

Fine-grained Code Coverage Measurement in Automated Black-box Android Testing

ALEKSANDR PILGUN, SnT, University of Luxembourg

OLGA GADYATSKAYA, LIACS, Leiden University

YURY ZHAUNIAROVICH, Independent Researcher

STANISLAV DASHEVSKYI, Forescout Technologies

ARTSIOM KUSHNIAROU, iTechArt Inc.

SJOUKE MAUW, CSC & SnT, University of Luxembourg

Today, there are millions of third-party Android applications. Some of them are buggy or even malicious. To identify such applications, novel frameworks for automated black-box testing and dynamic analysis are being developed by the Android community. Code coverage is one of the most common metrics for evaluating effectiveness of these frameworks. Furthermore, code coverage is used as a fitness function for guiding evolutionary and fuzzy testing techniques. However, there are no reliable tools for measuring fine-grained code coverage in black-box Android app testing.

We present the Android Code Coverage Tool, ACVTool for short, that instruments Android apps and measures code coverage in the black-box setting at class, method and instruction granularity. ACVTool has successfully instrumented 96.9% of apps in our experiments. It introduces a negligible instrumentation time overhead, and its runtime overhead is acceptable for automated testing tools. We demonstrate practical value of ACVTool in a large-scale experiment with Sapienz, a state-of-art automated testing tool. Using ACVTool on the same cohort of apps, we have compared different coverage granularities applied by Sapienz in terms of the found amount of crashes. Our results show that none of the applied coverage granularities clearly outperforms others in this aspect.

CCS Concepts: • **Security and privacy** → **Mobile platform security**; • **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**.

Additional Key Words and Phrases: Android, Automated Software Testing, Code Coverage, Instrumentation

ACM Reference Format:

Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskiy, Artsiom Kushniarou, and Sjouke Mauw. 2020. Fine-grained Code Coverage Measurement in Automated Black-box Android Testing. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (July 2020), 37 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Authors' addresses: Aleksandr Pilgun, SnT, University of Luxembourg, aleksandr.pilgun@uni.lu; Olga Gadyatskaya, LIACS, Leiden University, o.gadyatskaya@liacs.leidenuniv.nl; Yury Zhauniarovich, Independent Researcher, yury@zhauniarovich.com; Stanislav Dashevskiy, Forescout Technologies, stanislav.dashevskiy@forescout.com; Artsiom Kushniarou, iTechArt Inc., artsiom.kushniarou@itechart-group.com; Sjouke Mauw, CSC & SnT, University of Luxembourg, sjouke.mauw@uni.lu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1049-331X/2020/7-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Code coverage measurement is an essential element of software development and quality assurance cycles for all programming languages and ecosystems, including Android. It is routinely applied by developers, testers, and analysts to understand the degree to which the system under test has been evaluated [4], to generate test cases [73], to compare test suites [24], and to maximize fault detection by prioritizing test cases [75]. In the context of Android application analysis, code coverage has become a critical metric. Fellow researchers and practitioners evaluate the effectiveness of tools for automated testing [18, 38, 51, 67] and security analysis [35, 50] using code coverage, among other metrics. It is also used as a fitness function to guide application exploration in testing [39, 51, 62].

Unfortunately, the Android ecosystem introduces a particular challenge for security and reliability analysis: Android applications (apps for short) submitted to markets (e.g., Google Play) have been already compiled and packaged, and their source code is often unavailable for inspection. Measuring code coverage achieved in testing and analysis is not a trivial endeavor in this setting. This is why some third-party app testing systems, e.g., [17, 48, 57], use open-source apps for experimental validation, whereby the source code coverage could be measured by popular tools developed for Java, such as EMMA [56] or JaCoCo [37]. These, and other systems will benefit from a reliable tool for measuring code coverage in testing third-party Android apps.

In the absence of source code, code coverage is usually measured by instrumenting the bytecode of applications [41]. Within the Java community, the problem of code coverage measurement at the bytecode level is well-developed and its solution is considered to be relatively straightforward [41, 64]. However, while Android applications are written in Java, they are compiled into bytecode for the register-based Dalvik Virtual Machine (DVM), which is quite different from the Java Virtual Machine (JVM). Thus, there are significant disparities in the bytecode for these two virtual machines.

