Profile Coverage: Using Android Compilation Profiles to Evaluate Dynamic Testing

Jakob Bleier
jakob.bleier@seclab.wien
TU Wien
Vienna, Austria

Felix Kehrer
felix.kehrer@seclab.wien
TU Wien
Vienna, Austria

Jürgen Cito
juergen.cito@tuwien.ac.at
TU Wien
Vienna, Austria

Martina Lindorfer
martina@seclab.wien
TU Wien
Vienna, Austria

***Abstract*—The rising complexity of Android apps makes comprehensive dynamic testing infeasible, especially for third-party apps. Knowing which methods are exercised by real users typically requires costly user studies or access to usage telemetry. We show that Android's *compilation profiles*, specifically *Cloud Profiles* collected by the Google Play Store, offer a readily available, underutilized source of such information. These operational profiles aggregate which methods are commonly executed across users and guide ahead-of-time compilation during app installation. We provide the first in-depth characterization of *Baseline Profiles* and *Cloud Profiles* and show that over 99.89 % of the top 1,000 apps include usage-derived cloud profiles.**

Based on this insight, we introduce *profile coverage*, a novel metric that measures how well dynamic testing exercises the methods real users interact with. This metric builds on the idea of operational coverage and enables a more holistic evaluation of automated test input generators. To enable profile coverage measurements, we develop a lightweight tracer, ProfTrace, based on Linux kernel uprobes that requires no app or system modifications. We demonstrate its utility by comparing three tools and a no-interaction baseline on 50 popular apps, showing that profile coverage reveals differences that traditional code coverage misses. For instance, in *Candy Crush*, automated testing achieves only 2.22 % method coverage, but 21.39 % profile coverage—indicating better alignment with user behavior than traditional code coverage would suggest.

I. Introduction

Dynamic analysis of Android apps plays an important role in both quality assurance and security assessments. From developers trying to detect bugs and crashes [54, 65, 76], to researchers studying entire app ecosystems to detect and characterize malware [10, 45, 46] and assessing different aspects of app privacy [38, 56, 60–62, 66, 75], automating the interaction with apps is a task with many applications.

Limitations of Traditional Coverage Metrics. As app complexity increases, test input generation techniques are typically evaluated using code coverage metrics, which report the share of code executed during testing. These metrics range in granularity from instruction-level to method, class, or activity coverage. However, due to time constraints and diminishing returns [14, 50], exhaustive coverage is rarely pursued in practice. Even manual efforts fall short: Akinotcho et al. [2] show that human testers typically explore no more than 30 % of an app's code. Traditional coverage helps answer whether testing exercises "enough" code, but leaves open the question of whether it is the right code. Not all code is equally important: some methods are critical to core functionality, while others are rarely executed in real-world usage. Thus, traditional coverage cannot distinguish between coverage of highly relevant vs. rarely used parts of the app.

Understanding Profiles. To address this gap, we turn to Android's *compilation profiles*, which represent actual usage patterns at scale. In particular, *Cloud Profiles* are aggregated from telemetry data collected by the Google Play Store during real-world use and are used by the Android Runtime (ART) to optimize app startup. Despite being widely deployed, they remain underdocumented. We begin by answering **RQ1:** *What profiles exist, how are they created, and for what are they used?*, based on a deep dive into Android's source code.

We further ask and answer **RQ2:** *How prevalent are Baseline Profiles and Cloud Profiles among popular apps on the Google Play Store?*, and **RQ3:** *Do developer-supplied Baseline Profiles differ from usage-derived Cloud Profiles?* To this end, we collect over 43,056 APKs and 43,009 *Cloud Profiles* from the top 1,000 apps. We show that profiles are widely available and often diverge, underlining the value of metrics based on telemetry for testing and optimization.

Profile Coverage for Testing Effectiveness. We propose *profile coverage* as a novel metric for evaluating dynamic testing techniques. It measures the proportion of methods listed in an app's *Cloud Profile* exercised during testing. This approach builds on the concept of *operational coverage* [53], which evaluates testing against actual usage behavior. Unlike operational coverage, however, profile coverage does not require app-specific instrumentation or large-scale user studies; it leverages usage data already collected and curated via the Google Play Store. Profile coverage provides a more grounded and holistic assessment of test effectiveness by contextualizing which parts of an app are covered. This perspective is especially valuable when analyzing third-party apps, where the developer's intent or internal structure is often unavailable.

Beyond large-scale evaluation, profile coverage is also directly useful for developers. Since it reflects how well tests align with real-world usage, it can help gauge testing effectiveness in individual apps, particularly when adding new features or maintaining legacy code. Because *Cloud Profiles* are versioned and updated over time, profile coverage can also be incorporated into continuous integration (CI) pipelines to detect regressions in behavioral coverage across releases.

To illustrate, consider this example of testing Candy Crush with Android's UI/Application Exerciser Monkey [35] ("Monkey" for short): It yields just 2.22 % traditional method coverage, which might suggest a weak test. But profile coverage is 21.39 %, indicating that the most-used features are better tested. With reversed results, testing might appear thorough, but it would miss the features people use the most.

Enabling Profile Coverage Measurement. To investigate whether profile coverage can differentiate testing approaches, we ask **RQ4:** *Can profile coverage be used to measure differences in dynamic testing success?* We compare the profile and code coverage of three input generation tools: Monkey [35], DroidBot [42], and Fastbot2 [51] on 50 apps. Our findings show that profile coverage can reveal differences in tool behavior that traditional code coverage fails to surface: For the aforementioned Candy Crush, method coverage increases by 74.59 % when running Fastbot2 [51] compared to Monkey, from 22.52 % to 39.33 %. Profile coverage, however, drops significantly, from 21.39 % to just 3.13 %. This oversight could undermine the value of any behavioral or security insights derived from the test. Similarly, in Microsoft Authenticator, code coverage increases by 44.56 % but profile coverage only by 3.39 %. These examples highlight why profile coverage matters: it provides an additional lens to interpret coverage metrics, helping both researchers and practitioners better assess test effectiveness, especially when traditional coverage numbers are ambiguous or misleading.

To make profile coverage measurable, we develop ProfTrace, a lightweight method tracer for Android based on ART's compilation model and Linux's uprobe infrastructure. Unlike traditional tools, it does not require repackaging or system modification and runs on both emulators and hardware with root access. ProfTrace successfully measures execution on all 827 candidate apps, while ACVTool [59], used for method coverage, only works on 114 (13.78 %).

In summary, we make the following contributions:

- We characterize compilation profiles and study their prevalence, showing that virtually all popular apps have *Cloud Profiles*, i.e., aggregated usage statistics per app version. We further investigate differences between developer-provided *Baseline Profiles* and *Cloud Profiles*.
- We propose *profile coverage* as a metric for developers and researchers to measure how dynamic testing approaches approximate real users' interactions with an app.
- We develop ProfTrace, an open-source method tracer based on Linux kernel uprobes that allows us to measure profile coverage without modifying and repackaging the app under test, or a custom Android system.
- We measure profile coverage with ProfTrace for three UI input generators and compare it to code coverage. We show that it can be used to find differences in dynamic testing that method coverage alone would not uncover.
- We discuss additional future applications of *Cloud Profiles*, from guiding dynamic testing and fuzzing to providing developers insights into their app's usage patterns.

Artifacts. To enable the reproducibility of our results and to foster research in this area, we provide our source code for profile collection as well as ProfTrace at https://github.com/SecPriv/android-profile-tracing. We provide our dataset of apps with their corresponding profiles upon request.

II. Android Compilation Profiles

A. Background: Android Apps and Runtime

Android apps are distributed in the Android Package file (APK) format, which is a signed zip archive that must include the `AndroidManifest.xml` file that stores metadata, such as the app name and app ID (also called package name). The latter uniquely identifies an app on a device and on the Google Play Store. An APK can also contain additional files that may be required for running an app, such as native libraries, images, and data. Originally, apps were distributed as single APKs, with resources for multiple device configurations packed into the same file. However, devices cannot use resources they are not compatible with, like high-resolution images on a low-resolution device or languages that are not activated on the device. That is why developers are now required to upload new apps as App Bundles, which contain all code and resources but can be bigger than the maximum APK size of 150MB in the Google Play Store, which in turn creates appropriate variant APKs for different target architectures. These additional APKs are called "split APKs" and are served to users to avoid sending unnecessary files to devices that cannot use them.

The actual code of an app is stored in one or more Dalvik executable (DEX) files within the APK, within which each string, field, class, and method has a numeric identifier. Because the number of identifiers is limited per DEX file to 65,536, complex apps have more than one DEX file. To identify a method uniquely per app, we refer to the combination of DEX file and method identifier as method ID.

