Mobile Application Coverage: The 30% Curse and Ways Forward

Faridah Akinotcho
Univ. of British Columbia, Canada
faridah.akinotcho@alumni.ubc.ca

Lili Wei
McGill University, Canada
lili.wei@mcgill.ca

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

***Abstract*—Testing, security analysis, and other dynamic quality assurance approaches rely on mechanisms that invoke the software under test, aiming to achieve high code coverage. A large number of invocation mechanisms proposed in the literature, in particular for Android mobile applications, employ GUI-driven application exploration. However, studies show that even the most advanced GUI exploration techniques can cover only around 30% of a real-world application. This paper aims to investigate "the remaining 70%". By conducting a large-scale experiment involving two human experts, who thoroughly explored 61 benchmark and 42 popular apps from Google Play, we show that achieving a substantially larger coverage for real-world applications is impractical even if we factor out known GUI-based exploration issues, such as the inability to provide semantic inputs and the right order of events. The main reasons preventing even human analysts from covering the entire application include application dependencies on remote servers and external resources, hard-to-reach app entry points, disabled and erroneous features, and software/hardware properties of the underlying device. Thus, future investment in GUI-based exploration strategies is unlikely to lead to substantial improvements in coverage. To chart possible ways forward and explore approaches to satisfy/bypass these "blockers", we thoroughly analyze code-level properties guarding them. Our analysis shows that a large fraction of the blockers could actually be successfully bypassed with relatively simple beyond-GUI exploration techniques. We hope our study can inspire future work in this area; it also provides a realistic benchmark for evaluating such work.**

***Index Terms*—Testing, Mobile applications, Empirical studies**

I. Introduction

In a variety of quality assurance scenarios, an independent auditor is required to inspect applications (apps) written by a third party. For example, Google Play Store analysts review applications submitted to the store, to ensure they meet the quality and security standards. Likewise, auditing companies inspect third-party apps to identify security and privacy violations. Such inspections often involve dynamic analysis and, thus, rely on triggering the app under test.

To help trigger a large number of third-party applications at scale, numerous GUI-based automated app exploration techniques have been recently proposed [1]–[22]. Most of these techniques generate user interface and system-level events to trigger the application under test, simulating the ways users interact with the application and device. For example, the most basic yet most prominent of such techniques – Android Monkey [1], which is part of the standard Android distribution, generates pseudo-random streams of GUI events, such as clicks, touches, or gestures, to drive the execution of the application. Other techniques extend Monkey by selecting and prioritizing the generated events, aiming to maximize code coverage and/or bug detection frequency for the application under test.

With the continuous emergence of new and improved GUI-based exploration strategies, a number of studies also compare the capabilities of the emerging tools on a set of benchmark applications [23]–[29]. The results of these studies show that even the most advanced tools are able to cover only around 30% of an industrial-scale application. They suffer from several limitations, such as difficulties to produce semantically-meaningful inputs (e.g., only zip codes in country-specific formats are accepted) and order of events (e.g., the need to add items to the shopping cart before checking out).

This paper deals with the question of why application exploration techniques "hit a ceiling" of 30% and what the possible ways forward are. To answer these questions, we factor out known difficult-to-resolve issues, such as semantic inputs and order of events, by employing two skilled human analysts to perform a thorough manual exploration of a large number of third-party apps. We treat these results as an intelligent GUI-based exploration "tool". As subject applications, we use popular Android benchmark applications [23] and a set of 42 large and popular applications from Google Play. Aligned with the third-party application analysis goal, our analysts explored the applications in a black-box fashion, assuming only public knowledge about the apps.

We also run two prominent and available GUI-based exploration tools on the same set of subject applications: APE [13], which was identified as one of the most performant tools in several earlier studies [27], [29], and the most recent publicly available tool – FastBot2 [21].

When comparing the coverage results obtained by automated vs. manual exploration, we were surprised to observe that humans could exceed the activity-level coverage achieved by the tools only by a small fraction. Specifically, the average coverage achieved by both tools combined is 22.4%, while the coverage achieved by humans is 28.1%. Moreover, the combined coverage of all tool and human runs is 29.6%, on average. While we analyze the differences in automated and manual results later in the paper, more importantly, these results imply that further improving GUI-based exploration is unlikely to result in a substantial increase in application coverage.

To understand the reasons for "the missing 70%", we further reverse-engineer and manually analyze a subset of our case study applications, mapping out the reasons and code-level properties preventing GUI-based exploration from reaching the remaining activities. To the best of our knowledge, our work is the first to perform a detailed analysis of applications' code, on a large set of popular Google Play applications, to explain the low coverage results. It thus differs from the majority of comparative studies that report on the limited abilities of GUI-based exploration tools [23]–[30], without looking at code-level application properties.

The results of our analysis show that dependencies on remote servers and external resources, as well as hard-to-reach app alternate entry points, disabled and erroneous features, and the device software/hardware properties, are among the main reasons preventing even human analysts from exploring a large portion of an application. Luckily, we also observe that more than 20% of the unreached activities can actually be triggered with relatively simple beyond-GUI exploration techniques that can start the activities programmatically. Another nearly 25% of the "blockers" can be bypassed with techniques that generate data parameters, albeit requiring the generation of String and Object data types. We hope our study will inspire development of such techniques, enabling a substantial breakthrough in the efficiency of automated app exploration. Our work also contributes a new benchmark of real-world applications, which can be used to effectively evaluate and compare the techniques.

Data Availability. To support further work in this area, our data and analysis results are available online [31].

II. Background

In this section, we provide the necessary background on the Android platform and on automated app exploration techniques.

Android Platform. Android is an open-source Linux-based software stack created for a wide range of devices [32]. Android-native applications are written in Java or Kotlin and are compiled into Dalvik Executable (DEX) bytecode. The bytecode is further packaged with resource files to form an Android Application Package (APK).

The Android Java API Framework (or, simply, the framework) provides the interface between applications and lower-level layers of the Android stack, such as Android Runtime, Native C/C++ Libraries, Hardware Abstraction Layer, and Linux Kernel. The framework exposes the entire feature set of the Android OS to apps through APIs written in Java. These APIs form the building blocks applications use to generate user interfaces, manage data and resources on the device, and manage the application lifecycle. Framework APIs can largely be divided into *callbacks* and *methods*. Callbacks serve as entry points into the apps; they are overridden by an app to implement custom responses to events and are invoked by the framework when a particular event occurs. For example, a *Button::onClick* callback is invoked when a button is clicked. The framework also invokes callbacks when the state of the device changes, e.g., the network is disconnected, or when the application lifecycle state changes, e.g., on creation or destruction. Unlike callbacks, API methods are invoked by the apps directly, to access functionalities exposed by the framework. For example, *TelephonyManager::getDeviceId* is a framework method used to obtain the unique identifier of the underlying device.

Android apps are built from four types of components: activities, services, broadcast receivers, and content providers. Activities serve as the primary interaction points between the app and the user. Each app contains a main activity, which serves as the main entry point into the app. Services execute long-running background tasks, e.g., playing music while the user is interacting with another activity. Broadcast receivers respond to notifications from the system or other app components, e.g., when the battery is low or a file download is completed successfully. Content providers manage access to data, e.g., for storing and retrieving contacts. Most Android components must be explicitly *declared* in the Android Manifest file to be launched (besides dynamic broadcast receiver that can be registered and unregistered at runtime, and can only respond to events while the app is running). Declared components can only be launched within the scope of the app. To make a component launchable by other apps, it must be explicitly *exported* in the Manifest file. Such exported components can serve as additional entry points into the app.

Transitions between Android components rely on *Intents*, which specify, either explicitly (by name) or implicitly, the target component to invoke and the data to be passed. Implicit intents specify the type of interactions they handle via *action* elements of *intent-filters*, which can be either standard Android or custom developer-defined actions. Intents and intent filters can also contain other fields, such as *data* and *extras*, which carry extra information needed for the requested action.

The Android Debug Bridge (ADB) is a command-line tool included in the Android Software Development Kit (SDK) to facilitate communication with devices. It is used to install and debug apps, send inputs/intents, start *exported* activities at runtime, and more.

