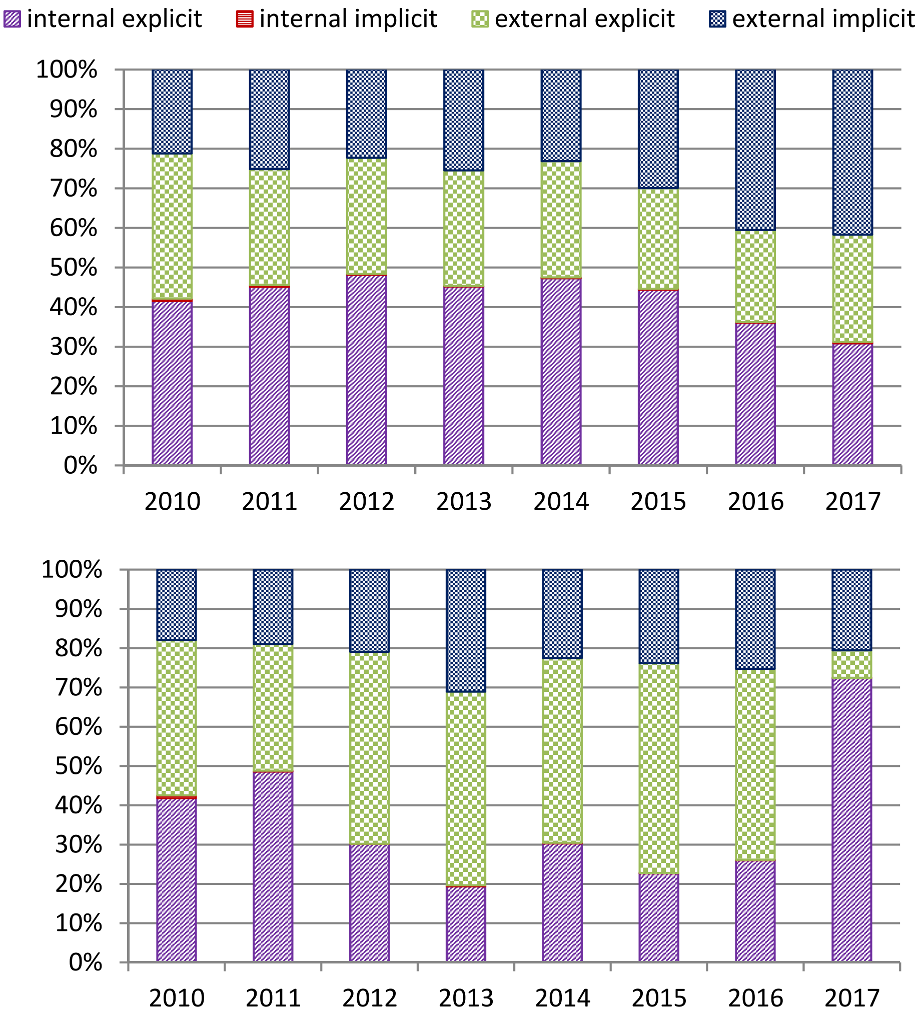


Figure 4: Percentage of ICC calls over all method calls in benign apps versus malware. (p=0.742, Δ=0.250)

inter-app communication, yet there were built-in apps in Android (e.g., `Camera`, `Photo Viewer`, etc.) which may communicate with our sample apps when they were exercised. Thus, *external* ICCs did present in our app executions. We are also concerned about the messages transferred between components—the data carried by the ICC Intents.

Counting all ICC API calls during app executions, Figure 4 reveals that despite the varying absolute numbers of such calls, ICC calls were only a tiny portion of all method calls. In both benign app and malware, the percentage of ICCs of any types was at most 1.7% across the past eight years. What is also noteworthy is that total use of ICCs in both benign apps and malware experienced a rise in the period of year 2010 to 2011, and dropped continuously ever since. This can be explained by the possible movement that newer apps, regardless of their security orientation, tend to enhance their maintainability/changeability by improving their cohesion while reducing coupling, at least at component level; thus, individual components are increasingly independent of other components to fulfill their functionalities, resulting in less needs for ICCs. Also, in absolute terms, malware exercised ICCs more frequently than benign apps did. However, the differences between the two groups were not significant at all, as per the p and Δ values.

Figure 5 shows how ICCs in benign apps and malware were distributed



St	int. exp.	int. imp.	ext. exp.	ext. imp.
p	0.3828	0.2500	0.1094	0.0547
Δ	-0.2500	-0.2500	0.7500	-0.7500

Figure 5: Distribution of component communication over the four types of ICCs in benign apps (top) versus malware (bottom).

over the four ICC types (noted in the legend). This result reveals that malware mostly had greater use of *external explicit* ICCs, but lesser use of *external implicit* and *internal explicit* ones, than benign apps. This implies that when communicating with built-in apps, malware tended to do so more often explicitly while benign apps more often implicitly. Yet, none of these differences were significantly large as the p and Δ values indicated. In both malware and benign apps, components within apps rarely communicated implicitly, hence the negligible portion of *internal implicit* ICCs.