The idea of using a list of method IDs for partial ahead-of-time compilation on Android was first introduced by Lim et al. [44] in 2012. It was not adopted in Android until version 7 (Nougat, released in 2016) as *compilation profiles*. At this point the Android Runtime (ART) executed app code by either compiling the Dalvik code to native code ahead-of-time (AOT), just-in-time (JIT), or by interpreting it. The system decides this on a per-method basis [12, 25]. AOT compilation of important methods happens during installation and downtime, that is, when the phone is idle.

Other software, such as Mozilla Firefox and Google Chrome, also uses compilation profiles for profile-guided optimizations [70], i.e., to heavily optimize code that is executed more frequently. But Android is currently the only platform where this is used by the app runtime itself, and for which a large number of profiles are available for apps from a variety of different developers. We empirically show this in Section III.

B. Compilation Profiles

Compilation profiles have received limited attention from the research community. To the best of our knowledge, related work so far has only explored the optimization of their

generation: Visochan et al. [71] investigated how to generate useful profiles using machine learning, but did not cover the current implementation of all the different kinds of compilation profiles on Android. One of the reasons might be that their documentation has been sparse: In our quest to investigate profile generation, usage, and general characteristics, only the Android source code provided definitive answers to some questions, although the documentation keeps expanding.

Compilation profiles are binary files that start with the magic header '`pro\0`' followed by their version number. Until Android 11 (R, released in 2020) version `010` was used, since Android 12 (S, released in 2021), up to and including the latest Android 16 as of September 2025, version `015` is used [5, 28]. They are used not only for regular apps but also for system components such as code found in the "bootclasspath" [29].

Each compilation profile contains three lists of method IDs per DEX file, which contain the following categories of methods (also see Listing 1 for an example *Cloud Profile* for the Signal app[1] we extracted using `profman`):

- *hot methods* are hot code in the usual sense: These methods get called most often, so compiling them AOT should result in the greatest speedup. By default, the ART will always compile them AOT for regular apps. Thresholds for when a method is considered hot can be configured [6, 32] when building an Android system.
- *startup methods* are involved in getting an app from not running to showing the first screen. These are also compiled AOT by default unless the system has been built with the "LowMemoryMode" setting [4]. In that case, the code is either compiled JIT or interpreted.
- *post-startup methods* run after the first screen is shown to the user, but before the main functionality of the app is reached. This could mean, for example, methods related to login or methods that load resources while a loading screen is shown. In practice, this is not used for AOT compilation on Android 10 through 15.

Figure 1 provides an overview of the path of compilation profiles throughout an app's lifecycle, from the development of an app, to the distribution through the Google Play Store, and to its execution on users' devices. Depending on how profiles are generated, we can distinguish between the following types:

Baseline Profiles, also referred to as "Developer Profiles," are created by the app developer and bundled in the APK at `assets/dexopt/baseline.prof` [28]. While they can be hand-written lists of methods that should be included, these lists can also be automatically generated with Android Studio tools that use the automated testing framework to run a series of developer-defined commands and collect statistics on the called methods. This only works if the app source code is available, since Android Studio uses a unit-testing library to track method calls and not the ART, and requires the app to be marked as debuggable. *Baseline Profiles* can also be generated for libraries and will be merged into the app's baseline profile during compilation. Note that creating *Baseline Profiles* is optional, and we investigate their prevalence in Section III.

[1]Signal Private Messenger (version 7.40.2) https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms&hl=en

```
=== Dex files  ===
/<path-to>/base.apk [checksum=1c7b6436]
<...>
=== profile ===
ProfileInfo [015]

classes.dex [index=0] [checksum=1c7b6436]
    hot methods:
        void org.thoughtcrime.securesms.MainActivity.
             onCreate(android.os.Bundle, boolean)[],
        void org.thoughtcrime.securesms.conversation.v2.
             ConversationActivity.onCreate(android.os.Bundle
             , boolean)[],
        <...>
    startup methods:
        void org.thoughtcrime.securesms.MainActivity.
             onCreate(android.os.Bundle, boolean),
        void org.thoughtcrime.securesms.conversation.v2.
             ConversationActivity.onCreate(android.os.Bundle
             , boolean),
        <...>
    post startup methods:
    classes:
        org.thoughtcrime.securesms.MainActivity,
        org.thoughtcrime.securesms.conversation.v2.
             ConversationActivity,
        <...>
<...>
```

Listing 1: *Example `profman` output for the Cloud Profile for the Signal app.* If the APK is available, DEX method IDs can be resolved to classnames as shown here. The brackets for hot methods may contain additional information to optimize megamorphic functions. In this case the *Cloud Profile* does not contain anything in the post-startup methods list.

Cloud Profiles, which more accurately could be referred to as "Aggregated Usage Profiles" (see Listing 1 for an example), are based on usage patterns observed by Google and distributed alongside apps installed through the Google Play Store in a DEX metadata (DM) file as `primary.prof`. To collect the usage data for these profiles, any app installed through the Google Play Store is profiled during its execution on the users' phones by default, unless users choose to disable "App Install Optimizations" in their settings with a link to "Learn more." It leads to a Google support page [34] explaining that they collect information on "which parts of an app you use the first time you open it after installation" and clarify that no content, such as usernames and uploaded data, is collected. How the usage data from individuals is aggregated is not publicly documented. In Section III we show that *Cloud Profiles* are available for virtually all popular apps.

Current Profiles collect the run-time information of an app while it is being used on a device. If an Android device has multiple users set up, Android creates separate profiles for an app for each user [7]. After a user installs an app, this profile is empty, and it gets populated with methods during execution that are considered "warm" by the runtime. Typically, this means they have been compiled JIT and were called often enough. The exact threshold when this happens depends on an internal counter that can be configured for the system and represents "number of calls, backward branches, and other

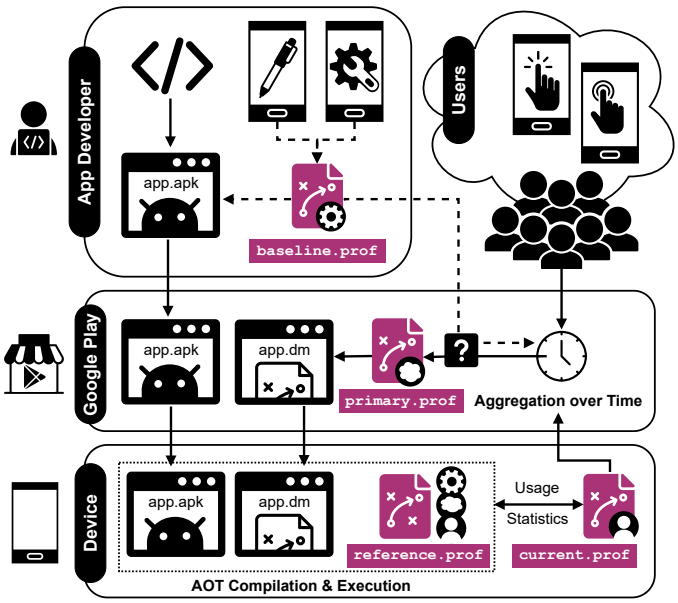


Fig. 1: *Lifecycle and relationship of compilation profiles.* Baseline Profiles are created by developers and contained in the APK (`.apk`), while Cloud Profiles (`primary.prof`) are created by the Google Play Store based on data collected from its users and are distributed in an extra DEX metadata file (`.dm`). Current Profiles represent the usage patterns of individual users on a device that are fed back into Google's data collection. All these, if available, are merged into the Reference Profile used by the ART to AOT-compile app code. We refer to the Current Profile as `current.prof` and the Reference Profile as `reference.prof`, because both files are originally also named `primary.prof`.

factors" [32]. Because the default values are too conservative for benchmarking [3, 8], these profiles are not suitable to collect a definitive list of executed methods.

Reference Profiles combine an app's *Baseline Profile*, *Cloud Profile*, and *Current Profile*. Android automatically merges the available profiles and passes them to the ART's AOT compiler [9]. During installation is not the only time profile-guided compilation happens on Android. As mentioned, the usage of an app creates the *Current Profile*, which changes over time and is the main input to "background dexopt" [27]. This task runs once a day and combines an app's available *Current Profile*, *Cloud Profile*, and *Baseline Profile* into the *Reference Profile* to update the `odex` ("optimized DEX") file, which contains the AOT compiled binary code (an "Of Ahead Time" or OAT file for short). Internally, `dex2oat` performs the compilation as part of the background task, during app installation, as well as app execution for JIT compilation [9, 12, 13], but it can also be invoked as a command-line tool manually on an app or through the package manager `pm`.

Boot Profiles are not used to optimize regular apps. Instead, they use the profile information from apps to optimize "system-level components such as system server and boot classpath" [29]. The Android command-line program `package` can be used to aggregate a boot profile from all individual app profiles of the boot image. However, we consider them out of scope for the remainder of this work and focus on individual apps' compilation profiles instead.