GUI-Based App Exploration. The vast majority of automated app exploration techniques proposed in the literature are GUI-based, simulating user interactions with apps' graphical interface. To further emulate user actions, these techniques can also send device- or system-level events, such as screen rotations or network connection status changes. Monkey [1], a pseudo-random exploration tool developed by Google, is part of the standard Android distribution and one of the earliest yet most prominent techniques, despite its simple exploration strategy. Aside from random approaches [3], [8], [11], more sophisticated approaches use app models [4], [5], [10], [12], [13], [17], [18], [21], evolutionary algorithms [6], [9], [16], symbolic execution [2], [7], and deep learning [14], [19], [21]. APE [13] and FastBot2 [21] are prominent model-based and reinforcement-learning-based approaches, which we evaluate in Section IV-A.

III. Methodology

We now discuss our study methodology. Specifically, we describe our app selection strategy and our experimental setup. Our study is driven by the following research questions:

RQ1: How does the app activity-level coverage achieved by humans compare with that of automated tools?

RQ2: What are the main reasons for unreachability in real-world applications?

RQ3: What are the activation properties of unreached application activities?

A. Experimental Setup: RQ1

App Selection. We started our analysis from the 68 open-source applications in the AndroTest [23] dataset, which has become a de facto standard benchmark used to evaluate numerous existing approaches, e.g., [9], [10], [12]–[15], [20], [33], [34]. As this benchmark was created in 2015, newer versions of the apps have become available throughout the years; these apps are commonly used to evaluate more recent tools [16], [35]. We thus collected the most recent version of each app in the dataset from the open-source Android app repository, F-Droid [36] (37 apps), from GitHub [37] (19 apps), and from Google Code archives [38] (5 apps). We could not find source code for four apps and thus excluded them from our analysis. We further excluded three apps that crashed on startup. At the end of this process, we obtained a set of 61 applications spanning the years 2007 to 2023.

As only 14 of the 61 benchmark applications are currently available on the official Google Play app store (all ranked below 200 in popularity) and only four of the benchmark apps are from 2023, we further collected our own dataset consisting of popular contemporary apps on Google Play. Specifically, in late 2023, we traversed the ordered list of the top 100 free applications from the Google Play store in Canada. We filtered out newly added apps with less than 500,000 downloads and then selected the most popular app from each of the Google Play categories, to ensure a representative sample that covers a wide range of behaviors. This process resulted in the selection of 20 apps from the list of 100 top free applications on Google Play from 20 distinct categories.

After initial experiments with this dataset, a few months later, we further extended it to include at least one app from each of the 32 Google Play categories (all categories besides Games). To this end, we sampled the most popular app from each category. As the popularity of 10 apps did not change and they were already included in our earlier sample, we extended the dataset with 22 additional apps, producing a benchmark of 20+22=42 applications in total.

In summary, our selected apps can be divided into three groups:

1. 47 open-source apps from the AndroTest benchmark, which range between 2007 and 2022 and do not appear on Google Play. We refer to these apps as *BenchNotGP*.
2. 14 open-source apps from the AndroTest benchmark, which range between 2012 and 2023 and are present on Google Play. We refer to these apps as *BenchGP*; we further split them into 10 older apps from 2021-2022 and 4 newer apps from 2023.
3. 42 closed-source popular apps from Google Play in 2023. We refer to these apps as *TopGP*.

The number and distribution of the benchmark apps by years, as well as the average number of activities in each set of benchmark apps, are summarized in the first three columns of Table I. Table II provides details about the Google Play apps. Specifically, the first six columns of the table list the id we assigned to the apps, their names, number of downloads, popularity rank, Google Play category, and the number of activities. The full list of applications we considered and additional details, e.g., their SDK levels and requested permissions, can be found online [31].

Metrics. In both manual and human exploration, we measured activity coverage by calculating the fraction of activities reached during app exploration, out of the total number of activities implemented by an app, as was done in earlier work [24], [25]. We picked this metric to be able to support our findings with the manual unreachability analysis in RQ2 and RQ3, which would be unfeasible to perform at a method- or statement-level: most of our *TopGP* apps are multi-dex, i.e., contain more than 65,536 methods [39]. Moreover, low activity coverage generally implies low method or statement coverage. Thus, we prioritize addressing low activity coverage first.

Manual Exploration. For the manual exploration, the first author of this paper and an externally recruited analyst experienced with mobile app development independently ran each application, aiming to systematically explore all visible user interfaces while making an effort to provide semantically-meaningful inputs. Consistent with our goal of supporting automated quality assurance of third-party apps, the analysts performed the exploration in a black-box manner, i.e., without prior knowledge of the application's code [40]. They aimed to reach as much of the application functionality as possible and were encouraged to follow the app prompts, e.g., to change device settings or interact with other apps. The analysts were equipped with a script that showed coverage achieved during the exploration, to be used as an indication of progress. During the exploration, the analysts also noted any "blockers" they observed, i.e., conditions preventing them from proceeding with the exploration. In case of blockers, they backtracked and made an effort to explore other parts of the application. The analysts were asked to stop exploration after spending five minutes being unable to further increase app coverage.

As most *TopGP* and some *BenchGP* applications require user authentication, each analyst created fresh accounts for each of the available login options in an application (e.g., Google, Facebook, and custom login). The analysts first explored the app without logging in and then created the needed credentials to log in in each of the available ways, to further explore the app.

TABLE I: Coverage for *BenchNotGP* and *BenchGP*

Benchmark (Year)	#Apps	Avg. # of Activities	% Reached Activities		
			Tools	Manual	Total
BenchNotGP (2007-2022)	47	5.9	83.2	88.7	88.7
BenchGP (2012-2022)	10	11.3	73.6	78.9	81.3
BenchGP (2023)	4	58.8	46.7	67	68
BenchGP (Total)	14	24.9	65.9	75.5	77.5

TABLE II: Coverage for Google Play Apps (*TopGP*)