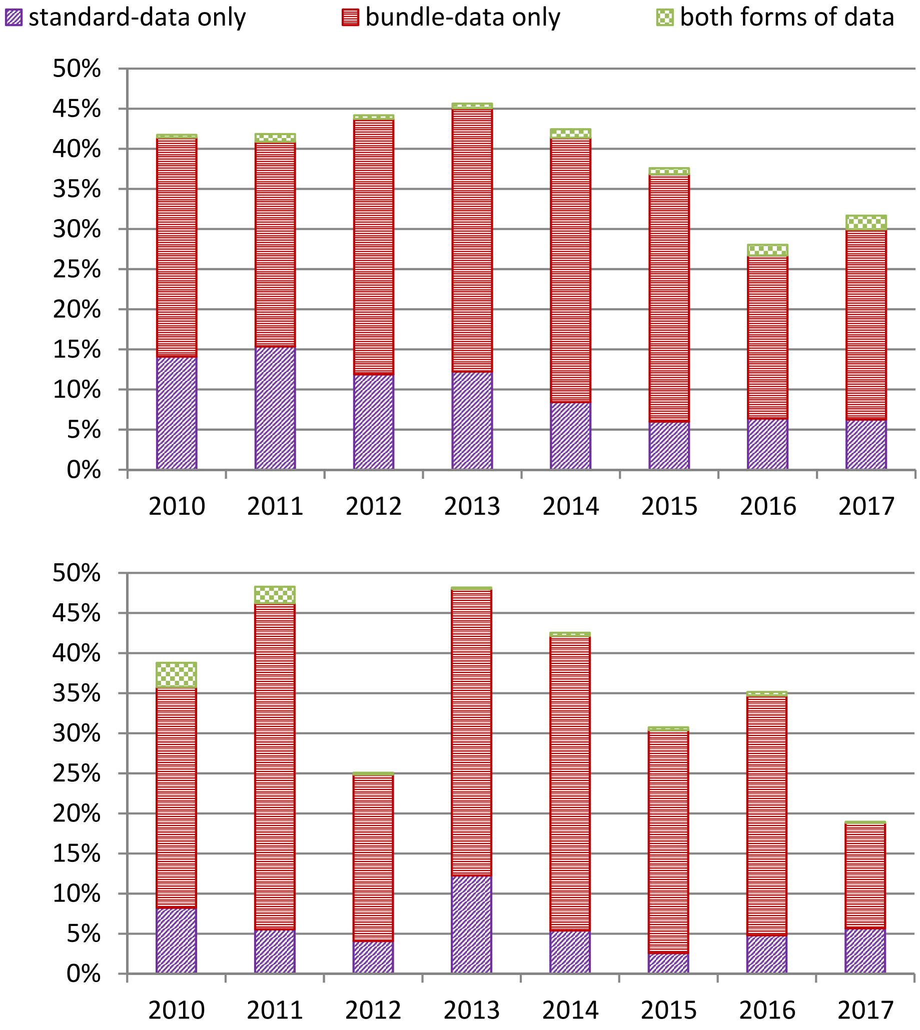
Finding 4: *Overall use of ICCs has been steadily dropping, but always with similarly small portions of all method calls, in both malware and benign apps. Over the years, malware used mostly larger portions of explicit, but lesser of implicit, external ICCs than benign apps did.*

We further characterize component communication by looking into the ways ICCs carried data, if any. In particular, we examine the total percentage of ICCs that carried data, and among data-carrying ICCs the distribution over the three ways (see Section 3) of doing so. Figure 6 shows the overall reduction (albeit slow) in carrying data in ICC Intents by any means, in both benign and malicious apps. Among the ICCs that did carry any data, bundle data has been consistently the primary means, in both groups. Very few ICCs carried data in both the `data` and `extras` fields of their Intent objects, though. Our p and Δ values show that the evolution of the two groups on ICC data transfer was only significantly different in carrying standard data only.

Finding 5: *Like benign apps, malware did not transfer any data in most of their ICCs, and if doing so bundles were always the preferred means. Over the years, benign apps had significantly larger portions of data-carrying ICCs that contain standard data than did malware; yet both groups had seen decreasing total invocation of data-carrying ICCs.*

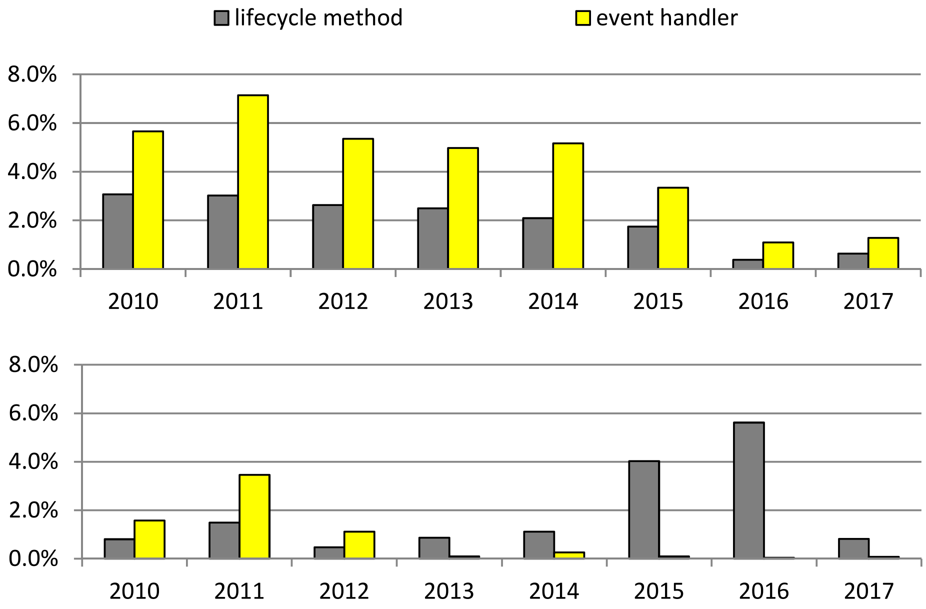
5.3. RQ3: Callback Use and Categorization

Through this research question, we intended to investigate how Android apps invoke callbacks of various kinds during their execution, and in particular



St	standard-data only	bundle-data only	both forms of data
p	**0.0156**	0.8438	0.6406
Δ	**-0.7500**	0.2500	-0.5000

Figure 6: Distribution of component communication over the three ways of carrying data in ICCs in benign apps (top) versus malware (bottom).