Startup Profiles are used during the creation of DEX files [31]. Their goal is to bundle Dalvik methods used for starting an app into the first DEX file that the ART loads, avoiding delays in startup caused by loading multiple files into memory. They do not influence which methods are compiled AOT by the ART and are only available to the app developer.

RQ1: What profiles exist, how are they created, and for what are they used?
While the *Reference Profile* is a combination of all profiles for AOT compiling an app to optimize its performance, the *Baseline Profile* and *Cloud Profile* are availalble at install time. The former is part of the APK and created by the developer, the latter is aggregated by the Google Play Store from *Current Profiles* of real users interacting with the app.

III. Prevalence of Compilation Profiles

Having now understood what kinds of profiles an app *can* have, the next question is which kinds of profiles apps actually *do* have and use. To answer this question, we collect a dataset of the most popular apps on the Google Play Store and collect their profiles daily for over seven weeks.

A. Profile and App Collection

Extending our work on app collection that had issues with exhaustive app discovery [67], we first collected the Google Play Store's *sitemap.xml*, which provides a list of *all* available app IDs. We then downloaded each apps' metadata and sorted it by number of downloads to select the most popular $1,000$ apps on March 28th, 2025. Each of the apps has at least 100 million downloads, with an average of 461.9 million total downloads. Over the course of seven weeks (March 31st – May 20th, 2025) we attempted to download each app on this list using a patched version of `apkeep` [20]. Our patch enabled us to download the *Cloud Profiles* without relying on a hardware phone, as well as all necessary split APKs in a configuration that allows us to run them on our phones later on. We open-source this patch to upstream this functionality.

For some apps, the download failed for various reasons: The most common ones were that apps were removed between the creation of our initial list, or that they were not available in our geographic region, or not supported on the target device we specified. In case of network errors, we did not attempt to re-download an app, which sometimes led to missing downloads as well. These problems affected a total of $15.58\,\%$ of our downloads. After this collection, we ended up with a total of $43{,}056$ base APKs that are unique per app ID and day, $81{,}509$ total split APKs, and $43{,}056$ DEX metadata files.

As discussed in Section II, the compilation profiles are distributed as binary files, which we extracted from the APKs and DEX metadata files, respectively. To parse them we used

`profman`, an Android system binary to create a human-readable version of the compilation profiles. By default, it only prints the method IDs, but can also be instructed to resolve and print the method names from the relevant DEX file instead.

B. Profile Availability and Updates

In our dataset we found that only about 85.93 % of top (i.e., most popular) apps had a *Baseline Profile*, but virtually all apps had *Cloud Profiles* available: We extracted 36,997 (85.93 %) *Baseline Profiles* from APKs, and 43,009 (99.89 %) *Cloud Profiles* from the DEX metadata files.

According to Google's documentation, it takes between "hours and days" [30] for *Cloud Profiles* to be available. For the apps in our dataset, the actual time seems to be lower. Only 47 (0.11 %) app downloads did not come with a *Cloud Profile*. However, as the Google Play Store supports staged rollouts, it is possible that profiles are collected from a smaller group of users first, and a *Cloud Profile* is already available when more users receive the update.

RQ2: How prevalent are *Baseline Profiles* and *Cloud Profiles* among popular apps on the Google Play Store?
While around 85 % of the top 1,000 most popular apps on the Google Play Store use developer-provided *Baseline Profiles*, virtually all (99.89 %) have *Cloud Profiles*.

We further investigated whether *Cloud Profiles* change over time, as this can influence any further comparison with *Baseline Profiles*. To determine this, we created a mapping of app ID and app version code combination (2,952 unique app version pairs) to the set of unique hashed *Cloud Profiles* contents. This allowed us to observe the evolution of app versions over time and to query whether any app version had multiple profiles, which was the case for 83 (2.81 %) app versions. We manually investigated 6 apps and found that in each case, the *Cloud Profile* changes only one day after a new version was released. Suspecting that after an update, when no *Cloud Profile* is available, the Google Play Store might use the *Baseline Profile* as *Cloud Profile*, we also compared those on the days of an app update and the following day. In the cases of an app version having more than one unique *Cloud Profile*, we found the profile served by the Google Play Store on the same day we observed a new app version indeed to be more similar to the *Baseline Profile*, before being updated. We assume that until enough user data is aggregated, the Google Play Store uses the *Baseline Profile*, if available, as a basis for the *Cloud Profile*. Once an aggregated *Cloud Profile* has been created, we saw no indication of it changing over time.

C. Baseline vs. Cloud Profiles

To explore the relationship between profile types, we selected data collected on a single day (May 7th, 2025). For each app, we collected the combined set of unique hot, startup, and post-startup method IDs in each available profile. We then calculated the number of methods in both profiles, unique to the *Baseline Profile*, and unique to the *Cloud Profile*. We normalized this by the overall number of methods in an app, which we counted using the Android SDK's `apkanalyzer`.

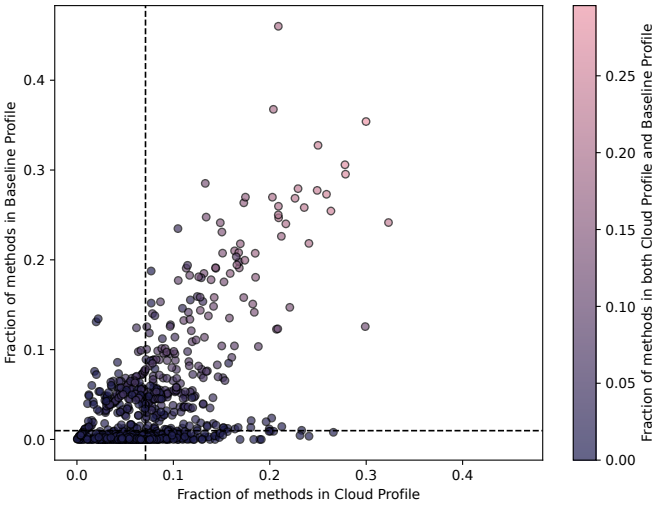


Fig. 2: *Methods in Baseline Profiles and Cloud Profiles, normalized by the total number of methods in an app.* We merged the hot/startup/post-startup methods of each profile, and computed the overlap and unique method IDs per profile type. The dashed lines indicate the median. We observe that a *Cloud Profile* is not just a superset of methods declared in a *Baseline Profile*: For most apps, the Google Play Store removes or replaces a large percentage of method IDs.

Figure 2 shows our results. While all of the 827 apps had a *Cloud Profile*, only 708 (85.61 %) had a *Baseline Profile* as well. If both were available, their content was largely similar, with generally more methods in the *Cloud Profiles*. However, the aggregated profiles were not just a superset of the *Baseline Profiles*, with *Cloud Profiles* often removing methods that were placed in the profiles by the developers.

Baseline Profiles alone are not enough to describe typical user interactions for two reasons. First, even for popular apps, they are often not present, while *Cloud Profiles* are available for virtually all apps. Second, the difference to *Cloud Profiles* suggests that the "common user interactions" developers are encouraged to capture in their profiles [28] do not always represent the actual user behavior as observed by Google. Investigating the difference between these profiles could help developers gain a deeper understanding of how their user base is interacting with their apps and to make better decisions about which methods to include in their *Baseline Profiles*.

RQ3: Are aggregated *Cloud Profiles* diverging from developer-supplied *Baseline Profiles*?
Cloud Profiles typically contain more methods than *Baseline Profiles*, but the overlap can be significant. The methods in *Cloud Profiles*, however, are not a strict superset of the methods in developer-provided *Baseline Profiles*. This indicates that for some apps, the actual usage of an app deviates from what the developer expects.

	Cloud Profiles	*Baseline Profiles*
Apps with available profile	827 (100.00 %)	708 (85.61 %)
Mean unique methods	24,567 (10.90 %)	18,240 (06.20 %)
Standard deviation	18,615 (07.16 %)	27,008 (08.61 %)
Min. unique methods	56 (00.06 %)	6 (0.002 %)
Median unique methods	20,785 (09.33 %)	5,639 (02.96 %)
Max. unique methods	118,095 (47.75 %)	171,228 (56.21 %)

TABLE I: *Statistics of Cloud Profiles collected on May 7th, 2025.* While all apps had *Cloud Profiles* available, just 85.61 % of top apps provided a *Baseline Profile*. Typically, the latter contains fewer methods, both in absolute numbers and as a percentage of an app's total methods, shown in brackets. An exception is the largest profile: *Cloud Profiles* are not just a superset of *Baseline Profiles* (see Section III-C).