ID	App Name	#Down-loads	Popu-larity	Category	#Acti-vities	#Reached Activities (%)		
						Tools	Manual	Total
1	WhatsAppMessenger	5B+	4	Communication	387	117 (30.2%)	131 (33.9%)	152 (39.3%)
2	SpotifyMusicandPodcasts	1B+	36	Music & Audio	98	13 (13.3%)	31 (31.6%)	31 (31.6%)
3	Instagram	1B+	25	Social Media	264	12 (4.5%)	17 (6.4%)	17 (6.4%)
4	TikTok	1B+	6	Social Media	325	49 (15.1%)	79 (24.3%)	83 (25.5%)
5	**Pinterest**	**500M+**	**53**	**Lifestyle**	**30**	**13 (43.3%)**	**15 (50.0%)**	**15 (50.0%)**
6	Uber-Requestaride	500M+	23	Maps & Navigation	128	2 (1.6%)	2 (1.6%)	2 (1.6%)
7	AmazonPrimeVideo	500M+	17	Entertainment	80	23 (28.8%)	35 (43.8%)	35 (43.8%)
8	Google Wallet	500M+	8	Finance	69	3 (4.3%)	3 (4.3%)	4 (5.8%)
9	CapCut-VideoEditor	500M+	8	Video Players	178	48 (27.0%)	54 (30.3%)	61 (34.3%)
10	**SamsungSmartSwitchMobile**	**100M+**	**20**	**Tools**	**51**	**10 (19.6%)**	**10 (19.6%)**	**10 (19.6%)**
11	Duolingo	100M+	51	Education	128	26 (20.3%)	27 (21.1%)	31 (24.2%)
12	AudibleAudioEntertainment	100M+	32	Books & Reference	63	12 (19.0%)	18 (28.6%)	18 (28.6%)
13	CanvaDesignPhotoVideo	100M+	31	Art & Design	29	7 (24.1%)	9 (31.0%)	10 (34.5%)
14	MicrosoftTeams	100M+	15	Business	299	7 (2.3%)	8 (2.7%)	8 (2.7%)
15	**Pluto TV**	**100M+**	**2**	**Entertainment**	**13**	**1 (7.7%)**	**1 (7.7%)**	**1 (7.7%)**
16	BumbleDatingFriendsapp	50M+	73	Dating	199	11 (5.5%)	38 (19.1%)	42 (21.1%)
17	BeCloserShareyourlocation	10M+	>200	Parenting	16	5 (31.3%)	10 (62.5%)	10 (62.5%)
18	CardBoard	10M+	>200	Libraries & Demo	17	7 (41.2%)	8 (47.1%)	8 (47.1%)
19	FIFA+ - Your Home for Football	10M+	70	Sports	35	7 (20.0%)	6 (17.1%)	8 (22.9%)
20	Expedia: Hotels, Flights & Car	10M+	68	Travel & Local	84	26 (31.0%)	25 (29.8%)	29 (34.5%)
21	**TemuShopLikeaBillionaire**	**10M+**	**1**	**Shopping**	**31**	**9 (29.0%)**	**13 (41.9%)**	**13 (41.9%)**
22	TheWeatherNetwork	10M+	64	Weather	63	19 (30.2%)	25 (39.7%)	26 (41.3%)
23	PictureThis-PlantIdentifier	10M+	25	Education	197	41 (20.8%)	37 (18.8%)	45 (22.8%)
24	Ticketmaster	10M+	22	Events	110	9 (8.2%)	16 (14.5%)	18 (16.4%)
25	ShopAllYourFavoriteBrands	10M+	19	Shopping	10	3 (30.0%)	3 (30.0%)	4 (40.0%)
26	Lensa	10M+	1	Photography	46	27 (58.7%)	29 (63.0%)	32 (69.6%)
27	**McDonaldsCanada**	**5M+**	**14**	**Food & Drink**	**157**	**33 (21.0%)**	**35 (22.3%)**	**35 (22.3%)**
28	FeverLocalEventsTickets	5M+	>200	Events	125	27 (21.6%)	36 (28.8%)	36 (28.8%)
29	SephoraBuyMakeupSkincare	5M+	75	Beauty	130	42 (32.3%)	52 (40.0%)	56 (43.1%)
30	Wonder - AI Art Generator	5M+	3	Art & Design	76	9 (11.8%)	8 (10.5%)	10 (13.2%)
31	AutoTrader-ShopCarDeals	1M+	>200	Auto & Vehicles	60	17 (28.3%)	24 (40.0%)	25 (41.7%)
32	MessengerLite-SMSLauncher	1M+	93	Personalization	65	11 (16.9%)	18 (27.7%)	19 (29.2%)
33	**LocalNews**	**1M+**	**74**	**News & Magaz.**	**75**	**32 (42.7%)**	**36 (48.0%)**	**37 (49.3%)**
34	AIMirrorAIArtPhotoEditor	1M+	55	Photography	63	1 (1.6%)	5 (7.9%)	5 (7.9%)
35	AirCanada+Aeroplan	1M+	53	Travel & Local	29	3 (10.3%)	4 (13.8%)	4 (13.8%)
36	FeelsyStressAnxietyRelief	1M+	27	Health & Fitness	34	4 (11.8%)	7 (20.6%)	7 (20.6%)
37	ChatGPTpoweredChat-NovaAI	1M+	5	Productivity	19	2 (10.5%)	3 (15.8%)	3 (15.8%)
38	REALTOR.caRealEstateHomes	500K+	>200	House & Home	15	8 (53.3%)	9 (60.0%)	9 (60.0%)
39	ShoppersDrugMart	500K+	>200	Medical	17	1 (5.9%)	2 (11.8%)	2 (11.8%)
40	VIZManga	500K+	>200	Comics	32	17 (53.1%)	17 (53.1%)	17 (53.1%)
41	CBCSportsScoresNews	500K+	102	Sports	35	11 (31.4%)	11 (31.4%)	11 (31.4%)
42	**PC Health**	**500K+**	**14**	**Health & Fitness**	**29**	**6 (20.7%)**	**8 (27.6%)**	**8 (27.6%)**
Average					93.1	22.4%	28.1%	29.6%

Both analysts were unable to create new credentials through the mobile interface of one app only (app #14 – Microsoft Teams) due to a documented bug with account creation in the mobile version of the app [41]. We thus report the observed coverage without logging in for this app.

Exploring an app took around 20 minutes, on average (min: 5, max: 55). We provide detailed plots of the achieved application coverage over time for each app and analyst online [31].

Automated Exploration. For the automated exploration, we focused on tools that are available, able to process large-scale applications, and able to run on a real device: close to half of the closed-source *TopGP* applications cannot be installed on emulators as these apps are suited to ARM architectures and emulating ARM on x86 desktops is too slow for a realistic exploration [25]). We ended up selecting two tools. The first is APE [13] – an automated exploration tool that was shown to outperform related techniques in a number of studies [27], [29] and that was recently updated to a newer version as part of the Themis project [29]. We further selected FastBot2 [21] – a recent publicly available tool designed explicitly for industrial-scale applications. We ran the single-device version of the tool as our discussions with the authors showed that the multi-device version is not part of the open-source implementation. Other tools we considered, e.g., [14], [16], [17], [19], were excluded either as we could not successfully run them in our setup or as the coverage results reported in comparative studies [27], [29] were lower than those of APE and FastBot2.

Consistently with the manual evaluation, we ran each tool twice: once without any manual intervention and once after we logged in into each app manually. To avoid cross-experiment contamination, we used different credentials for the tools and analysts runs. Each run was configured with a dynamic timeout, i.e., we stopped exploration when no improvement in coverage was achieved for one hour [12], [29]. Moreover,

tools could spontaneously stop their exploration, e.g., due to saturation. On average, the tools spent around 95 minutes on each app (min: 2, max: 261). The minimum exploration time of 2 minutes corresponds to two *BenchNotGP* applications for which exploration was interrupted by the tools due to consistent crashes detected in the app, even after multiple restarts. Detailed coverage reports for each tool are available in our online appendix [31].

The experiments were conducted on a Google Pixel XL running Android 10 (API level 29). We selected this model after ensuring compatibility with all the *TopGP* applications, i.e., API level 29 is within the minimum SDK and target SDK version of each app. For 23 older *BenchNotGP* apps incompatible with this model, we used emulators running the Android version compatible with the app.

B. Experimental Setup: RQ2 and RQ3

App Selection. To better understand the reasons for low app coverage and to extract patterns corresponding to the uncovered app activities, we selected and manually analyzed the code of 11 benchmark and Google Play applications.

For the benchmark apps, we included the four most recent open-source apps from the *BenchGP* dataset, all from 2023: WordPress, MyExpenses, K9-Mail, and Wikipedia. We selected these apps as (a) they include the largest number of unreached activities among all benchmark apps and (b) being more recent, do not include failures resulting from incompatible and missing servers. For *TopGP*, due to the extensive manual effort required to analyze reverse-engineered code of partially obfuscated large closed-source apps, we focused on seven case study applications. We selected these applications using a bin sorting technique: we sorted our 42 applications by the total number of downloads, divided them into seven bins of equal range, and randomly selected one app as a representative for each bin. The resulting set of applications is bold-faced in Table II and includes apps #5 (Pinterest) – a platform to create and share digital content; #10 (Samsung SmartSwitch) – a data transfer application; #15 (Pluto TV) – a free streaming app with access to hundreds of TV channels and movies; #21 (Temu) – a shopping application, which is also the most popular app on the Play Store; #27 (McDonald's Canada) – a menu ordering application which also provides access to promotional content; #33 (Local News) – an app which delivers daily personalized news to users; and #42 (PC Health) – an app that provides a variety of health-related services, such as tips, contents, prescriptions, communication and booking appointments with health care providers.

We believe our selection of apps is diverse and representative. The selected apps have 621 activities in total, out of which 363 are reached by neither humans nor tools (58.4%).

Manual Analysis. Unreached activities are guarded by conditions. Understanding the properties of these conditions can help devise strategies for triggering these activities in means other than GUI-based. To help with this goal, we systematically studied and characterize properties of conditions guarding the execution of unreached activities.

To this end, we first decompiled APKs of closed-source apps using JADX [42] – a decompiler for the Java programming language. We also used JADX's built-in deobfuscation functionality to map obfuscated names to more readable unique identifiers, which simplified the analysis of the code. Then, for both open- and closed-source apps, we performed a manual reachability analysis to identify paths leading to each activity not reached during exploration, as shown in Figure 1. Specifically, for each unreached activity, we searched for statements launching the activity, i.e., invoking an API such as *startActivity* (point ① in Figure 1).