St	lifecycle method	event handler
p	0.1953	**0.0078**
Δ	-0.7500	**-1.0000**

Figure 7: Overall run-time invocation of callbacks (y axis) in benign apps (top) versus malware (bottom).

attempted to characterize the evolution of callback use in malware versus in benign apps.

5.3.1. Extent of Use

Figure 7 depicts the overall use of callbacks in benign apps and malware, measured as the total percentage of exercised callbacks over all methods called during app executions. The changes in the callback use over the years were plotted separately for the two kinds of callbacks: lifecycle methods and event handlers. The result shows the overall small portion of callbacks among all method calls (at most 7.2%) in any of the eight years studied, despite the security groups of the apps. This indicates that relative to the total amount of method calls, callbacks were not so frequently exercised, albeit more than ICC calls (see Figure 4). This is consistent with our prior observations on the execution composition in terms of functionality scopes (of Figure 2): the portion of calls from SDK to user code, which correspond to callback invocations, was almost negligible. Comparatively, however, benign apps exercised callbacks more prevalently than malware, which is again consistent with the significant larger portions of *SDK→UserCode* calls in benign apps seen in Figure 2.

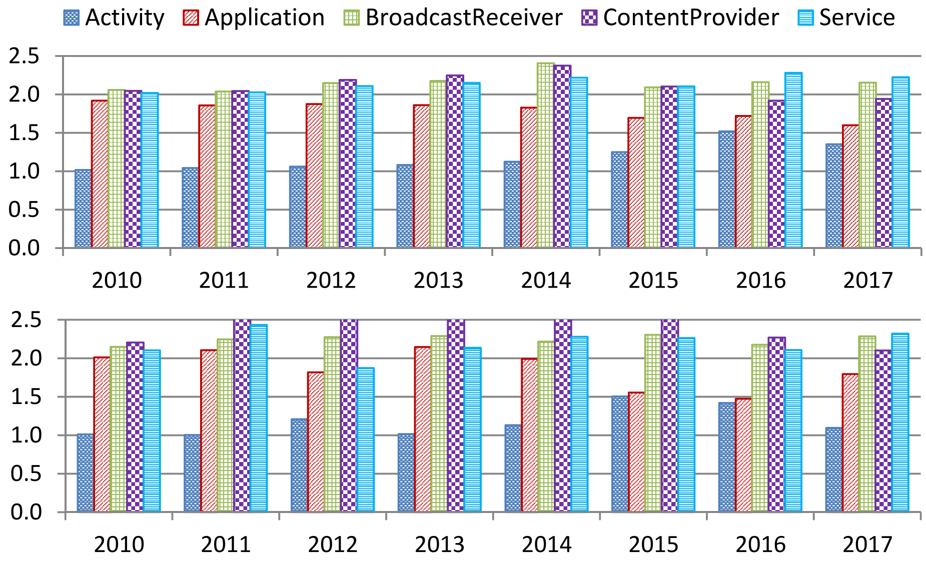
In terms of the evolution differences between the two groups, the generally steady decrease in both kinds of callbacks in benign apps is opposed to the overall drastic increase in the invocation of lifecycle callbacks in malware, till year 2016. Event-handling callbacks in malware were exercised decreasingly over the eight-year span, and consistently less than in benign apps—the gap was significant and large as per the p and Δ values. In terms of total amount of callback usage, malware saw general growth while benign apps saw gradual declination.

Finding 6: *Callbacks were not very frequently invoked in malware, nor in benign apps. Consistently, malware executed event handlers significantly less often than did benign apps. Overall, total callback execution was on the decline in benign apps, but generally on the rise in malware.*

5.3.2. Callback Distribution

For a closer look into callback usage, we further examine the distribution of invoked callbacks over the major callback categories, for lifecycle methods and event handlers separately (Section 3).

Figure 8 presents the ranks of the top-five lifecycle callback categories in benign apps and malware. One clear observation is that in both app groups, *Activity* was constantly the top category, meaning the highest percentage of lifecycle callbacks were always attributed to managing the lifecycle of *Activity* components. This is not surprising given our observation from Figure 3 that this type of components constantly dominated in the component distribution



St	Activity	Application	Service	ContentProvider	B. Receiver
p	0.6406	0.3125	0.1094	**0.0078**	0.6406
Δ	-0.2500	0.2500	0.7500	**1.0000**	0.2500

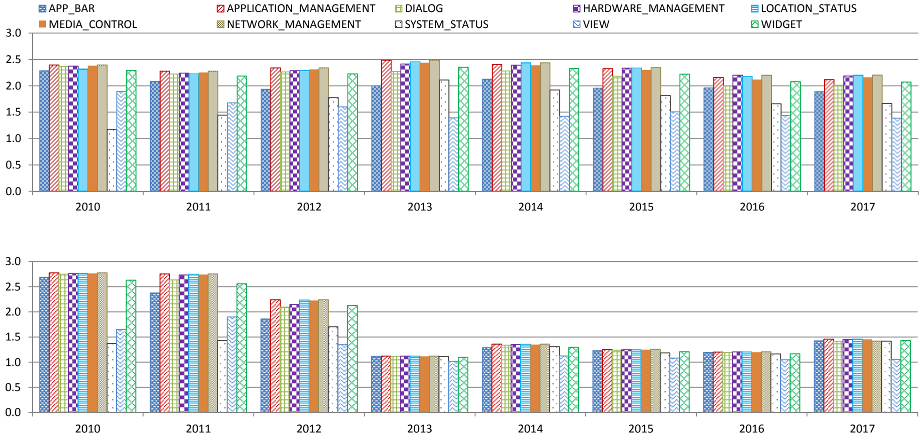
Figure 8: Distribution of exercised lifecycle callbacks over the five categories in benign apps (top) versus malware (bottom).

of app executions. The second highest percentage of lifecycle callbacks were invoked due to the framework's management of the app as a whole. The top two ranks were always taken by *Activity* and *Application*, in any given year and in both benign apps and malware. The ranking of other three categories, however, varied slightly across years in both groups, with *Content Providers* being the least frequently exercised lifecycle callback type in most years. Also, the differences in the ranks of these three categories were generally small in absolute terms.