Type of Methods Declared in Cloud Profiles	# of Apps	
Hot, Startup, and Post-startup	2	(00.24 %)
Hot and Startup	825	(99.76 %)
Hot and Post-startup	2	(00.24 %)
Startup and Post-startup	0	(00.00 %)
Hot only	478	(57.80 %)
Startup only	557	(67.35 %)
Post-startup only	2	(00.24 %)

TABLE II: *Overview of how methods are tagged as which type in the Cloud Profiles.* When a method is in the *Cloud Profile*, it is usually tagged hot, startup, and post-startup at the same time. Only 2 apps' *Cloud Profiles* contain methods that are tagged as only post-startup, or hot *and* post-startup. This is not surprising, because post-startup methods currently do not affect AOT compilation (see Section II-B).

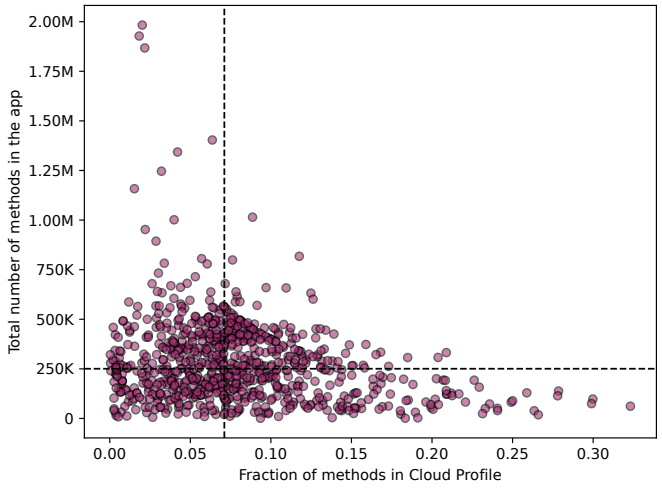


Fig. 3: *Fraction of methods in the Cloud Profile of an app plotted against the app size, in number of methods.* The dashed lines indicate the median. While a handful of apps have more than a quarter of their methods in the *Cloud Profile*, as the size of an app increases, the maximum fraction of methods tends to decrease. This is expected because compiling too many methods during installation takes a long time and goes against the idea of profile-guided compilation of apps.

D. Hot/Startup/Post-startup Methods in Cloud Profiles

The apps we collected on May 7th have, on average, 282,111.65 methods with a standard deviation of 208,079.71. Table I provides an overview of how many methods the profiles for an app contain. Through the *Cloud Profile*, an average of 7.88 % (standard deviation of 5.17 %) of an apps' methods is compiled AOT. Figure 3 shows an overview of how many methods an app has, how many of those are in the *Cloud Profiles*, while Table II shows how often methods are classified as hot, startup, post-startup, or a combination. While virtually all profiles have methods that are considered only hot or startup, other combinations are far less prevalent.

For our example app Signal, the APK contains 306,334 Dalvik methods. Of those, 55,025 (17.96 %) are in the *Cloud Profile* and of those, 12,101 (21.99 %) belong to the class `org.thoughtcrime.securesms` matching the app ID. Focusing only on methods from the app ID package and excluding potential library code might seem reasonable, but it is problematic for two reasons: First, library code in the profile indicates that it is important to the typical usage of an app, and second, reliably distinguishing between app code and library code is non-trivial [74]. For example, Signal also uses the classes `org.signal` and `org.whispersystems` with 994 and 486 methods in the *Cloud Profile*, respectively. Note that discerning libraries from app code is out of scope for this paper, and for our experiments, we do not distinguish between methods based on their package.

Finally, the lack of post-startup methods for Signal (see Listing 1 in Section II-B) represents the overall distribution well: Only two apps in our dataset had *Cloud Profiles* that contained methods in the post-startup category, which aligns with our finding that it is effectively ignored on Android for AOT compilation as of the latest Android version.

IV. Profile Coverage

Automated test input generators aim to cover the most relevant parts of an app, under certain time and resource constraints. A notion of code or activity coverage is typically used as a metric to quantify the success in testing the target app [2, 16, 17, 42, 51], e.g., whether an approach is better in (exhaustively) testing an app compared to previous ones, or how close it comes to simulating the manual input from humans [62, 73]. This coverage can also be integrated in the decision-making process for coverage-guided testing, most notably during fuzzing [65].

Code coverage can be calculated at different granularities, e.g., how many instructions, branches, methods, or classes are executed, which correlate strongly with each other [41]. Code coverage can be also be limited by only taking actually reachable code into account [49]. In the case of Android apps, the code distributed with an APK should typically be reachable, because the Android build toolchain also includes recommended steps to remove dead code from the app and

its dependencies [12, 33], which are enabled by default. However, an app's dependencies on external resources or disabled features make complete coverage infeasible [2].

We argue that in a variety of use cases, it can be beneficial to narrow down the scope by only focusing on the code that is covered by typical user interactions. For example, to know whether the code parts frequently executed by users have been sufficiently tested, or whether a discovered bug or vulnerability is part of a common user interaction, or not. We call this metric the *profile coverage* and define it in the following way, where M_{prof} is the set of methods in the *Cloud Profile* and M_{exec} is the set of executed methods in a specific test scenario:

$$\text{Profile Coverage} = \frac{|M_{\text{prof}} \cap M_{\text{exec}}|}{|M_{\text{prof}}|}$$

As we empirically evaluated in Section III and discussed in Section II, not only are *Cloud Profiles* readily available for apps on the Google Play Store to their developers as well as to the research community, they also represent aggregated user interactions from potentially millions of users of an app.

A. Challenges for Measuring Profile Coverage

To measure profile coverage, we require a way to trace the method calls of Android apps to compare them against the methods in the profile (see Section IV-B), as well as a source of interactions (see Section IV-C). For the latter, we selected automated UI test input generation, as it is the predominant use case where generating inputs close to the ones real users would provide is the most desirable. Finding available and robust tooling turned out to be a challenge for both, since most tools have been deprecated or make assumptions that we consider to be too invasive for dynamic testing. For example, a common approach for both method tracing and test input generators is to repackage an app and insert additional code. This fails for apps that check whether they have been tampered with [37, 55, 64, 78]. Apps also check whether they are being executed in an emulator or on a modified Android system [39, 55, 64, 68, 78], either by querying system properties or reasoning about values reported by various sensors. For example, if the battery never decreases even though the system is not charging, it is likely an emulated system. While removing these checks is sometimes possible, it needs to be done manually and is not an option for large-scale evaluations. We aim to minimize changes made to the app and the Android system in order to achieve compatibility with most apps.

Existing approaches to trace methods usually have one or more of the following limitations: They require rebuilding the Android system [11, 19], repackaging the app under test [21, 22, 47, 59, 77], only work with an app that has the *debuggable* or *profileable* flag set [26, 58], or they attach a debugger to the app process [23]. BPFRoid [1] is the only tool that works on hardware as well as emulated devices and requires only root. In preliminary tests, we found that attaching eBPF programs to methods does not scale beyond hundreds of methods and is more suited to analyze a limited number of specific methods.

Finally, as described in Section II, the ART has built-in functionality to profile apps and uses it to generate the *Reference Profiles*. Unfortunately, this only works with a statistical threshold depending on how Android was configured and will not mark all executed methods as "hot," making precise coverage computations impossible. For developers, Android Studio offers the generation of *Baseline Profiles* using automated macro-benchmarks. This works if the source code is available because it depends on the Android test suite being compiled into the app, or the app being built in a "debuggable" mode. This generates a human-readable file, which the developers are invited to fine-tune, and serves as the basis of *Baseline Profiles*.

B. Our Approach: ProfTrace

We propose a new method tracer based on the Linux kernel's uprobes, called ProfTrace, which computes the profile coverage of any app execution to address the limitations of existing tracing methods. ProfTrace requires neither changes to the system nor the app itself, and it runs on emulators as well as hardware, i.e., physical phones. The only requirements for ProfTrace are root access on the device to interact with the tracing API, as well as a kernel (4.11+) supporting uprobes (`CONFIG_UPROBE_EVENTS=y`). The flag is enabled by default for all supported kernels in the Android Open Source Project (AOSP), from Android 11 up to and including Android 16 [24], and is enabled in the recommended configuration as well. Ultimately, it is up to the manufacturer to enable it, but uprobe tracing is supported by at least the Android 14 and Android 15 emulators, as well as the Pixel 8 hardware phones, which we used for testing.

ProfTrace handles all stages of measuring profile coverage. The installation of an app uses `adb install-multiple`, which not only handles split APKs but also DEX metadata files (which include the *Cloud Profiles*), allowing us to install apps in the same way the Google Play Store app does on users' devices. This automates all necessary steps, but they can also be done manually: (1) installing one or more APKs, (2) putting the DEX metadata file in the app's data folder, and (3) triggering AOT compilation with `pm compile -m speed-profile $appid`. Afterward, the app is ready to start and all methods in the *Cloud Profile* have been compiled to binary code in an odex file.