Since the arrangement of the Dalvik bytecode complicates the instrumentation process [35], there have been so far only few attempts to track code coverage for Android applications at the bytecode level [77], and they all still have limitations. The most significant one is the *coarse granularity* of the provided code coverage metric. For example, ELLA [23], InsDal [46] and CovDroid [74] measure code coverage only at the method level. Another limitation of the existing tools is the *low percentage* of successfully instrumented apps. For instance, the tools by Huang et al. [35] and Zhauniarovich et al. [80] support fine-grained code coverage metrics, but they could successfully instrument only 36% and 65% of applications from their evaluation samples, respectively. Unfortunately, such instrumentation success rates are prohibitive for these tools to be widely adopted by the Android community. Furthermore, the existing tools suffer from *limited empirical evaluation*, with a typical evaluation dataset being less than 100 apps. Sometimes, research papers do not even mention the percentage of failed instrumentation attempts (e.g., [13, 46, 74]).

Remarkably, in the absence of reliable fine-grained code coverage reporting tools, some frameworks integrate their own black-box code coverage measurement libraries, e.g., [13, 40, 48, 51, 60]. However, as code coverage measurement is not the core contribution of these works, the authors do not provide detailed information about the rates of successful instrumentation, as well as other details related to the code coverage performance of these libraries.

In this paper, we present ACVTool – the Android Code coVerage measurement Tool that does not suffer from the aforementioned limitations. The paper makes the following contributions:

- An approach to instrument Dalvik bytecode in its `smali` representation by inserting probes to track code coverage at the levels of classes, methods and instructions. Our approach is fully self-contained and transparent to the testing environment.
- An implementation of the instrumentation approach in ACVTool, which can be integrated with any testing or dynamic analysis framework. Our tool presents the coverage measurements and information about encountered crashes as handy reports that can be either visually inspected by an analyst, or processed by an automated testing environment.
- Extensive empirical evaluation that shows the high reliability and versatility of our approach.
 - While previous works [35, 80] have only reported the number of successfully instrumented apps¹, we also verified whether apps can be successfully executed after instrumentation. We report that **96.9%** have been successfully executed on the Android emulator, which is only 0.9% less than the initial set of successfully instrumented apps.
 - In the context of automated and manual application testing, ACVTool introduces only a **negligible instrumentation time overhead**. In our experiments ACVTool required on average 33.3 seconds to instrument an app.
 - The **runtime overhead introduced by ACVTool is very low** for real apps: in experiments with real Android apps the mean CPU overhead introduced by ACVTool is 0.53%.
 - We have evaluated whether ACVTool reliably measures the bytecode coverage by comparing its results with those reported by JaCoCo [37] and ELLA [23]. Our results show that the ACVTool results can be **trusted**, as code coverage statistics reported by ACVTool, JaCoCo and ELLA are highly correlated.
 - By integrating ACVTool with Sapienz [51], an efficient automated testing framework for Android, we demonstrate that our tool can be **useful** as an integral part of an automated testing or security analysis environment. With ACVTool, we were able to compare how different coverage metrics fare in bug finding with Sapienz. We show that different coverage granularities do not tend to find the same crashes, but none of them clearly outperforms the others. The question of the right granularity of code coverage to be used in search-based testing remains open.
- We release ACVTool as an **open-source tool** to support the Android testing and analysis community. Source code and a demo video of ACVTool are available for the community².

ACVTool can be readily used with various dynamic analysis and automated testing tools, e.g., IntelliDroid [70], CopperDroid [63], Sapienz [51], Stoa [62], DynoDroid [49], CuriousDroid [15], PATDroid [57], Paladin [48], to mention a few, to measure code coverage. This work extends our preliminary results reported in [22, 54].

This paper is structured as follows. We give the necessary background information about Android applications and their code coverage measurement aspects in Section 2. The ACVTool design and workflow are presented in Section 3. Section 4 details our bytecode

¹For ACVTool, it is 97.8% out of 1278 real-world Android apps.

²<https://github.com/pilgun/acvtool>

instrumentation approach. In Section 5, we evaluate the effectiveness and efficiency of ACVTool and assess how the coverage data reported by ACVTool is compliant to the data measured by the JaCoCo system on the source code and Ella without source code. Section 6 presents our results on integrating ACVTool with the Sapienz automated testing framework, evaluates the impact of ACVTool instrumentation on app runtime behavior, and discusses the contribution of code coverage data to bug finding in Android apps. Then we discuss the limitations of our prototype and potential threats to validity for our empirical findings in Section 7. We provide an overview of related work and compare ACVTool to the existing tools for black-box Android code coverage measurement in Section 8. We conclude with Section 9.

2 BACKGROUND

2.1 APK Internals

Android apps are distributed as *apk* packages that contain the resource files, native libraries (**.so*), compiled code files (**.dex*), manifest (`AndroidManifest.xml`), and developer's signature. Typical application resources are user interface layout files and multimedia content (icons, images, sounds, videos, etc.). Native libraries are compiled C/C++ modules that are often used for speeding up computationally intensive operations.