Overall, the evolutionary pattern of lifecycle categorization was similar between benign apps and malware. Our statistical analysis results (p and Δ) pinpointed that, during the eight-year evolution, invocation of callbacks associated with *Content Providers* in malware was significantly and largely lesser (ranked lower) than in benign apps (greater rank value means lower rank). This is consistent with, and can be explained by, the finding from our results on component distribution (Figure 3) that the malware samples executed lower percentages of components of this type than did benign apps with statistical significance and large effect sizes. However, although the malware executed significantly higher percentages of *Services*, callbacks for managing these components' lifecycle were not significantly different between malware and benign-app executions. The reason was that many of the exercised methods in *Service* components were not lifecycle callbacks.

Finding 7: *Lifecycle callback distribution was largely similar between malware and benign apps, with* Activity *callbacks constantly dominating and the overall category ranking being consistent over time. Yet malware invoked significantly less* Content Provider *callbacks than benign apps.*

The distribution of event-handling callbacks over the five selected categories of UI events and five selected categories of system events is depicted in Figure 9 in a similar format to Figure 8. Except for the first two years of the studied span, *View* was generally the most handled UI events, followed by *AppBar* and then by *Dialog*, in both benign app and malware. Among system events, *System status* was consistently the top category, also in both groups. Other categories



St	sys. st.	view	app_bar	widget	loc. st.	dialog	hard. mgt.	media ctl.	app. mgt.	net. mgt.
p	**0.039**	**0.016**	0.078	0.078	0.078	0.078	0.078	0.078	0.078	0.078
Δ	**-0.750**	**-0.750**	-0.500	-0.500	-0.500	-0.500	-0.500	-0.500	-0.500	-0.500

Figure 9: Distribution of exercised event handlers over the ten major categories in benign apps (top) versus malware (bottom).

were not as consistently distinguishable with noticeable rank differences.

In terms of evolution, the ranking of these event-handling callback categories was largely stable over time in benign apps. In malware, however, the rank differences among the categories were generally decreasing over time. Overall, the difference in the ranking evolution between benign apps and malware was mostly very small for most categories. For the top system and UI events (i.e., *System status* and *View*, respectively), the ranks were significantly lower in benign apps than in malware, as per relevant p and Δ values. (For ease of presentation, we abbreviated in the table the category names with the first two or three letters of each word in the full names as shown in the legend of Figure 9.)

> **Finding 8**: *Event-handling callback ranking was generally stable over time in both groups. While the rank differences among event-handler categories were largely constant in benign apps, the ranks had become hardly differentiable in malware. Also, malware had significantly higher ranks than benign apps for the top system- and UI-event callbacks.*

6. Lessons Learned

In this section, we discuss the implications of our major empirical findings to both app testing and security analysis.

On functionality scope. Our evolution study on app execution distribution over functionality scopes revealed that benign apps and malware were largely the same (without significant differences) in both the distribution and the evolutionary patterns over the eight-year span. This similarity (*Finding 1*) suggests that learning-based malware detector may not benefit much from using features that characterize how apps execute user code relative to the execution of library code. Thus, *relevant features (e.g., the percentage of user-code or third-party-library calls over all method calls) should be avoided in learning-based malware classification as they would confuse the classifier hence downgrade its malware detection performance.* Meanwhile, the steady drop in user-code involvement in app executions indicates the promise of prioritizing user code in app testing and security defense (for better cost-effectiveness), assuming that the framework itself is more secure and less defective. Moreover, this prioritization strategy can be increasingly justifiable given the increasing portion of app executions being carried out through calls within the SDK and shrinking interaction between user code and libraries (e.g., calls from user code to SDK). However, run-time activities within third-party libraries remain a standing, substantial portion of app executions, confirming the necessity of security screening of third-party libraries for a holistic app security defense solution.

In contrast to benign apps, malware tended to make calls to the SDK via third-party libraries much more substantially than benign apps did (*Finding 2*), likely due to malware intention of impeding detection based on framework-based characteristics (e.g., features based on usage of APIs). In contrast, benign apps' execution of framework functionalities was mostly bounded within the framework. Thus, securing benign apps relies more on the security of the framework itself, while malicious behaviors of malware are more likely to be rooted in the insecurity of third-party libraries. Accordingly, *third-party libraries' traits (e.g., percentage of method calls originated and targeting these libraries over all method calls) can be modeled as features for more effective malware detection.*

On app components. Despite the general declination in exercising *Activities* in both benign app and malware executions (possibly due to the increasing use of web content in newer apps), *Activity* components remain the dominant type among all exercised components. Thus, user interfaces are still, and likely continue to be in the future, a major attack surface in Android apps. In particular, the fact that *Activity* execution in malware rose again in recent years after experiencing two decreasing periods (*Finding 3*) implies that securing users against attacks via user interfaces need to be emphatically attended. More (less) intensive use of *Content Providers* (*Services*) in benign apps than in malware, as well as that of associated callbacks, could be sustaining differentiators to be leveraged by dynamic malware detectors, given the significant and large differences in the use of these components (*Finding 3*) and the use of associated callbacks (*Finding 7*) between malware and benign apps. Accordingly, *app features that characterize the use of these components (e.g., the percentage distribution of executed components over varied component types) may be used by the detectors based on supervised learning.*