Once located, we traversed the paths (marked with ② in Figure 1) to the activity backwards until we reached the *onCreate* method of the most immediate caller activity (point ③ in Figure 1). To account for activities started through implicit intents or exported components, we also searched for callers in the manifest and resource files. We leveraged knowledge from the app's manual exploration to guide the search, e.g., using the displayed text to aid locating the right activities.

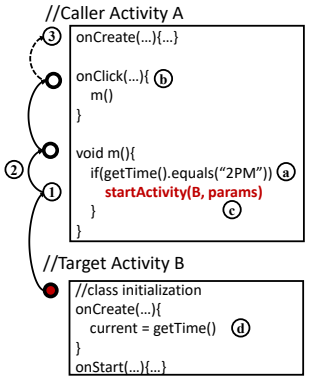

Fig. 1: Path analysis

If an activity had multiple callers, we sorted the obtained paths by size and picked the shortest one for further analysis. When the analysis of the selected path was inconclusive, we tried the next path in order and declared "unknown" if none of the path analysis provided conclusive results and there were no paths left to analyze.

For each of the analyzed paths, we collected all conditional checks of the path (e.g., the check ⓐ in Figure 1). Then, two authors of the paper independently summarized the checks in human-readable text descriptions (e.g., time-related), also taking into the considerations "blockers" reported by the app exploration analysts. We further used a card-sorting technique, which is typically employed to organize information into logical groups [43], [44], to derive a categorization of reasons for unreachability. Specifically, the authors met to discuss the produced descriptions (cards), refine their labels, merge related labels when necessary, and group them into high-level categories: reasons for unreachability described in Section IV-B. Disagreements (around 5%) were resolved in a joined discussion involving a third author.

Next, to analyze the code-level activation setup of unreached activities, we analyzed the origin of information used in the collected conditional checks (e.g., internal to the app or coming from the Android platform). Furthermore, for each path, we extracted the activity activation location, such as a callback in code (ⓑ in Figure 1) or the manifest file, and its exported status. We also inspected the target activity entry point (the *onCreate* method of Activity B in Figure 1), to identify types and usage pattern of data passed to the activity (ⓒ and ⓓ). Like with the unreachability reasons, we used a card-sorting technique to

extract properties of conditions guarding unreached activities: exported status, activation location, activation guards, and data usage. These are described in Section IV-C. The app analysis process took around 3-4 days per app, on average.

IV. Results

A. RQ1: Application Coverage

The last three columns of Tables I and II show the coverage results for benchmark and Google Play apps, respectively. Specifically, we report the coverage achieved by (1) both tools, combined, (2) both human analysts, combined, and (3) total coverage achieved by tools and humans, combined. We refer to these results as *Tools*, *Manual*, and *Total*, respectively. The difference between the coverage results achieved by the two tools is below 0.3%; the difference between the results achieved by the human analysts is also very low (<0.1%), confirming the reliability of our manual inspection. We use the *Total* coverage achieved by all tools and humans collectively to estimate the "best achievable result" and as a baseline for unreachability investigation in RQ2 and RQ3 (Section IV-B). Detailed coverage results for each tool and each human analyst are available online [31].

Overall, the tools were unable to outperform the human analysts. The average coverage achieved by the tools for *TopGP* apps is 22.4%, while humans could reach 28.1% coverage. As expected, we noted several issues preventing the tools from achieving the same results as humans, including the inability to provide semantic inputs (zip code in app #3 and a selfie image in app #16).

The performance of the tools also decreased as the size, number of activities, and the set of features provided by the app increased: they cover 83.2%, 65.9%, and 22.4% of activities, on average, for *BenchNotGP*, *BenchGP*, and *TopGP*, respectively.

While humans could generally reach more activities than tools, as they can bypass "semantic barriers" outlined above, we also observed a few cases reached by tools and not humans (5.6% of all reached activities). Such cases are caused by non-deterministic app behaviors, errors humans did not trigger, and human "exhaustion". For example, in app #13, actions that are visible after scrolling for more than 10 seconds were missed by humans, who most likely assumed that such a long scrolling action is redundant.

Yet, to our surprise, the human analysts only exceed the tools coverage result by a relatively small fraction. When combining the results of tools and humans, while the total coverage results are relatively high for the benchmark apps, we observe a total coverage of only 29.6%, on average, for *TopGP* apps. This motivates the need to better understand reasons preventing both automated and manual exploration from reaching the remaining >70% portion of these applications. We explore this next, focusing on a subset of seven apps from *TopGP* and four recent, 2023, apps from *BenchGP*.

Answer to RQ1: While the activity-level app coverage is relatively high for benchmark apps, especially those from 2007-2022, neither human nor automated tools exceed 30% coverage, on average, for newer popular Google Play apps.

B. RQ2: Reasons for Unreachability

By manually analyzing 363 unreached activities in our selected subset of 11 apps, we identified 10 main reasons preventing tools and human analysts from reaching 301 of these activities. We could not reliably recover unreachability reasons for the remaining 62 activities, as described below, and, thus, mark them as "inconclusive".

The first column of Table III lists the reasons for unreachability that we identified; the second column shows the total number and percentage of activities unreached for each particular reason in all the analyzed apps, combined. The remaining columns of the table show the breakdown of unreached activities for each app individually. While the number of individual app activities in each row adds up to the total number of activities in a row, the number of activities in a column does not add up to the total number of unreached activities. This is because an activity can have multiple reasons for unreachability. Several of the reasons we identified could only be observed in the closed-source *TopGP* apps, alluding to the need to extend the benchmark applications with a more comprehensive set of apps. We now discuss each reason for unreachability in detail:

1. **Server** dependencies correspond to cases where app behaviors depend on the results of communication with a third-party server, via HTTP or a socket, and constitute the largest reason for unreachability in our case studies (14.3%).

The majority of server dependencies that we observed are *push-based*, i.e., initiated by a server as a notification rather than initiated by the app via an explicit (pull) request. For example, Pluto TV sends catalog updates and displays in-app advertisement when specific events occur on the server side. Such server-side updates introduce non-determinism when running an application and result in large portions of an application that cannot be explored unless specific data is received from the server.

Our analysis shows that applications often use third-party libraries to receive server-side notifications. Moreover, these libraries combine code written in Java with code written in JavaScript. For example, Pluto TV, McDonald's, and Samsung Smart Switch use the *com.braze* marketing automation library that manages push notifications, in-app messages, and other forms of dynamic content. The library uses a JavaScript bridge to define APIs invoked by a remote server and delegates the invocations to Java methods, as shown in Figure 2.

For pull-based scenarios, the application typically fetches structured data from the server to display it to the end user, e.g., a list of available pharmacies in PC Health. Apps also use information pulled from the server as a feature toggle that enables/disables features on demand. For example, Pluto TV, WordPress, and Temu load configurations from the server at runtime to select which version of an activity to display.

```
1 var brazeBridge = {
2    logCustomEvent: function (name, properties) {
3    // invokes a Java method logCustomEvent
4 }};
```

Fig. 2: Server push-based dependencies in Pluto TV.

TABLE III: Reasons for Unreachability

Reasons	Total (363/621 unreached)	WordPress (72/111 unreached)	MyExpenses (7/43 unreached)	K9-Mail (13/33 unreached)	Wiki (5/48 unreached)	Pinterest (15/30 unreached)	SmartSwitch (41/51 unreached)	PlutoTV (12/13 unreached)	Temu (18/31 unreached)	McDonalds (122/157 unreached)	LocalNews (38/75 unreached)	PCHealth (21/29 unreached)
1. Server	52 (14.3%)	12		1				4	1	11	16	7
2. Alternate entry	50 (13.8%)	9	5	3		2	2	1	1	21	4	2
3. External resources	47 (12.9%)	1					12		5	29		
3.1. Equipment	15 (4.1%)	1					12		1	1		
3.2. Information	32 (8.8%)								4	28		
4. Disabled	40 (11%)	2	1		1	1	1	1	1	32		
4.1. For a version	30 (8.3%)	1								29		
4.2. For a user	10 (2.8%)	1	1		1	1	1	1	1	3		
5. Error handling	22 (6.1%)	3	1		1	4	2	2	1	4	2	2
6. Device	18 (5%)	1	1	2		1	8		1		3	1
6.1. Software	10 (2.8%)	1	1	2		1	4					1
6.2. Hardware	10 (2.8%)						6		1		3	
7. Environment	9 (2.5%)									6	1	2
8. Usage patterns	3 (0.8%)							1			1	1
9. Transitive	71 (19.6%)	21		1		1	19		1	23	5	
10. No caller	45 (12.4%)	22	1	1	1	1	1	2		1	12	3
Inconclusive	62 (17.1%)	5		4	2	7		2	8	24	4	6
Total	363 (100%)	72	7	12	5	15	41	12	18	122	38	21

Enforcing or simulating specific server behavior falls outside of the capabilities of typical GUI-based exploration tools.