In order to use uprobe-based event tracing, we need two things: an executable binary file and the offsets within the file of the functions we wish to trace. Fortunately, the odex file created during installation, which contains all methods declared hot or startup from the profile, is suitable, and we can use it directly as a target for tracing. ProfTrace translates the Dalvik method IDs to offsets in the binary using the built-in tools `profman` and `oatdump`. Finally, it prepares the necessary instructions for the kernel to place a probe at each function offset. We also name each probe based on the offset, which allows us to uniquely map them back to the profiled methods. When all probes are registered, we enable tracing and wait for the events to appear in an output file.

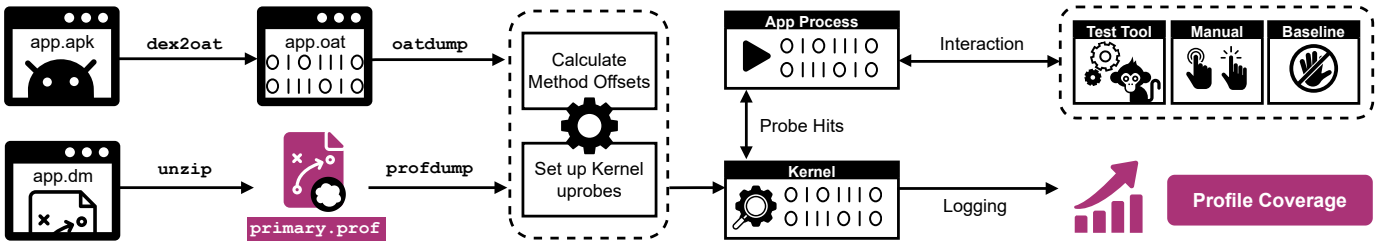


Fig. 4: *Overview of* ProfTrace*'s Data Collection.* `dex2oat` is not called directly, but as part of the installation, thus, all methods in the profile are compiled ahead-of-time (AOT) and traceable, as uprobes can only trace binary and not Dalvik code.

At this point, interaction with the app can begin and will be measured, e. g., by an automated test input generator (see Section IV-C). When the app starts, the odex file is mapped into the process memory, and the kernel is notified when a traced method is executed. This approach also has the added benefit that *any* process loading and executing the app is traced, meaning background workers and multi-threaded apps are handled by default. During app execution, each uprobe that is triggered creates a log entry with a timestamp. Currently, ProfTrace supports running a baseline configuration (No-Interaction), Monkey, DroidBot, and Fastbot2, as described in Section V-A, but can easily be extended depending on the availability of UI input generation tools and use case.

After testing finished, we disable the probes, pull the output file from the device, and process it to compute the number of unique method calls over time, i. e., the *profile coverage*. Figure 4 shows an high-level overview of ProfTrace's setup and data collection process based on the use case we selected: automated UI input generation.

C. Use Case: Automated UI Input Generation

We started our susrvey of the available state-of-the-art UI input generation approaches with Choudhary et al.'s [17] comparison of automated test input generators. While their latest evaluated SDK level was 19, which corresponds to Android 4 (Kitkat, released 2013), their classification of *random*, *model-based*, and *systematic* tools remains useful. Mao et al. [52] extended this list in 2016, and we added additional candidate tools published more recently as well. To limit the number of tools under test, we discarded any that did not receive updates after 2019 (Android 9) or were not publicly available. We found the majority of tools have not been updated for a long time and have limits that go against our tracer design: Of the publicly available tools, we discarded any that did not receive updates after 2019 [16, 43], depend on deprecated tooling (such as Python 2) [40, 57, 69], require closed-source APIs [48], cannot run on hardware devices [18], or tools that we failed to run with Android 15 (released 2024) [36, 63, 72].

Two tools required no or minimal effort to use: DroidBot [42] does not require instrumentation and works on both emulators and hardware. It models the app under test on the fly through screenshots and UI hierarchy dumps. Fastbot2 [51] also is a model-based approach that first extracts widget text labels from the APK and then uses these in a probabilistic model and reinforcement learning agent during interaction to increase activity coverage. Finally, the UI/Application Exerciser Monkey [35] is the de facto standard UI input generation approach widely used by both developers and researchers [17]. Although it only naively generates random events, it is integrated into Android and thus readily available. We thus select Monkey (*random*), DroidBot (*model-based*), and Fastbot2 (*model-based*) for our evaluation.

V. Evaluation

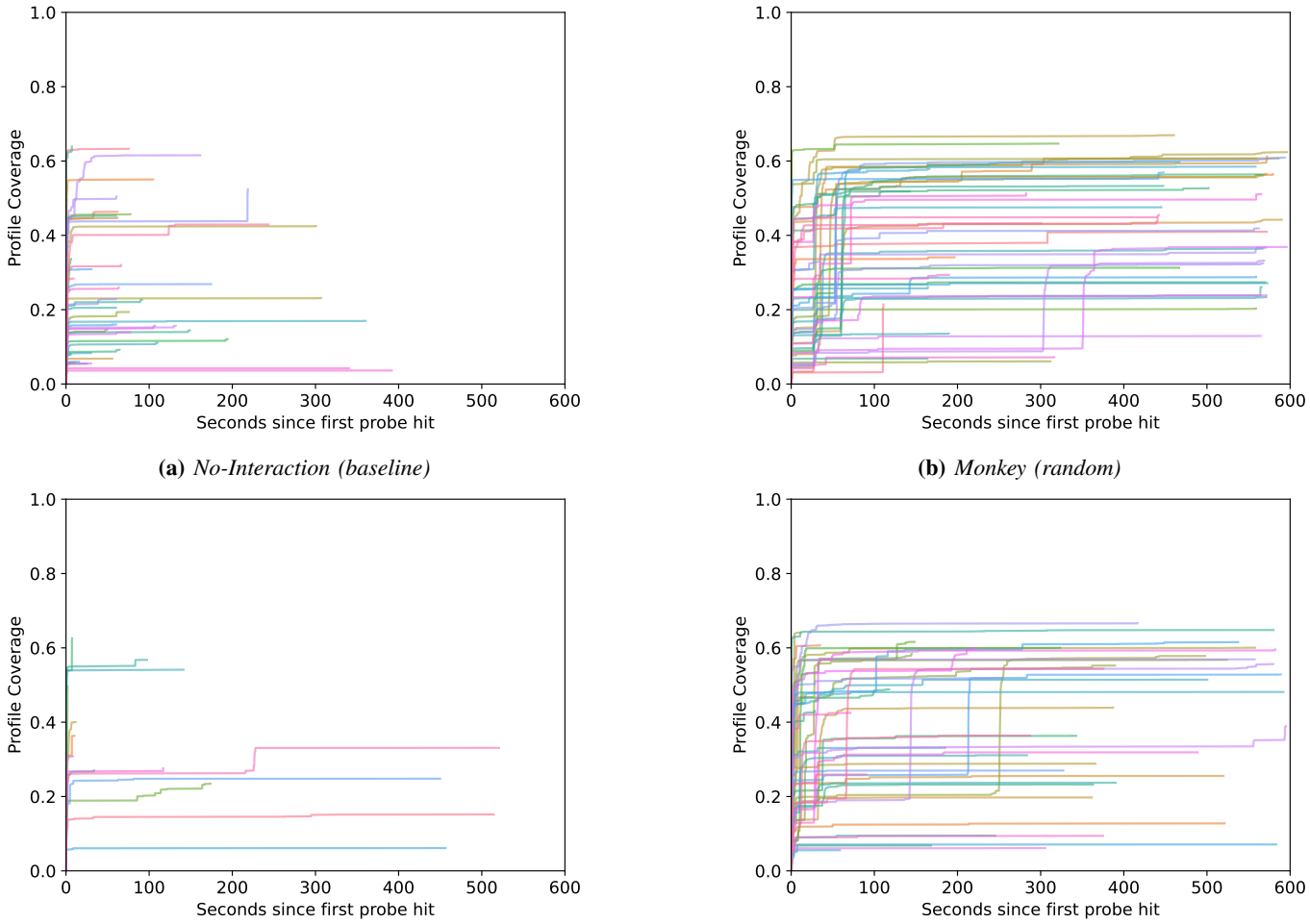
A. Experimental Setup

In addition to Monkey [35], DroidBot [42], and Fastbot2 [51], we also record a baseline by starting the target app but otherwise do not interact with it ("No-Interaction"). We parallelize our testing on four Pixel 8 phones running Android 15 (patch level BP1A.250305.019), rooted with Magisk, and set up with internet access through WiFi.