ICC is the primary communication channel within and across Android apps, hence has been considered a major attack surface in Android. However, our observations on ICC use during app executions indicate its continuous decrease in both benign apps and malware (*Finding 4*). A potential consequence of this trend is that securing ICC as the major means of security apps [59] is likely to have a diminishing return. Meanwhile, app testing focusing on covering ICCs might suffer from relatively low overall coverage. The finding that both newer benign apps and newer malware tended to exercise less ICCs than before implies that major attack surface might have been shifted to other points of the apps. On the other hand, malware had noticeably higher frequency of exercising ICCs than benign apps did, suggesting ICCs might not be entirely dismissed as an attack surface in security defense in the near future. In addition, ICC-based malware detector should pay more attention to the possible manipulation of built-in apps by malware, given the steadily substantial portion of ICCs being external ones. However, since there were no consistent and substantial differences between benign app and malware in terms of data payloads in ICCs (*Finding 5*), whether the ICCs carry data or not might not be a good indicator of malicious behaviors. Meanwhile, the observation that data-carrying ICCs delivering the data less often via standard data fields can be utilized as a promising differentiator between benign apps and malware that has not yet been leveraged for malware detection. Thus, *features characterizing such preferences (e.g., the percentage of ICCs that carry standard data over all exercised ICCs) can be part of the feature embeddings for apps in dynamic malware detector.* Note that these ICC-based features are relative statistics (i.e., percentages), clearly different from those used in prior work that are based on concrete ICC Intent fields (e.g., [59]). Also, prior work in this line typically uses static approaches (e.g., parsing the Intent information from apps' manifest files and/or source code), as opposed our ICC-based features being purely dynamic.

On callbacks. Analysis of callbacks in Android apps has proven to be highly expensive [41, 60]. However, our dynamic evolution study revealed that callbacks were not very frequently invoked, neither in benign apps nor in malware (*Finding 6*). Thus, it would be cost-beneficial to fully track callback data/control flows for fine-grained security analyses and app testing. On the other hand, our finding that malware had increasing (while benign apps had decreasing) use of callbacks, especially significantly less frequent invocation of event handlers than in benign apps, suggest potential additional means for differentiating malware from benign apps. Thus, *relevant features (e.g., the frequency of event-handling callbacks relative to total method calls) would strengthen the performance of a dynamic malware detector*. Meanwhile, with fewer callback executions in benign apps, precise taint analysis is expected to scale better to newer benign apps than to old ones, given that callback analysis was known to be a major performance barrier in such analyses.

The steady dominance of callbacks for managing *Activities* among all lifecycle callbacks (*Finding 7*) implies the lasting merits of focusing on *Activity* callbacks in callback control flow analysis [60]. However, lifecycle callback distribution over varied categories may not be useful for detecting malware, given the consistently similar ranking between the two groups over time. Therefore, *relevant features characterizing this distribution should be avoided in the design of a dynamic malware detector*. However, a much lower percentage of invocation of *Content Provider* callbacks could point to malware behaviors. Another good indicator of malware is the great closeness of ranks among all event-handler categories, and/or significantly higher ranks of callbacks for handling *View* and *System status* events (*Finding 8*).

Importantly, as we discuss the security implications of our empirical findings, there are two caveats to keep in mind. First, how effective the features learned from the study would be for sustainable malware analysis depends on how stable the evolutionary patterns in malware behaviors and their differences from benign behaviors will be in the future. Only when the patterns are reasonably stable, would the features effectively contribute to the long-span capabilities of the techniques built on those features. Second, the motivation of this paper is not to identify features once for all that can work for future effective security defense solutions forever, but rather to explore and demonstrate how to discover such features. Realistically, these features, albeit relatively more sustainable (than, for instance, features extracted from a set of apps in a time-agonistic manner), would expectedly become outdated after some time. Yet, the methodology might still be valid and applicable for discovering other, more discriminatory features for future newer malware and benign apps.

7. Threats to Validity

Threats to internal validity. We used our toolkit to trace the sample apps studied and compute the dynamic measures that define the execution structure we focus on in this study. While it handles reflective calls and calls via exceptional control flows, this toolkit does not monitor native calls or calls buried in dynamically loaded code. Thus, the method calls and ICCs we analyzed could be incomplete. Moreover, although we have applied a non-trivial coverage threshold when selecting sample apps to make sure the app executions we studied reasonably represent a major portion of common app behaviors, the random test inputs from Monkey we used might have missed representative app execution paths. As a result, our characterization might have not always captured the typical execution structure of some sample apps, which could be more concerning for malware samples since malware is known to be able to hide their behaviors especially when being executed on an emulator. With a small set of malware, we performed our comparative study with a real device versus the emulator and did not see significant differences in terms of our dynamic measures.

Notably, our study focuses on structural characteristics of malware (and benign apps), rather than explicit security traits (e.g., access of sensitive data, known malicious instances/sequences of calls to suspicious functions, etc.). The metrics used in our characterization have shown to be able to capture the behavioral differences between malware and benign apps [36, 61], despite the known evasiveness of malware and the impediments of varied obfuscation schemes adopted in both malware and benign apps. This implies that our metrics can *implicitly* capture the *essence* of app behaviors. Yet, to more convincingly reduce this threat, a more extensive verification would be required.