2. **Alternate entry** refers to activities started through entry points other than the Android main activity. This includes exported activities visible to other apps and activities started through shortcuts or widgets. For example, the McDonald's app exports the *UberDeepLinkHandlerActivity*, which is called via an implicit intent when the user interacts with McDonald's through Uber's app delivery service.

Apps can also expose activities through external and deep links – a mechanism that takes the user directly to the app or a specific destination within an app. For example, the link *mcdmobileapp://aetcalendar?campaign=sigcrafted-launcher* is used in McDonald's to show a promotional campaign which was not triggered during manual and tool-assisted app exploration. In our further analysis, we could confirm that the activity can indeed be started through this link but displays an error indicating that the data associated with the link is missing.

Identifying such alternate ways to invoke an activity, including proper data types necessary for the activation, requires extensive analysis of the app and configuration files and is, again, beyond the scope of a typical GUI-based exploration tool. In our case studies, this reason caused unreachability in 13.8% of the cases.

3. **External resources** are dependencies on equipment and information not readily available at testing time. This is another major reason contributing to the lack of reachability in our case studies (12.9%). For example, Samsung Smart Switch requires a secondary phone to enable an activity that transfers data; McDonald's ordering features all require valid payment information, which restricts access to 28 activities in this app.

The interactions with external resources are typically implemented using GUI-, hardware- and sensors-related framework methods, often in third-party libraries. While straightforward semantic inputs can often be provided by a human running the app, such user-specific and, often, sensitive information is much harder and more expensive to generate, limiting the exploration of such apps.

4. **Disabled** are cases where an activity is explicitly "blocked", either for a particular version of the app or for a particular user. In our case studies, 11% of unreached activities are in this category.

We observe that application developers reuse the same code to create different versions of an application. For example, the McDonald's app contains a number of country-specific configuration files that determine the features that will be available for a specific app version. As the Canadian JSON configuration file does not include the *surveyEnabled* key, none of the survey-related activities can be reached. This key is included in other configuration files, e.g., for Slovakia. In fact, based on this configuration file, 29 activities were disabled in the Canadian version of the app, primarily for location-specific campaigns and promotions.

We also observe apps that include debugging activities and other non-user-facing functionality, which are disabled for the app end-users. For example, Pinterest includes a view only available to developers. Testing such activities requires enabling the corresponding flags in configuration files and/or code and cannot be "expected" from GUI-based exploration tools.

5. **Error handling** refers to activities that handle unexpected and erroneous cases. For example, all but one app in our dataset include the *PlayCoreMissingSplitsActivity* activity from Google's *com.google.android.play.core* library, which deals with corrupted installations of applications that rely on Dynamic Delivery or Split APKs to distribute app components. This activity can only be activated when the installation of the application was incomplete. As an example of a custom error recovery, Pinterest contains an activity that is started only if a crash occurred in the previous application launch. Such errors are hard to predict and simulate at testing time, contributing to the limited app reachability (6.1% of unreachable activities).

6. **Device** refers to dependencies on certain software or hardware specifications, which were not satisfied during exploration. This reason led to 5% of all unreachable activities. Software properties correspond to content, configurations, or applications that must be present on the device for an activity to be reached. While a human tester could deduce some of

such properties by interacting with the app, we were unable to do so in 2.8% of the cases. For example, one of the Pinterest activities can only be reached by sharing content with another app, Line. However, this option only becomes visible if Line is already installed on the device.

```
public void onClick(View v) {
  Intent i;
  if(SystemInfoUtil.isSamsungDevice()){
    i = new Intent(this, IntroduceSamsungActivity.
        class);
  }
  else if { ... }
  startActivity(i);
}
```

Fig. 3: Device hardware properties in Samsung Smart Switch.

Device hardware properties are immutable characteristics specific to the device type. Apps display particular content depending on such device characteristics (2.8% of the unreached activities). For example, Samsung Smart Switch displays different activities depending on whether it is running on a Samsung phone or another Android device, as shown in Figure 3. It also checks for the device's locale to deal with access restrictions, e.g., for certain countries where the Google Play store is unavailable. Similarly, Temu contains activities that are only displayed on devices with at least 600 dpi (i.e., large-screen devices, like tablets).

Checks conditioned on software and hardware properties are typically implemented by retrieving info from Android API methods and evaluating it against a set of pre-defined values. In practice, to design an appropriate test setup, an analyst would need to know the correct device and software specifications, which would be challenging in our target scenario, i.e., for testers without deep prior knowledge of the app.

7. **Environment** describes the situation of the user and its surroundings, such as being at a certain location or using the app during a specific date/time interval. For example, the user must be at the pickup location and retrieve their order, to trigger the McDonald's activity that changes the status of the order to "completed". Similarly, one can only chat with a PC Health agent during pre-defined operating hours, which fell outside of our testing time. Apps also use internally-timed events to redirect users to certain activities, e.g., for periodically claiming gifts in McDonald's.

To facilitate environment checks, apps use Android framework methods to retrieve information, such as the current time and location; they also register callbacks that trigger actions even when the app is not running, e.g., for alarms and location updates. Enforcing proper environment constraints at third-party testing time is challenging as testing budgets are typically limited and such testers cannot simply move to the right location or wait for days/months. Moreover, satisfying environment dependencies might require domain-specific knowledge, such as where the pickup location is and what the operating hours are. In our analysis, 2.5% of activities were unreached for reasons in this category.

8. **Usage patterns** refer to dependencies on the type and frequency of app usage. For example, PC Health keeps track of

```
public boolean completedEnoughHealthJourneys(){
return SharedPreferences.getInt("
    numJourneysCompleted", -1) >= 5;
}
public void promptUserForStoreRating(Activity a){
  if(completedEnoughHealthJourneys()){
    startActivity(new Intent(a,
        PlayCoreDialogWrapperActivity.class))
  }
}
```

Fig. 4: Usage pattern check in PC Health.

the user's completed health journey activities in persistent local storage; it then checks if the user has completed at least five activities before asking the user to rate the app. Figure 4 shows an example of such a check. Similarly, Pluto TV checks for the number of times the user switched channels prior to asking for feedback. In our analysis, 0.8% of unreached activities depended on the usage patterns; they were not reached during the exploration due to the limited time budget and lack of knowledge about such application-specific use.

9. **Transitive** are activities that do not require additional configuration and data parameters but would be reached if their caller activity were successfully reached during exploration. We observe that 19.6% of the unreached activities in our case studies belong to this category.

10. **No caller** refers to cases where no entry point triggering the activity under test was found in the app. This occurs in 12.4% of the cases and mainly corresponds to activities that are either used as interfaces/superclasses for other activities or that are deprecated. For example, 30 of the WordPress activities that could not be mapped to a caller context mainly consist of features that have been migrated to an auxiliary app, JetPack, and that will be removed in future versions of WordPress according to its documentation [45]. Another type of activities in this category, annotated with @Deprecated (i.e., outdated), mostly originate from common libraries and not (yet) removed by the library developers, although the libraries already include newer versions of the activities.

Inconclusive are 62 activities for which we could not reliably confirm the reasons for unreachability by analyzing the code: either because we could not build the complete path to the statement launching an activity or because we could build the path but could not resolve the constraints on the path. This mostly happened in highly-obfuscated applications, with around one-third of the cases likely originating from third-party libraries. It could be that at least some of these cases correspond to code that is not supported on the device or in an app, e.g., many activities of Google's augmented reality SDK, *com.google.ar.code*, used by both McDonald's and Pinterest for scanning features they implement, are only enabled on a subset of devices.

Answer to RQ2: The main reasons preventing GUI-based testing from covering a large portion of an app include app dependencies on remote servers and external resources, as well as app alternate entry points, error-handing and disabled code, and device software/hardware properties. Further improvements in GUI-based exploration strategies will not lead to substantial improvements in application coverage.