For running *Monkey*, we set a seed for the random events and throttled the interaction speed to one event per second. We also set commonly used flags to not abort the testing prematurely, such as `--ignore-crashes`. For *DroidBot*, the default settings were sufficient. *Fastbot2* required a custom patch. Despite claiming to be compatible with Android 14, it threw an irrecoverable exception when trying to rotate the screen, seemingly because of a change in the Android API. This also applies to Android 15, and due to a lack of documentation on how to adapt Fastbot2, we decided to patch out this particular event, assuming that screen rotation will not have a significant impact on coverage.

To compare profile coverage with more general code coverage, we also set up ACVTool [59]. Because it needs to rewrite the app under test and potentially changes code in a way that profiles are no longer compatible (e. g., changing DEX method IDs by adding methods), we evaluated it separately and did not run it with ProfTrace at the same time.

In a preliminary evaluation, we run all 827 apps in the no-interaction configuration for 10 seconds to establish how robust tracing is. Our ProfTrace successfully reports profile coverage for 779 (94.20 %). Of the unsuccessful apps, 29 had no activities to start (e. g., `com.google.android.safetycore`, which provides a background service for messages), 5 failed to install (e. g., `com.google.android.webview`, which reported a missing library), and 14 had a nonstandard setup or malformed files (e. g., `com.android.chrome`, which could not be uninstalled, only downgraded).



(a) *No-Interaction (baseline)*

(b) *Monkey (random)*

(c) *DroidBot (model-based)*

(d) *Fastbot2 (model-based)*

Fig. 5: *Profile Coverage of different UI interaction approaches with compatible apps.* We executed each app four times with each tool, but only show the curve with the highest overall profile coverage. The time limit was 10 minutes for all apps, and we show only the last new unique traced method per app.

We then ran ACVTool on the working 779 apps in the same no-interaction configuration, and it only managed to successfully analyze 114 (14.63 %). In 183 (23.49 %) instances, it failed to instrument the APK, 477 (61.23 %) instrumented APKs failed to be installed, and in 5 (0.64 %) cases, the recording of the coverage failed. For the final evaluation, we randomly sampled 50 of the 779 apps that work with ProfTrace iteratively until at least 15 were also in the set of 114 apps that ACVTool can successfully analyze in order to provide a robust comparison between coverage distributions.

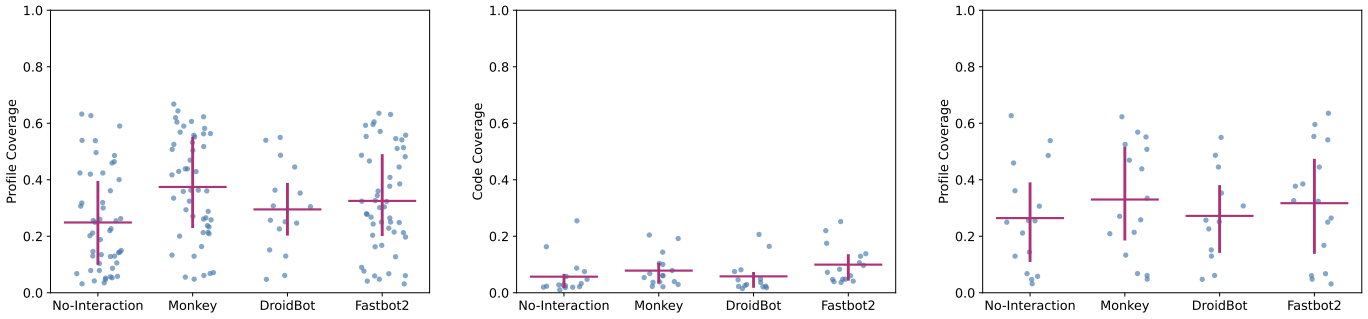
We run all four configurations of input generations for 10 minutes per app and repeat this four times. We selected this relatively short duration to reflect downstream use cases of test input generation [38, 60–62, 68], which typically favor shorter run-times for scalability reasons, in the order of a couple of minutes or even less, and because longer test durations have been shown to have a diminishing impact on the overall coverage [41]. The total time our experiments took is 6.56 days for ProfTrace and 3.51 days for ACVTool.

B. Results

We first investigate the profile coverage over the execution time of each app. Because we execute each app with each test input generator four times, we first investigate the execution with the highest achieved profile coverage. Figure 5 shows the coverage over time, with the line ending when the last unique method trace was recorded, even if the tool kept running.

Profile Coverage over Time. Most notably, DroidBot underperforms significantly. We found that one of its dependencies is outdated and fails to parse 34 of the 50 APKs under test. When the app was started but not interacted with, no app registered new traced methods after 400 seconds. For Monkey and Fastbot2, even if the number of unique methods plateaus for a while, both manage to find more unexplored methods in the apps in the later stage of testing.

Over all four recorded interactions, Monkey achieved a higher average profile coverage per app than the other presumably "smarter" model-based tools, with an average of 37.45 % of methods in the *Cloud Profile* executed. The No-



(a) *Profile Coverage (using* PROFTRACE*)*

(b) *Code Coverage (using ACVTool)*

(c) *Profile Coverage (only ACVTool-compatible apps)*

Fig. 6: *Profile Coverage vs. Code Coverage.* The horizontal line shows the mean, and the vertical line stretches from the .25 percentile to the .75 percentile. While a clear difference in profile coverage between Monkey and the other tools can be seen, the differences are not as clear between the other tools and the baseline. Since ACVTool fails to run the app under test in more than two-thirds of the cases, we additionally show the profile coverage for only the apps that ACVTool successfully analyzed.

Interaction baseline, DroidBot, and Fastbot2 achieved only 24.87 %, 29.50 %, and 32.51 %, respectively. This relatively low coverage shows that there is space for more research into how to interact with apps more efficiently and more similar to how real users are interacting with them. Additionally, it seems there is also a ceiling for profile coverage: In no instance did a test input generation tool achieve more than 70 %.

Comparison to Code Coverage. We expect code coverage to be smaller than profile coverage for two reasons: First, known limits to test input generation make it unlikely to achieve a code coverage of more than 30 % [2]. Second, the number of methods in the *Cloud Profile* is, on average, only 7.88 % of all app methods (see Section III-D).

Figure 6 shows the average profile coverage and code coverage of the four tests of each app per tool. The higher profile coverage is visible, but the difference between the tools is less pronounced for the code coverage. Here, Fastbot2 covers on average 9.95 % of an app's methods and is outperforming Monkey, which manages to cover an average of 7.83 %. DroidBot achieves a code coverage of 5.80 %, barely above the No-Interaction baseline with 5.69 %.

Additionally, the requirement to repackage apps severely limits ACVTool, only being able to trace 16 of our selected apps, but of those, DroidBot successfully interacts with 14. We selected those apps that worked with ACVTool and plotted their profile coverage again to exclude the possibility of biasing the result by excluding the apps that achieved a high profile coverage. Monkey still performs best on the subset of apps that we used for computing the code coverage. This indicates that while Fastbot2 might have achieved a similar code coverage to Monkey, the latter managed to cover a higher percentage of methods considered important enough to be compiled AOT based on aggregated usage data.

Finally, code coverage and profile coverage correlate at least moderately. The No-Interaction baseline shows the highest correlation with 0.73. DroidBot and Fastbot2 score 0.68 and 0.61, respectively. We compute the lowest correlation of 0.5 for Monkey, explained by its random exploration strategy.

RQ4: Can profiles be used to measure differences in opaque dynamic testing success? Yes, measuring profile coverage in addition to code coverage can uncover differences in behavior. It also enables comparing the coverage of dynamic testing to aggregated usage information, as we showed in the use case of dynamic UI input generators. Surprisingly, Monkey outperformed the more sophisticated model-based tools in terms of average profile coverage, executing 37.45 % of methods in the cloud profile. Moreover, our results suggest that the coverage ceiling identified in prior work also constrains profile coverage with no tool exceeding 70 % on any app.

VI. DISCUSSION

We showed that profile coverage can give additional insight into the analysis of arbitrary apps by comparing three automated test input generation tools and a baseline. We also introduce PROFTRACE, a minimally invasive and lightweight tracer to accurately measure profile coverage in practice.

Inverted Profile Coverage. While we only discussed profile coverage, *inverted profile coverage*, i. e., focusing on the methods that are not in the *Cloud Profile*, could be a fruitful direction to explore in future research. Because these methods represent less frequently executed functionality, bugs and crashes might be underrepresented in reports by users. Using inverted profile coverage could lead testing approaches and fuzzers to code that is more likely to contain unexplored edge cases and bugs, because they would be less likely to be encountered and reported by typical users.

Developer Use of Profiles. Analyzing their own app's *Cloud Profiles* could be beneficial for developers, especially when compared to the *Baseline Profiles*. If they match closely, this can give developers confidence in their approach to generate *Baseline Profiles*. Conversely, if there are stark differences, this can be a good starting point for an investigation into why their *Baseline Profiles* differ so much from actual user behavior.