Threats to external validity. While we purposely managed to choose a sizable dataset for benign apps and malware for each year, our yearly benchmark suite size is small relative to the entire app population of each year. More importantly, it is not clear how our chosen benchmarks are representative of respective populations. Consequently, our evolution study results may not be generalizable to other apps, nor to the entire app population per year. Thus, the presented observations and findings are best interpretable with respect to the particular apps we chose and studied. Particularly, an additional threat to external validity comes from the uneven distribution of our yearly benign-app and malware set. We used substantially more samples of certain years (e.g., benign apps of year 2014) than others. Our rationale was that the total app population in the real-world more likely than not varies in size year by year as well, thus choosing the same numbers of samples for different years may not respect the reality. Meanwhile, this poses a validity threat because we do not know the accurate sheer totals of apps in each year, thus the unevenness of the distribution of our yearly datasets may not represent the plausible unevenness of the distribution of actual app population sizes across years in reality.

Threats to construct validity. Given our focus on evolutionary patterns rather than absolute values of dynamic measures, we derived our findings mainly on an average-case basis—we compared benign apps and malware in terms of the average values of the chosen dynamic measures. Although the standard deviations (not reported in paper) were generally small relative to the mean values, more thorough statistical analyses would be more desirable to corroborate our conclusions. In particular, the behaviors of malware (with respect to certain metrics) may have considerably different *distribution* from those of benign apps (with respect to the same metrics) while still having very similar summary statistics (e.g., the average metric values). Thus, our current study might have missed some behavioral differences between the two app groups. More in-depth data analysis (including that of data distribution) could have revealed even more interesting and important findings. These further examinations are indeed part of immediate future work following this study.

Threats to conclusion validity. To corroborate the practical usefulness of our study findings, we discussed how the empirical results on the contrasts between benign apps and malware in their behavioral differences can be used for sustainable malware detection. Such benefits of our results, however, are limited to the behavioral characteristics we actually considered in this study. Thus, for malicious behaviors that are not relevant to any of our characterization metrics, our results would not be directly applicable for detection purposes. For instance, a ransomware working on SD cards would not be detected by solely using the app features we studied since its maliciousness is not exhibited through execution-structural differences from benign apps. Similarly, our recommendations and lessons learned from the empirical results for the optimizations/prioritizations of app analyses are also limited to those that are relevant to the app characteristics addressed in our longitudinal study.

8. Related Work

Concerning the comparison between benign apps and malware in Android and their characteristics or behaviors, our study presented in this paper is not the first. Nevertheless, our study clearly distinguished itself from prior peer work in multiple ways, including *the nature of study*, *study scope*, and *study perspectives*; and it contributes to the understanding of applications in Android in a distinct manner hence complements existing studies in this domain. In this section, we elaborate how this paper is situated in the relevant literature, while highlighting the key differences between current similar studies and ours.

Nature of study. Previous characterization studies of Android apps mostly focused on *static characteristics* of apps in terms of metrics extracted from the apps' code. For example, in [62], 1,100 popular apps were studied to understand their use and misuse of private information of mobile phones and users through reverse engineering. For another example, in [63] the authors studied code reuse in Android apps in varied ways, including reusing individual classes (e.g., through inheritance) and framework-wise reuse. A follow-up study further investigated how code obfuscation and the use of third-party libraries can affect the results of code reuse studies for Android apps [64]. **None of these studies examined the evolution of apps, though, compared to our study featuring a longitudinal and evolutionary lens into app characterization**.

A few earlier studies did characterize the evolution of Android apps. The study in [30] focused on the evolution of Android in terms of the permissions provided by the Android platform and their use by apps and third-party libraries. In [31], the authors studied the updates of APIs during the Android SDK evolution as well as API usage evolution in dependent apps, and discovered update rate of Android APIs and lags in apps' according update of their use of those updated APIs thus characterized the API stability in the Android ecosystem. The study in [65] looked into the versioning, source size, and third-party library usage of 20 sample apps to understand the differences between Android apps and traditional software applications. Researchers have also studied the evolution of apps in terms of the presence of anti-patterns [33], by looking into a few thousands of versions of a hundred of sample apps. Yet **these studies are still static in nature, compared to our characterization being purely dynamic**—instead of looking at static artifacts (e.g., code, manifest files, etc.) of apps, we solely focus on examining the run-time executions of apps.

Study scope. A few dynamic characterization studies exist, which concern the installation and activation methods of malware only [66] or execution structure of benign apps only [67, 36]. Other examples include the dynamic study [68] that profiles apps in terms of their inter-component communications (ICC) and network traffic. CopperDroid [69] was used for characterizing system calls in apps, and the toolkit in [44] serves both static and dynamic characterizations. Other recent studies [38, 19, 70, 37] are also related to ours but they focused exclusively on malware only. In contrast to **the prior work studying either benign apps or malware only, our study looks into both benign and malicious apps and, more importantly, examines the differences between these two groups and implications of these differences**. Our own latest empirical work on characterizing Android apps, with a focus on compatibility issues [28, 29], is also limited to benign apps.

Study perspectives. The prior studies of different scope and nature from ours as discussed above also did not study **the execution structure at code level and from a programming perspective** (e.g., the callback mechanism, interactions among code layers, etc.) as we do in this paper.