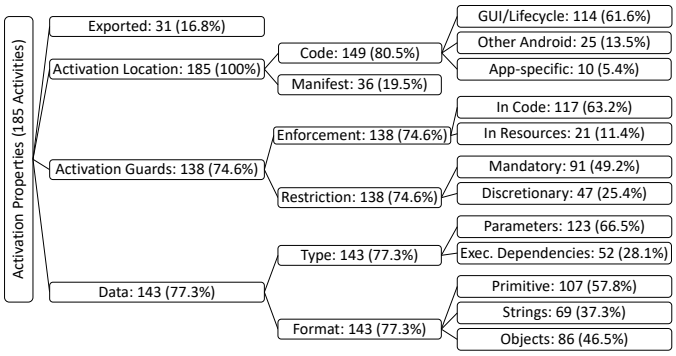

Fig. 5: Activation Properties

C. RQ3: Activation Properties

To characterize activation properties of unreached activities, we analyzed all activities considered in RQ2, excluding those where no calling path was found, i.e., activities in the *No Caller* and *Inconclusive* categories, as well as activities in the *Transitive* category as their activation is straightforward given their caller activity is triggered. That is, we analyzed the total of 363-45-62-71=185 activities. As discussed in Section III-B, we extracted four main activation properties: exported status, activation location, activation guards, and data usage, and further broke them into sub-properties. Figure 5 shows our categorization scheme, together with the number of activities in each category. We describe them next.

1. **Exported** states whether an activity is exported in the manifest file. Only exported activities can be activated directly via the ADB prompt. In our analysis, less than 20% of the activities are exported by the apps, limiting the applicability of tools that rely on this type of activation.

2. **Activation Location** describes where the activity is launched (point ① in Figure 1): either in code (80.5% of the analyzed activities) or in the Manifest file (19.5%). For the in-code location, we further separate activities launched from *GUI or lifecycle callbacks*, such as *onClick* or *onResume* (61.6%); *other Android callbacks*, such as *onReceive* for system-wide broadcasts (13.5%); or *App-specific callbacks*, such as sockets or cross-language events, as in Figure 2 (5.4%). When building tools that help increase application coverage, app-specific callbacks could be harder to identify than generic Android-based locations.

3. **Activation Guards** describe the properties of conditions constraining the launch of an activity, such as specific time or location. In our analysis, only 76.4% of activities have any activation constraints, which means that the remaining activities could be launched without any condition-related restrictions (these activities could still rely on data parameters, as described in the next category). For the activities that do have activation constraints, we distinguish between the constraints' *Enforcement* location and *Restriction* type. In 63.2% of the cases, the enforcement location is *in code*, i.e., appear as a conditional statement, such as *if* and *while* (point ⓐ in Figure 1). In 11.4% of cases, the enforcement location is *in resources*, e.g., the activities are flagged as disabled in the Manifest or other resource files. For example, in the Pinterest app, the 'android:enabled' property of a button is set to false in the production version of the app and, thus, can not be clicked by users to reach a developer-only debugging activity.

For the restriction type, we deem a restriction *mandatory* if it corresponds to a "hard constraint" and an activity cannot be triggered if the constraint is not met. For example, in the SmartSwitch app, a secondary device must be present for the majority of activities to be activated. We deem the restriction *discretionary* if it corresponds to a "soft constraint" – a choice in the app to show an activity only under specific circumstances. For example, in McDonald's, an advertisement activity is only shown to users located in California, as part of a promotional campaign. However, the activity's actual content is independent of the location and would still display correctly even if shown elsewhere, i.e., the constraint on the location does not impact the behavior of the activity when launched. In our analysis, we observe that the restrictions are mandatory in 49.2% of the cases and are discretionary (and thus can be more easily bypassed by the app exploration tools) in 25.4% of the cases.

4. **Data** refers to the information an activity relies on once it is launched. Only 77.3% of activities in our analysis rely on any data. For those, we distinguish between data of two different *Types*: data passed as *Parameters* in Intent fields (point ⓒ in Figure 1) and data retrieved as *Execution Dependencies* via global state, i.e., through APIs, shared variables, or shared preferences (point ⓓ in Figure 1). Furthermore, both parameters and execution dependencies can have different *Format*: *Primitives*, *Strings*, and *Objects*. We collect this information as primitive types are easier to generate when launching an activity than parameters of type String and, even more so, Object. Yet, an activity would typically have parameters of multiple types.

Combinations of Properties. After discussing each property individually, it is important to understand how these properties are combined, e.g., when an activity has both activation guards and data. Overall, there are more than 200 different

TABLE IV: Activation Properties

	Exported	Activation Location	Activation Guards	Data	#Activities (%)
Straightforward Activation: 42 Activities (22.7%)					○○○
1	Yes	Manifest	None	No	8 (4.3%)
2	Yes	GUI/Lifecycle	Code-enforced, Discretionary	No	1 (0.5%)
3	Yes	Manifest	Resource-enforced, Discretionary	No	1 (0.5%)
4	No	Manifest	None	No	4 (2.2%)
5	No	GUI/Lifecycle	None	No	2 (1.1%)
6	No	GUI/Lifecycle	Code-enforced, Discretionary	No	19 (10.2%)
7	No	GUI/Lifecycle	Resource-enforced, Discretionary	No	7 (3.8%)
Data Support: 45 Activities (24.3%)					○○●
8	Yes	Manifest	None	Yes	20 (10.8%)
9	No	Manifest	Nome	Yes	2 (1.1%)
10	No	GUI/Lifecycle/OtherAndroid	None	Yes	7 (3.8%)
11	No	GUI/Lifecycle/OtherAndroid	Code-enforced, Discretionary	Yes	13 (7%)
12	No	GUI/Lifecycle	Resource-enforced, Discretionary	Yes	3 (1.6%)
Data Support + Location Resolution: 7 Activities (3.8%)					●○●
13	No	App-specific	None	Yes	4 (2.2%)
14	No	App-specific	Code-enforced, Discretionary	Yes	3 (1.6%)
Data Support + Guards Resolution: 88 Activities (47.6%)					○●●
15	Yes	Manifest	Resource-enforced, Mandatory	Yes	1 (0.5%)
16	No	GUI/Lifecycle/OtherAndroid	Code-enforced, Mandatory	Yes	78 (42.2%)
17	No	GUI/Lifecycle	Resource-enforced, Mandatory	Yes	9 (4.9%)
Data Support + Location Resolution + Guards Resolution: 3 Activities (1.6%)					●●●
18	No	App-specific	Code-enforced, Mandatory	Yes	3 (1.6%)

TABLE V: Used Data Types

Primitives only	18 (9.8%)
Primitives + Strings	40 (21.6%)
Primitives + Strings + Objects	85 (45.9%)

combinations of the aforementioned properties, not all of which are present in our analyzed apps. Table IV summarizes the existing combinations, grouping them into five sections we identified, according to our perceived easiness of triggering activities of each types by the current/future tools that extend beyond GUI-based app exploration. More details about all combinations is in our online appendix [31].

The first column of the table assigns each combination type a unique id, to ease the discussion. The next four columns summarize the categories of each combination based on the description above. To simplify presentation, we treat data as a binary feature, i.e., with or without data, and further elaborate on data in Table V. In the last column, we present the number and the percentage of unreached activities in each combination.

Based on our analysis, we identified three main sources of exploration difficulty: (1) the ability to identify the activation location and then explicitly trigger the activity outside of its "normal" execution context, (2) the ability to resolve activation guards, and (3) the ability to provide the right data for the execution. These are marked by circles on the right-hand side of the section name: a full circle ● means "relatively difficult" and an empty circle ○ – "easier").

Activities in the first section of the table (rows 1-7, 22.7%) have either none or discretionary guards and rely on no data. Furthermore, they are activated from the commonly known Android callbacks and the Manifest file. Such activities can be triggered either directly via ADB (exported activities in rows 1-3) or by generating a custom activation file that starts the activities programmatically, similar to the mechanism employed by FlowDroid [46] (unexported activities in rows 4-7). We consider these cases to be the easiest targets for techniques that extend beyond GUI-based app exploration, which could identify the activities in the application code and subsequently trigger their execution.