Thus, digging deeper into the automated generation of these profiles could be an interesting avenue for future work.

Threats to Validity. Compilation profiles might not be available for every app just after an update. In Section III-B we showed that *Cloud Profiles* are readily available for popular apps, but according to Google's documentation, it can take up to several days for them to be prepared [30]. While we can show that the current profiles are generated on devices and the Google Play Store app informs users of their collection, we ultimately have to trust the aggregation process to represent, as the documentation claims, "*real-world user interaction with the app*". We understand this to mean that Google calculates an average over all collected profiles and as such, the resulting *Cloud Profiles* represent an average over a large group of users. This makes sense, both from an engineering and a business perspective, and lines up with the information available.

VII. Conclusion

We shine a light on the as-of-yet unexplored compilation profiles on Android by describing different types and empirically studying the *Cloud Profiles* available from the Google Play Store. Because they are aggregated from a large number of users, they can be used as a proxy for user interactions. They are easy to collect at scale and over version changes, which allows them to be used on for various use cases, from guiding UI testing approaches to helping developers understand how their app is being used.

We apply them to compute a novel metric we call *profile coverage* and evaluate three dynamic UI testing approaches. We show that profile coverage can give insights into testing results that would not be detectable with code coverage alone. To measure the profile coverage using minimally invasive techniques, we develop a method tracer, ProfTrace, based on Linux kernel uprobe events. In contrast to existing work, it does not require changes to the Android system or the app under test and works both on hardware phones and emulators.

Finally, we discuss the opportunities for future work that the *Cloud Profiles*, i.e., the readily available, aggregated usage information for Android apps, opens up.

Acknowledgements

We thank Ludwig Burtscher, whose Master thesis [15] served as the foundation of ProfTrace. This work is based on research supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22060 and Grant ID: 10.47379/ICT19056], the Austrian Science Fund (FWF) [Grant ID: 10.55776/F8515-N], the European Union's Horizon research and innovation programme under grant agreement No. 101086248 (Cloudstars), and SBA Research (SBA-K1), a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

References

[1] Y. Agman and D. Hendler. *BPFroid: Robust Real Time Android Malware Detection Framework*. 2021. DOI: 10.48550/arXiv.2105.14344. URL: http://arxiv.org/abs/2105.14344.

[2] F. Akinotcho, L. Wei, and J. Rubin. "Mobile Application Coverage: The 30% Curse and Ways Forward". In: *Proc. of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025. DOI: 10.1109/ICSE55347.2025.00142.

[3] AOSP. *art*. (line 334). 2023. URL: https://cs.android.com/android/platform/superproject/main/+/3796e88d5d9a6ecc095d32e5ebde11b9f27fef6b:art/tools/art.

[4] AOSP. *compiler_driver.cc*. (line 417). 2023. URL: https://cs.android.com/android/platform/superproject/main/+/8fd86be0bd99d39a9c2251b0da4f2a7ed7f9ec53:art/dex2oat/driver/compiler_driver.cc.

[5] AOSP. *profile_compilation_info.cc*. (lines 69–72). 2023. URL: https://cs.android.com/android/platform/superproject/main/+/f97a2749a5b48334860bb466e27b0d39e5879559:art/libprofile/profile/profile_compilation_info.cc.

[6] AOSP. *profman.cc*. (lines 164–175). 2024. URL: https://cs.android.com/android/platform/superproject/main/+/b324fd0966c713428e39c5a54a130b85f898ff26:art/profman/profman.cc.

[7] AOSP. *profman.cc*. (line 222). 2024. URL: https://cs.android.com/android/platform/superproject/main/+/7c26f98bf3d95616d516dd8cdb0ff4ef64e0b219:art/artd/path_utils.cc.

[8] AOSP. *profman.cc*. (line 207). 2024. URL: https://cs.android.com/android/platform/superproject/main/+/main:art/libprofile/profile/profile_compilation_info.h.

[9] AOSP. *profman.cc*. (line 2418). 2024. URL: https://cs.android.com/android/platform/superproject/main/+/d5137445c0d4067406cb3e38aade5507ff2fcd16:art/dex2oat/dex2oat.cc.

[10] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitraș. "When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World". In: *Proc. of the USENIX Security Symposium*. 2021.

[11] M. Backes, S. Bugiel, O. Schranz, P. Von Styp-Rekowsky, and S. Weisgerber. "ARTist: The Android Runtime Instrumentation and Security Toolkit". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017. DOI: 10.1109/EuroSP.2017.43.

[12] J. Bleier and M. Lindorfer. "Of Ahead Time: Evaluating Disassembly of Android Apps Compiled to Binary OATs Through the ART". In: *Proc. of the European Workshop on System Security (EuroSec)*. 2023. DOI: 10.1145/3578357.3591219.

[13] J. Bleier and M. Lindorfer. "Back to the Binary: Revisiting Similarities of Android Apps". In: *Phrack* 16.72 (2025). URL: https://phrack.org/issues/72/13_md#article.

[14] M. Böhme and B. Falk. "Fuzzing: On the Exponential Cost of Vulnerability Discovery". In: *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020. DOI: 10.1145/3368089.3409729.

[15] L. Burtscher. "Tracing Android Apps based on ART Ahead-Of-Time Compilation Profiles from Google Play". MA thesis. TU Wien, 2022. DOI: 10.34726/hss.2022.90745.

[16] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda. "CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes". In: *Proc. of the International Conference on Financial Cryptography and Data Security (FC)*. 2017. DOI: 10.1007/978-3-662-54970-4_13.

[17] S. R. Choudhary, A. Gorla, and A. Orso. "Automated Test Input Generation for Android: Are We There Yet?" In: *Proc. of the IEEE/ACM International Conference on Automated*

Software Engineering (ASE). 2015. DOI: 10.1109/ASE.2015.89.

[18] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury. "Time-Travel Testing of Android Apps". In: *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE)*. 2020. DOI: 10.1145/3377811.3380402.

[19] A. Druffel and K. Heid. "DaVinci: Android App Analysis Beyond Frida via Dynamic System Call Instrumentation". In: *Proc. of the Workshop on Application Intelligence and Blockchain Security (AIBlock)*. 2020. DOI: 10.1007/978-3-030-61638-0_26.

[20] Electronic Frontier Foundation (EFF). *apkeep: A command-line tool for downloading APK files from various sources*. (last updated 2025-02-22). URL: https://github.com/EFForg/apkeep.

[21] *ELLA: A Tool for Binary Instrumentation of Android Apps*. URL: https://github.com/saswatanand/ella (visited on 06/04/2024).

[22] *EMMA: A Free Java Code Coverage Tool*. URL: https://emma.sourceforge.net/ (visited on 06/04/2024).

[23] *Frida*. URL: https://frida.re/ (visited on 06/04/2024).

[24] Google. *Android Common Kernels*. (last updated 2025-04-04). 2024. URL: https://source.android.com/docs/core/architecture/kernel/android-common.

[25] Google. *Android runtime and Dalvik*. (last updated 2024-08-26). 2024. URL: https://source.android.com/docs/core/runtime.

[26] Google. *Simpleperf*. (last updated 2024-01-03). 2024. URL: https://developer.android.com/ndk/guides/simpleperf.

[27] Google. *ART Service configuration*. (last updated 2025-04-04). 2025. URL: https://source.android.com/docs/core/runtime/configure/art-service.

[28] Google. *Baseline Profiles overview*. (last updated 2025-05-30). 2025. URL: https://developer.android.com/topic/performance/baselineprofiles/overview.

[29] Google. *Boot image profiles*. (last updated 2025-04-04). 2025. URL: https://source.android.com/docs/core/runtime/boot-image-profiles.

[30] Google. *Cloud Profiles*. (last updated 2025-05-30). 2025. URL: https://developer.android.com/topic/performance/baselineprofiles/overview#cloud-profiles.

[31] Google. *Dex Layout Optimizations*. (last updated 2025-02-27). 2025. URL: https://developer.android.com/topic/performance/baselineprofiles/dex-layout-optimizations.

[32] Google. *Dex2oat options*. (last updated 2025-04-04). 2025. URL: https://source.android.com/docs/core/runtime/configure#dex2oat_options.

[33] Google. *Shrink, obfuscate, and optimize your app*. (last updated 2025-05-30). 2025. URL: https://developer.android.com/build/shrink-code.

[34] Google. *Google Play Help: App Install Optimization on Google Play*. URL: https://support.google.com/googleplay/answer/10122796 (visited on 08/28/2024).

[35] Google. *UI/Application Exerciser Monkey*. (last updated 2023-04-12). URL: https://developer.android.com/studio/test/other-testing-tools/monkey.

[36] 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 IEEE/ACM International Conference on Software Engineering (ICSE)*. 2019. DOI: 10.1109/ICSE.2019.00042.