The study in [34] characterized 50 or more releases of 235 different apps in terms of their static (sensitive data leaks and permission uses) and dynamic (network traffic) traits. Similarly, Ren et al. studied how 512 unique Android apps (each with on average 15 versions) evolved over eight years in terms of their leakage of personal identifiable information (PII) [35]. Another study took a close look into the use of dangerous permissions in multiple snapshots of Google Play store within a year [39] to examine how permission uses evolved. **Unlike these prior studies** which focus on a few *particular* and *external* behavioral traits of apps (e.g., permission uses, network traffic, PII leaks), our study looked into **the *internal* structure of app executions** from a programming point of view with respect to the **underlying dynamics within apps** that may be exhibited as *a large variety* of external behavioral traits.

Also, a focus of our study is to reveal the security relevance of evolutionary traits (e.g., execution structure as we studied in this work) by separately looking at the characteristics of benign apps versus those of malware and contrasting them with respect to the characteristics. In comparison, previous peer studies tended to mostly conflate benign apps and malware in their study benchmark suites, without differentiating malware and benign apps. The characterization of evaluation datasets in [9] reveals the evolutionary characteristics of both malware and benign apps also, but it instead focuses exclusively on API calls in apps and is static.

In summary: compared to peer prior work, our study (1) *characterizes the run-time behaviors of apps*, (2) *examining both benign and malicious apps, both separately and comparatively*, (3) *in terms of code-level execution structures and from programming perspectives*, while (4) *applying a long-span (eight-year), evolutionary lens* at a relatively larger scale and with a broader scope. These distinctions clearly differentiate this paper from the current relevant literature. Also, with the unique combination of study nature, scope, and perspectives as discussed above, our study complements to the existing body of work on characterizing user applications in Android.

9. Conclusion

We presented a longitudinal study of Android apps that characterizes the behavioral evolution of benign apps versus malware in terms of execution structures. By monitoring the method calls and ICC Intents of 30.634 apps developed throughout the past eight years, we measured the *actually observed* behaviors of these apps to compare the dynamic traits and evolutionary patterns between the two groups. Our study revealed a number of previously unknown behaviors shared by both groups, as well as those that drastically differentiate malware from benign apps consistently over time. Our findings offer novel insights to enhancing future app testing and sustainable security analysis techniques.

References

[1] I. D. C. I. Research, Android dominating mobile market, http://www.idc.com/promo/smartphone-market-share/ (2016).

[2] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, L. Ziarek, Rtdroid: A design for real-time android, IEEE Transactions on Mobile Computing 15 (10) (2016) 2564–2584.

[3] Android malware accounts for 97% of all malicious mobile apps, http://www.scmagazineuk.com/updated-97-of-malicious-mobile-malware-targets-android/article/422783/ (2015).

[4] D. J. Tan, T.-W. Chua, V. L. Thing, et al., Securing Android: a survey, taxonomy, and challenges, ACM Computing Surveys 47 (4) (2015) 1–45.

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

[6] H. Cai, J. Jenkins, Towards sustainable android malware detection, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ACM, 2018, pp. 350–351.

[7] H. Cai, Longitudinal characterization and sustainable classification of android apps via SAD profiles, CoRR abs/1807.08221. arXiv:1807.08221. URL http://arxiv.org/abs/1807.08221

[8] X. Fu, H. Cai, On the deterioration of learning-based malware detectors for Android, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2019, pp. 272–273.

[9] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, G. Stringhini, MAMADROID: Detecting android malware by building markov chains of behavioral models, in: Proceedings of Network and Distributed System Security Symposium, 2017.

[10] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, Riskranker: scalable and accurate zero-day Android malware detection, in: Proceedings of ACM International Conference on Mobile Systems, Applications, and Services, 2012, pp. 281–294.

[11] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani, Crowdroid: behavior-based malware detection system for Android, in: Proceedings of ACM workshop on Security and privacy in smartphones and mobile devices, 2011, pp. 15–26.

[12] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, X. Wang, Effective and efficient malware detection at the end host., in: Proceedings of USENIX Security Symposium, 2009, pp. 351–366.

[13] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, "Andromaly": A behavioral malware detection framework for Android devices, Journal of Intelligent Information Systems 38 (1) (2012) 161–190.

[14] H. S. Galal, Y. B. Mahdy, M. A. Atiea, Behavior-based features model for malware detection, Journal of Computer Virology and Hacking Techniques (2015) 1–9.

[15] A. Saracino, D. Sgandurra, G. Dini, F. Martinelli, Madam: Effective and efficient behavior-based Android malware detection and prevention, IEEE Transactions on Dependable and Secure Computing.

[16] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, L. Cavallaro, DroidSieve: Fast and accurate classification of obfuscated android malware, in: Proceedings of ACM Conference on Data and Application Security and Privacy, 2017, pp. 309–320.

[17] S. Chen, M. Xue, Z. Tang, L. Xu, H. Zhu, Stormdroid: A streaminglized machine learning-based system for detecting Android malware, in: Proceedings of ACM Asia Conference on Computer and Communications Security, 2016, pp. 377–388.

[18] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, 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.

[19] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, 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.

[20] Y. Aafer, W. Du, H. Yin, DroidAPIMiner: Mining API-level features for robust malware detection in Android, in: SecureComm, 2013, pp. 86–103.

[21] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, Drebin: Efficient and explainable detection of Android malware in your pocket, in: Proceedings of Network and Distributed System Security Symposium, 2014.

[22] M. Zhang, Y. Duan, H. Yin, Z. Zhao, Semantics-aware Android malware classification using weighted contextual api dependency graphs, in: Proceedings of ACM Conference on Computer and Communications Security, 2014, pp. 1105–1116.

[23] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, E. Bodden, Mining apps for abnormal usage of sensitive data,

in: Proceedings of IEEE/ACM International Conference on Software Engineering, 2015, pp. 426–436.

[24] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, Y. Le Traon, et al., Empirical assessment of machine learning-based malware detectors for android, Empirical Software Engineering 21 (1) (2016) 183–211.