All the remaining sections include activities that rely on data. Table V breaks down the required data by types, where we consider *Primitives only* (the simplest data to generate when triggering an activity, 9.8% of all cases), *Primitives and Strings* (21.6%), and *Primitives + Strings + Objects* (the most difficult case, 45.9%). The table shows that, in the majority of cases which require data generation, dealing with variables of types Strings and Object is necessary. This was already attempted by some beyond-GUI exploration tools [35], [47].

With the right data generated, the second section of Table IV (rows 8-12, 24.3% of activities) lists activities that can be triggered in a relatively straightforward manner because, like in the data-free case, these activities are triggered from the commonly known Android locations and have no mandatory guards. The remaining sections of the table bring additional complexities: the need to resolve activation location (rows 13-14, 3.8% of activities), the need to satisfy mandatory activation guards (rows 15-17, 47.6%), and the combination of the above (row 18, 1.6%).

Answer to RQ3: More than 20% of all unreached activities can be triggered in a rather straightforward manner, e.g., via ADB or simple injected calls. Almost 25% of the remaining activities can be triggered provided the necessary data is generated, albeit requiring the generation of String and Object data types. Triggering the remaining activities requires sophisticated techniques that can analyze app-specific custom callbacks and resolve nontrivial constraints guarding activity execution.

V. Discussion and Implications

Our analysis demonstrates that future investment in GUI-based app exploration techniques, i.e., techniques that simulate user interactions with apps' user interface, might not be fruitful to "break the ceiling" of around 30% coverage for large Google Play applications. To efficiently cover such applications – a necessary step in many functional and security testing scenarios, we envision the following three directions:

1. Analyze applications to identify activation conditions of untriggered activities, as we did manually in RQ3. Such an approach will enable (a) interactive decisions on which activation strategy to use for each case and (b) for cases that cannot be triggered automatically, delegation to human analysts while providing sufficient information about the required activation conditions (e.g., modifications of the mobile OS version or app environment). Effectively, such an approach will also help setting up a realistic application-specific coverage "upper-bound", instead of blindly striving for practically unachievable 100% coverage.

2. Develop automated techniques for triggering activities using an appropriate activation strategy: simply via ADB, by generating code-level activation, or by forcing the original application on the execution path leading to the activity.

3. Develop techniques to generate (or extract from the application code) data values needed for successful activation.

A number of existing approaches, often used in security, performance, and concurrency testing domains, aim to address points 2 and 3 above. In fact, in our work, we identified and classified 22 relevant techniques [33], [35], [47]–[66], which are described in more detail in the online appendix [31]. In a nutshell, these techniques introduce a variety of complicated activity triggering and value generation procedures, based on app, framework, and device manipulation. For example, one line of work modifies the control flows of a program to steer execution towards a desired point of interest. Another type of techniques customize the Android OS to inject events at the framework layer. The techniques use heuristic, symbolic-execution-based, and dynamic-value-harvesting strategies to generate the required data types.

However, among the 22 identified techniques, 16 are outdated and not compatible with the latest Android versions, and the remaining six are not publicly available. Out of these six, we contacted the authors of the two latest techniques, CAR [47] and COLUMBUS [35], but they indicated that the tools are not

ready for public release yet. Moreover, even when available, existing literature reports on numerous crashes induced by the invasive nature of such techniques [35], [47], [64], [67].

We believe that the complicated strategies these techniques employ make them hard to deploy, maintain, and upgrade to newer versions of the Android operating system. As a way forward, we thus suggest future research to invest in automatically extracting types of unreached activities (point 1 above) and then "staging" activation by focusing on activities that can be triggered with the most lightweight triggering strategy first. Such an approach can also avoid application crashes – an unfortunate side effect of "one-size-fits-all" exploration tools.

Our analysis shows that even the first stage of such an approach can increase application coverage by close to 25%, which is substantial, given the overall average coverage of 30% in large real-world applications. The most promising to apply at this stage would be techniques that support direct activity invocation via ADB or instrumentation of the application to activate its components, e.g., [48], [49], [55], [59], [65], [66].

Next, activities that use only primitive data could be invoked. We believe such a staged addition of "safe-to-activate" activities will result in a substantial increase in app coverage without associated crashes and tooling-related faults.

VI. Threats to Validity

The main threat to the **internal validity** of our results stems from the manual analysis we performed: when identifying reasons or properties for unreachability, we could have missed or misinterpreted some relevant dependencies or implementation patterns. To mitigate this threat, we cross-validated all findings between at least two authors of this paper, involved additional researchers from our group, and extensively discussed the categorization we present in this work. We thus believe that our results are reliable. Moreover, we believe that the diversity of reasons and conditions identified in our dataset is a good indicator of the quality/usefulness of the data produced in this work. For the **external validity**, our findings might not generalize beyond the dataset we considered. We mitigated this threat by carefully designing the data collection process to include a diverse set of real-world apps with large user bases, as well as popular benchmarks for testing tool evaluation. We thus believe that our data is representative.

VII. Related Work

We discuss related work among two dimensions: work that empirically compares app exploration techniques and work that provides insights into reasons for low coverage.

Comparison of GUI-based Exploration Techniques. Choudhary et al. [23] propose the first comparative study of GUI-based tools on 60 open-source apps, along several dimensions, such as ease of use, code coverage, and fault detection. The authors explicitly exclude tools that do not focus on app coverage, e.g., triggering crashes, and report that Monkey performed the best (around 45% coverage), despite being random. Wang et al. [25] extend this study with newer tools and apps from the Google Play Store and report a decrease in tool performance on industrial apps (around 30% coverage). While these studies focus on tool performance comparison, our work aims to explain such low observed performance.

Insights on Reasons for Low Coverage. A few authors [29], [30], [68] identify issues that automated tools cannot deal with, such as complex event sequences and app settings. Our work factors out challenges of inefficient GUI exploration and focuses on the remaining reasons for low application coverage. Azim and Neamtiu [5] leverage human input through a study with seven users who achieve around 30% coverage, on average. The authors report that the incomplete exploration by the users is mainly due to the lack of knowledge of application features or lack of interest. In contrast, our work involves skilled human analysts (incentivized by the desire to explore a large fraction of a mobile application, to reduce the effort of manual analysis). Furthermore, we perform a detailed code-level analysis for the reasons of limited coverage.

The work closest to ours is probably by Zheng et al. [24], who propose a GUI-based tool built on top of Monkey and manually analyze uncovered activities on one industrial app, WeChat. The authors identify reasons such as lack of login or financial information, historical data, dead activity, etc., and tool-specific reasons (e.g., text inputs, system events). They also discuss the need for developer-provided seed data and rules as testing efforts. By extending our analysis to more than one app, we were able to identify challenges not reported by Zheng et al., such as the need to satisfy certain environmental properties, usage patterns, and error-handling scenarios. We also augment findings of Zheng et al. around hardware properties, interface-only cases, and deep link routing in the alternate entry category. Moreover, we provide a fine-grained classification of code-level conditions required to trigger different types of activities.

VIII. Conclusion

In this paper, we compared the activity-level coverage achieved by state-of-the-art Android GUI-based application exploration techniques with that of skilled human analysts. To our surprise, the tool and the human analysts reached a similar level of coverage. This result indicates that improving application coverage by investing in GUI-based exploration strategies that trigger applications in a way similar to a human might not bring substantial improvements. Further work should thus rather focus on approaches for reaching the remaining 70% of the applications. To make progress in this direction, we performed a detailed manual code-level analysis of 11 large and recent applications, extracting reasons for the lack of reachability and suggestions for the staged application of relatively simple beyond-GUI activation strategies. We hope our work will inspire future techniques on both measuring the achievable app-specific reachability and making steps towards achieving it.

Acknowledgments. We thank Louie Tang and Kevin Liu from UBC, and Siddharth Gupta from IIT Patna, for their help in cross-validating the analysis results. We also thank the anonymous reviewers whose comments helped us improve the work.

REFERENCES

[1] "Monkey," https://developer.android.com/studio/test/other-testing-tools/monkey, 2023.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proc. of the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2012, pp. 1–11.

[3] A. Machiry, R. Tahiliani, and M. Naik, "DynoDroid: An Input Generation System for Android Apps," in *Proc. of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 224–234.