[37] M. Ibrahim, A. Imran, and A. Bianchi. "SafetyNOT: On the Usage of the SafetyNet Attestation API in Android". In: *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2021. DOI: 10.1145/3458864.3466627.

[38] K. Kollnig, A. Shuba, R. Binns, M. V. Kleek, and N. Shadbolt. "Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps". In: *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*. 2022. DOI: 10.2478/popets-2022-0033.

[39] B. Kondracki, B. Amin Azad, N. Miramirkhani, and N. Nikiforakis. "The Droid is in the Details: Environment-aware Evasion of Android Sandboxes". In: *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 2022. DOI: 10.14722/ndss.2022.23056.

[40] 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 IEEE/ACM International Conference on Software Engineering (ICSE)*. 2024. DOI: 10.1145/3597503.3623344.

[41] Y. Lan, Y. Lu, M. Pan, and X. Li. "Navigating Mobile Testing Evaluation: A Comprehensive Statistical Analysis of Android GUI Testing Metrics". In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2024. DOI: 10.1145/3691620.3695476.

[42] Y. Li, Z. Yang, Y. Guo, and X. Chen. "DroidBot: A Lightweight UI-Guided Test Input Generator for Android". In: *Proc. of the IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. 2017. DOI: 10.1109/ICSE-C.2017.8.

[43] 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 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019. DOI: 10.1109/ASE.2019.00104.

[44] Y.-K. Lim, S. Parambil, C.-G. Kim, and S.-H. Lee. "A Selective Ahead-Of-Time Compiler on Android Device". In: *Proc. of the International Conference on Information Science and Applications (ICISA)*. 2012. DOI: 10.1109/ICISA.2012.6220938.

[45] M. Lindorfer, M. Neugschwandtner, and C. Platzer. "Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis". In: *Proc. of the IEEE Annual International Computers, Software & Applications Conference (COMPSAC)*. 2015. DOI: 10.1109/COMPSAC.2015.103.

[46] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors". In: *Proc. of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. 2014. DOI: 10.1109/BADGERS.2014.7.

[47] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. "InsDal: A Safe and Extensible Instrumentation Tool on Dalvik Byte-Code for Android Applications". In: *Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017. DOI: 10.1109/SANER.2017.7884662.

[48] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang. "Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions". In: *Proc. of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 2024. DOI: 10.1145/3597503.3639180.

[49] D. Liyanage, M. Böhme, C. Tantithamthavorn, and S. Lipp. "Reachable Coverage: Estimating Saturation in Fuzzing". In: *Proc. of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 2023. DOI: 10.1109/ICSE48619.2023.00042.

[50] D. Liyanage, S. Lee, C. Tantithamthavorn, and M. Böhme. "Extrapolating Coverage Rate in Greybox Fuzzing". In: *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE)*. 2024. DOI: 10.1145/3597503.3639198.

[51] 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 IEEE/ACM International Conference on Automated Software Engineering (ASE). 2022. DOI: 10.1145/3551349.3559505.

[52] K. Mao, M. Harman, and Y. Jia. "Sapienz: Multi-Objective Automated Testing for Android Applications". In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2016. DOI: 10.1145/2931037.2931054.

[53] B. Miranda and A. Bertolino. "Does Code Coverage Provide a Good Stopping Rule for Operational Profile Based Testing?" In: *Proc. of the International Workshop on Automation of Software Test (AST)*. 2016. DOI: 10.1109/AST.2016.012.

[54] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. "CrashScope: A Practical Tool for Automated Testing of Android Applications". In: *Proc. of the IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. 2017. DOI: 10.1109/ICSE-C.2017.16.

[55] Open Worldwide Application Security Project (OWASP). *OWASP Mobile Application Security Verification Standard (MASVS) v2.0.0*. https://github.com/OWASP/owasp-masvs/releases/tag/v2.0.0. 2023.

[56] E. Pan, J. Ren, M. Lindorfer, C. Wilson, and D. Choffnes. "Panoptispy: Characterizing Audio and Video Exfiltration from Android Applications". In: *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*. 2018. DOI: 10.1515/popets-2018-0030.

[57] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li. "Reinforcement Learning Based Curiosity-Driven Testing of Android Applications". In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2020. DOI: 10.1145/3395363.3397354.

[58] *Perfetto*. URL: https://perfetto.dev/ (visited on 06/04/2024).

[59] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw. "Fine-Grained Code Coverage Measurement in Automated Black-box Android Testing". In: *ACM Transactions on Software Engineering and Methodology* 29.4 (2020). DOI: 10.1145/3395042.

[60] A. Pradeep, M. T. Paracha, P. Bhowmick, A. Davanian, A. Razaghpanah, T. Chung, M. Lindorfer, N. Vallina-Rodriguez, D. Levin, and D. Choffnes. "A Comparative Analysis of Certificate Pinning in Android & iOS". In: *Proc. of the ACM Internet Measurement Conference (IMC)*. 2022. DOI: 10.1145/3517745.3561439.

[61] J. Ren, M. Lindorfer, D. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. "Bug Fixes, Improvements, ... and Privacy Leaks – A Longitudinal Study of PII Leaks Across Android App Versions". In: *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 2018. DOI: 10.14722/ndss.2018.23143.

[62] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman. ""Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale". In: *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*. DOI: 10.1515/popets-2018-0021.

[63] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. "Deep Reinforcement Learning for Black-Box Testing of Android Apps". In: *ACM Transactions on Software Engineering and Methodology* 31.4 (2021). DOI: 10.1145/3502868.

[64] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. "Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware". In: *Proc. of the ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS)*. 2024. DOI: 10.1145/3634737.3637658.

[65] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. "SoK: Prudent Evaluation Practices for Fuzzing". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 2024. DOI: 10.1109/SP54263.2024.00137.

[66] D. Schmidt, A. Ponticello, M. Steinböck, K. Krombholz, and M. Lindorfer. "Analyzing the iOS Local Network Permission from a Technical and User Perspective". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 2025. DOI: 10.1109/SP61157.2025.00045.

[67] M. Steinböck, J. Bleier, M. Rainer, T. Urban, C. Utz, and M. Lindorfer. "Comparing Apples to Androids: Discovery, Retrieval, and Matching of iOS and Android Apps for Cross-Platform Analyses". In: *Proc. of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 2024. DOI: 10.1145/3643991.3644896.

[68] M. Steinböck, J. Troost, W. Van Beijnum, J. Seredynski, H. Bos, M. Lindorfer, and A. Continella. "SoK: Hardening Techniques in the Mobile Ecosystem — Are We There Yet?" In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2025. DOI: 10.1109/EuroSP63326.2025.00050.

[69] 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 of the European Software Engineering Conference and the Foundations of Software Engineering (ESEC/FSE)*. 2017. DOI: 10.1145/3106237.3106298.

[70] J. Tan, S. Jiao, M. Chabbi, and X. Liu. "What Every Scientific Programmer Should Know About Compiler Optimizations?" In: *Proc. of the ACM International Conference on Supercomputing (ICS)*. 2020. DOI: 10.1145/3392717.3392754.

[71] A. Visochan, A. Stroganov, I. Titarenko, S. Lonchakov, S. Mologin, S. Pavlova, A. Lyupa, and A. Kozlova. "Method for Profile-Guided Optimization of Android Applications Using Random Forest". In: *IEEE Access* 10 (2022). DOI: 10.1109/ACCESS.2022.3214971.

[72] 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 ACM/IEEE International Conference on Software Engineering (ICSE)*. 2020. DOI: 10.1145/3377811.3380382.

[73] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. *Andrubis: Android Malware Under The Magnifying Glass*. Tech. rep. TR-ISECLAB-0414-001. TU Wien, 2014.

[74] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu. "Automated Third-Party Library Detection for Android Applications: Are We There Yet?" In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020. DOI: 10.1145/3324884.3416582.

[75] S. Zhang, H. Lei, Y. Wang, D. Li, Y. Guo, and X. Chen. "How Android Apps Break the Data Minimization Principle: An Empirical Study". In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023. DOI: 10.1109/ASE56229.2023.00141.

[76] Y. Zhang, X. Xie, Y. Li, S. Chen, C. Zhang, and X. Li. "EndWatch: A Practical Method for Detecting Non-Termination in Real-World Software". In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023. DOI: 10.1109/ASE56229.2023.00061.

[77] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci. "Towards Black Box Testing of Android Apps". In: *Proc. of the International Conference on Availability, Reliability and Security (ARES)*. 2015. DOI: 10.1109/ARES.2015.70.

[78] O. Zungur, A. Bianchi, G. Stringhini, and M. Egele. "AppJitsu: Investigating the Resiliency of Android Applications". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021. DOI: 10.1109/EuroSP51992.2021.00038.