[25] L. Wu, M. Grace, Y. Zhou, C. Wu, X. Jiang, The impact of vendor customizations on android security, in: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, 2013, pp. 623–634.

[26] L. Wei, Y. Liu, S.-C. Cheung, Taming android fragmentation: Characterizing and detecting compatibility issues for android apps, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 226–237.

[27] D. He, L. Li, L. Wang, H. Zheng, G. Li, J. Xue, Understanding and detecting evolution-induced compatibility issues in android apps, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 167–177.

[28] H. Cai, Z. Zhang, L. Li, X. Fu, A large-scale study of application incompatibilities in Android, in: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2019.

[29] Z. Zhang, H. Cai, A look into developer intentions for app compatibility in Android, in: IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft), 2019.

[30] X. Wei, L. Gomez, I. Neamtiu, M. Faloutsos, Permission evolution in the Android ecosystem, in: Proceedings of Annual Computer Security Applications Conference, 2012, pp. 31–40.

[31] T. McDonnell, B. Ray, 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.

[32] A. Moser, C. Kruegel, E. Kirda, Limits of static analysis for malware detection, in: Proceedings of Annual Computer Security Applications Conference, 2007, pp. 421–430.

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

[34] P. Calciati, K. Kuznetsov, X. Bai, A. Gorla, What did really change with the new release of the app?, in: Proceedings of the 15th International Conference on Mining Software Repositories, ACM, 2018, pp. 142–152.

[35] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, N. Vallina-Rodriguez, Bug fixes, improvements, ... and privacy leaks—a longitudinal study of pii leaks across android app versions, in: Proceedings of Network and Distributed System Security Symposium, 2018.

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

[37] F. Mercaldo, A. Di Sorbo, C. A. Visaggio, A. Cimitile, F. Martinelli, An exploratory study on the evolution of android malware quality, Journal of Software: Evolution and Process 30 (11) (2018) e1978.

[38] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, 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.

[39] V. F. Taylor, I. Martinovic, To update or not to update: Insights from a two-year study of android app evolution, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM, 2017, pp. 45–57.

[40] H. Cai, Assessing and improving malware detection sustainability through app evolution studies, ACM Transactions on Software Engineering and Methodology (TOSEM).

[41] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, 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.

[42] S. Rasthofer, S. Arzt, S. Triller, M. Pradel, Making malory behave maliciously: Targeted fuzzing of Android execution environments, in: Proceedings of IEEE/ACM International Conference on Software Engineering, 2017.

[43] Code and dataset on evolution of benign apps versus malware, `https://www.dropbox.com/s/2s05kalgkcqwchn/andro-evo-supplement.zip?dl=0` (2019).

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

[45] H. Cai, Systematic dynamic characterization of Android apps, `http://chapering.github.io/droidfax/` (2017).

[46] K. Allix, T. F. Bissyandé, J. Klein, Y. Le Traon, Androzoo: Collecting millions of android apps for the research community, in: Proceedings of Working Conference on Mining Software Repositories, 2016, pp. 468–471.

[47] virusshare.com, VirusShare, https://virusshare.com/ (2017).

[48] Google, Google Play Store, https://play.google.com/store?hl=en (2017).

[49] Google, Version your app, https://developer.android.com/studio/publish/versioning (2018).

[50] Google, Android Monkey, http://developer.android.com/tools/help/monkey.html (2017).

[51] A. Machiry, R. Tahiliani, 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.

[52] K. Mao, M. Harman, 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.

[53] P. Patel, G. Srinivasan, S. Rahaman, I. Neamtiu, On the effectiveness of random testing for android: or how i learned to stop worrying and love the monkey, in: Proceedings of the 13th International Workshop on Automation of Software Test, ACM, 2018, pp. 34–37.

[54] M. Mohammed, H. Cai, N. Meng, WIP: An empirical comparison between Monkey testing and human testing, in: ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2019, pp. 188–192.

[55] virustotal.com, VirusTotal, https://www.virustotal.com/ (2017).

[56] R. E. Walpole, R. H. Myers, S. L. Myers, K. E. Ye, Probability and Statistics for Engineers and Scientists, Prentice Hall, 2011.

[57] N. Cliff, Ordinal methods for behavioral data analysis, Psychology Press, 1996.

[58] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen's d indices the most appropriate choices, in: Annual meeting of the Florida Association of Institutional Research, Citeseer.

[59] K. Xu, Y. Li, R. H. Deng, ICCDetector: ICC-Based malware detection on Android, IEEE Transactions on Information Forensics and Security 11 (6) (2016) 1252–1264.

[60] S. Yang, D. Yan, H. Wu, Y. Wang, 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.

[61] H. Cai, N. Meng, B. Ryder, D. D. Yao, Droidcat: Effective Android malware detection and categorization via app-level profiling, IEEE Transactions on Information Forensics and Security (TIFS).

[62] W. Enck, D. Octeau, P. McDaniel, S. Chaudhuri, A study of android application security, in: Proceedings of the 20th USENIX conference on Security, 2011, pp. 21–21.

[63] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, A. E. Hassan, A large-scale empirical study on software reuse in mobile apps, IEEE software 31 (2) (2013) 78–86.

[64] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, 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.

[65] R. Minelli, 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.

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

[67] H. Cai, B. Ryder, Understanding application behaviours for android security: A systematic characterization, Computer Science Technical Reports.

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

[69] K. Tam, S. J. Khan, A. Fattori, L. Cavallaro, Copperdroid: Automatic reconstruction of android malware behaviors., in: Proceedings of Network and Distributed System Security Symposium, 2015.

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