[4] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," *ACM Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[5] T. Azim and I. Neamtiu, "Targeted and Depth-First Exploration for Systematic Testing of Android Apps," in *Proc. of the International Conference on Object-Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013, pp. 641–660.

[6] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *Proc. of the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2014, pp. 599–609.

[7] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-Droid: Automated System Input Generation for Android Applications," in *Proc. of the International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 461–471.

[8] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?" in *Proc. of the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2016, pp. 987–992.

[9] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 94–105.

[10] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, Stochastic Model-based GUI Testing of Android Apps," in *Proc. of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 245–256.

[11] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: a Lightweight UI-Guided Test Input Generator for Android," in *Proc. of the International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 23–26.

[12] D. Lai and J. Rubin, "Goal-driven Exploration for Android Applications," in *Proc. of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 115–127.

[13] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI Testing of Android Applications via Model Abstraction and Refinement," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2019, pp. 269–280.

[14] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1070–1073.

[15] J. Qin, H. Zhang, S. Wang, Z. Geng, and T. Chen, "Acteve++: An Improved Android Application Automatic Tester Based on Acteve," *IEEE Access*, vol. 7, pp. 31 358–31 363, 2019.

[16] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel Testing of Android Apps," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2020, pp. 481–492.

[17] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "ComboDroid: Generating High-quality Test Inputs for Android Apps via Use Case Combinations," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2020, pp. 469–480.

[18] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis," in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 557–568.

[19] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement Learning Based Curiosity-driven Testing of Android Applications," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 153–164.

[20] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep Reinforcement Learning for Black-Box Testing of Android Apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–29, 2022.

[21] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "FastBot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–5.

[22] Y. Lan, Y. Lu, Z. Li, M. Pan, W. Yang, T. Zhang, and X. Li, "Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2024.

[23] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[24] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *Proc. of the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 253–262.

[25] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An Empirical Study of Android Test Generation Tools in Industrial Cases," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2018, pp. 738–748.

[26] H. N. Yasin, S. H. A. Hamid, R. J. R. Yusof, and M. Hamzah, "An Empirical Analysis of Test Input Generation Tools for Android Apps through a Sequence of Events," *Symmetry*, vol. 12, no. 11, p. 1894, 2020.

[27] W. Wang, W. Lam, and T. Xie, "An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 165–176.

[28] W. Wang, W. Yang, T. Xu, and T. Xie, "Vet: Identifying and Avoiding UI Exploration Tarpits," in *Proc. of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 83–94.

[29] T. Su, J. Wang, and Z. Su, "Benchmarking Automated GUI Testing for Android Against Real-World Bugs," in *Proc. of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 119–130.

[30] F. Behrang and A. Orso, "Seven Reasons Why: an In-depth Study of the Limitations of Random Test Input Generation for Android," in *Proc.of the International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1066–1077.

[31] F. Akinotcho, L. Wei, and J. Rubin. (2024) Supplementary Materials. https://resess.github.io/artifacts/MobileCoverage.

[32] G. Dev., "Platform Architecture," https://developer.android.com/guide/platform, 2023.

[33] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 33–44.

[34] T. A. T. Vuong and S. Takada, "A Reinforcement Learning Based Approach to Automated Testing of Android Applications," in *Proc. of the International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST)*, 2018, pp. 31–37.

[35] P. Bose, D. Das, S. Vasan, S. Mariani, I. Grishchenko, A. Continella, A. Bianchi, C. Kruegel, and G. Vigna, "COLUMBUS: Android App Testing Through Systematic Callback Exploration," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2023.

[36] "F-Droid," https://f-droid.org/, 2023.

[37] "Github," https://github.com, 2023.

[38] "Google Code Archives," https://code.google.com/archive/, 2023.

[39] G. Dev., "Multi-dex over 64k methods," https://developer.android.com/build/multidex#about, 2024.

[40] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated Testing of Android Apps: A Systematic Literature Review," *IEEE Transactions on Reliability (TR)*, vol. 68, no. 1, pp. 45–66, 2018.

[41] "Bug Report," https://techcommunity.microsoft.com/t5/microsoft-teams/cannot-access-microsoft-teams/m-p/1212804#M46636, 2020.

[42] skylot, "JADX," https://github.com/skylot/jadx, 2021.

[43] N. Nurmuliani, D. Zowghi, and S. P. Williams, "Using Card Sorting Technique to Classify Requirements Change," in *Proc. of the International Requirements Engineering Conference (RE)*, 2004, pp. 240–248.

[44] E. F. Cataldo, R. M. Johnson, L. A. Kellstedt, and L. W. Milbrath, "Card Sorting as a Technique for Survey Interviewing," *Public Opinion Quarterly*, vol. 34, no. 2, pp. 202–215, 1970.

[45] "WordPress: Migration to JetPack," https://wordpress.com/blog/2023/02/15/switch-to-the-new-jetpack-mobile-app/, 2023.

[46] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[47] M. Y. Wong and D. Lie, "Driving Execution of Target Paths in Android Applications with (a) CAR," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2022, pp. 888–902.

[48] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving Apps to Test the Security of Third-party Components," in *Proc. of the USENIX Security Symposium (USENIX)*, 2014, pp. 1021–1036.

[49] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor," in *Proc. of the Ninth European Conference on Computer Systems*, 2014, pp. 1–15.

[50] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong, "GroddDroid: A Gorilla for Triggering Malicious Behaviors," in *Proc of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2015, pp. 119–127.

[51] J. Schütte, R. Fedler, and D. Titze, "ConDroid: Targeted Dynamic Analysis of Android Applications," in *Proc. of the International Conference on Advanced Information Networking and Applications (AINA)*, 2015, pp. 571–578.

[52] Wong, Michelle Y and Lie, David, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[53] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting Runtime Values in Android Applications that Feature Anti-analysis Techniques." in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[54] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2017, pp. 300–311.

[55] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI Testing for Android Applications," in *Proc. of the 32nd Conference on Automated Software Engineering (ASE)*, 2017, pp. 27–37.

[56] X. Wang, S. Zhu, D. Zhou, and Y. Yang, "Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2017, pp. 350–361.

[57] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART." in *Proc of the USENIX Security Symposium (USENIX)*, 2017, pp. 289–306.

[58] L. L. Zhang, C.-J. M. Liang, Y. Liu, and E. Chen, "Systematically Testing Background Services of Mobile Apps," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2017, pp. 4–15.

[59] C. Zuo and Z. Lin, "SmartGen: Exposing Server URLs of Mobile Apps with Selective Symbolic Execution," in *Proc. of the International Conference on World Wide Web (WWW)*, 2017, pp. 867–876.

[60] L. Bello and M. Pistoia, "Ares: Triggering Payload of Evasive Android Malware," in *Proc. of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 2–12.

[61] X. Wang, Y. Yang, and S. Zhu, "Automated Hybrid Analysis of Android Malware through Augmenting Fuzzing with Forced Execution," *IEEE Transactions on Mobile Computing (TMC)*, vol. 18, no. 12, pp. 2768–2782, 2018.

[62] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-Based Energy Testing of Android," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2019, pp. 1119–1130.

[63] S. Shi, X. Wang, and W. C. Lau, "MoSSOT: An Automated Blackbox Tester for Single Sign-on Vulnerabilities in Mobile Applications," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2019, pp. 269–282.

[64] D. Wu, D. He, S. Chen, and J. Xue, "Exposing Android Event-based Races by Selective Branch Instrumentation," in *Proc. of the International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 265–276.

[65] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry Testing of Android Applications by Constructing Activity Launching Contexts," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2020, pp. 457–468.

[66] A. Liu, C. Guo, N. Dong, Y. Wang, and J. Xu, "DALT: Deep Activity Launching Test via Intent-Constraint Extraction," in *Proc. of the International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 482–493.

[67] A. Salem, M. Hesse, J. Neumeier, and A. Pretschner, "Towards Empirically Assessing Behavior Stimulation Approaches for Android Malware," in *International Conference on Emerging Security Information, Systems and Technologies (SECUREWARE)*, 2019, pp. 47–52.

[68] F. Thung, I. C. Irsan, J. Liu, and D. Lo, "Towards Benchmarking the Coverage of Automated Testing Tools in Android Against Manual Testing," in *Proc. of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems*, 2024, pp. 74–77.