Android apps are usually developed in Java and, more recently, in Kotlin – a JVM-compatible language [19]. Upon compilation, code files are first transformed into Java bytecode files (**.class*), and then converted into a Dalvik executable file (`classes.dex`) that can be executed by the Dalvik/ART Android virtual machine (DVM). Usually, there is only one `dex` file, but Android also supports multiple `dex` files. Such apps are called multidex applications.

In contrast to most JVM implementations that are stack-based, DVM is a register-based virtual machine. It assigns local variables to registers, and the DVM instructions (opcodes) directly manipulate the values stored in the registers. Each application method has a set of registers defined in its beginning, and all computations inside the method can be done only through this register set. The method parameters are also a part of this set. The parameter values sent into the method are always stored in the registers at the end of the method's register set. For more details, we refer the interested reader to the official Android documentation about the Dalvik bytecode internals [25] and the presentation by Bornstein [12].

Since raw Dalvik binaries are hard to understand for humans, several intermediate representations have been proposed that are more analyst-friendly: `smali` [10, 28] and `Jimple` [65]. In this paper, we work with `smali`, a low-level programming language for the Android platform. `Smali` is supported by Google [28], and it can be viewed and manipulated using, e.g., the `smalidea` plugin for the IntelliJ IDEA/Android Studio [10].

The Android *manifest* file is used to set up various parameters of an app (e.g., whether it has been compiled with the `debug` flag enabled), to list its components, and to specify the set of declared and requested Android permissions. The manifest provides a feature that is very important for the purpose of this paper: it allows one to specify the instrumentation class that can monitor at runtime all interactions between the Android system and the app. We rely upon this functionality to enable the code coverage measurement, and to intercept the crashes of an app and log their details.

Before an app can be installed onto a device, it must be cryptographically signed with a developer's certificate (the signature is located under the `META-INF` folder inside an `.apk`

file) [79]. The purpose of this signature is to establish the trust relationship between the apps of the same signature holder: for example, it ensures that the application updates are delivered from the same developer. Still, these signatures cannot be used to verify the authenticity of the developer of an application being installed, as other parties can modify the contents of the original application and re-sign it with their own certificates. Our approach relies on this possibility of code re-signing to instrument the apps.

2.2 Code Coverage

The notion of *code coverage* refers to the metrics that help developers to estimate the portion of the source code or the bytecode of a program executed at runtime, e.g., while running a test suite [4]. Coverage metrics are routinely used in the white-box testing setting, when the source code is available. They allow developers to estimate the relevant parts of the source code that have never been executed by a particular set of tests, thus facilitating, e.g., regression-testing and improvement of test suites. Furthermore, code coverage metrics are regularly applied as components of fitness functions that are used for other purposes: fault localization [64], automatic test generation [51], and test prioritization [64].

In the Android realm, not only application developers are interested in measuring code coverage. For example, Google tests all submitted (already packaged) apps to ensure that they meet the security standards³. For independent testers and analysts it is important to understand how well a third-party app has been exercised [38], and various third-party app testing and analysis tools are routinely evaluated with respect to the achieved code coverage [18, 35, 38, 67].

There exist several levels of *granularity* at which the code coverage can be measured. *Statement* coverage, *basic block coverage*, and *function (method)* coverage are very widely used. Other coverage metrics exist as well: *branch*, *condition*, *parameter*, *data-flow*, etc [4]. However, these metrics are rarely used within the Android community, as they are not widely supported by the most popular coverage tools for Java and Android source code, namely JaCoCo [37] and EMMA [56]. On the other hand, the Android community often uses the *activity* coverage metric, that counts the proportion of executed activities [7, 15, 51, 77] (classes of Android apps that implement the user interface), because this metric is useful and is relatively easy to compute.

There is an important distinction in measuring the statement coverage of an app at the source code and at the bytecode levels: the instructions and methods within the bytecode may not exactly correspond to the instructions and methods within the original source code. For example, a single source code statement may correspond to several bytecode instructions [12]. Also, a compiler may optimize the bytecode so that the number of methods is different, or the control flow structure of the app is altered [41, 64].

It is not always possible to map the source code statements to the corresponding bytecode instructions without having the debug information. Therefore, it is practical to expect that the source code statement coverage cannot be reliably measured within the third-party app testing scenario, and with ACVTool we resort to measuring the bytecode instruction coverage. We call the third-party app testing scenario *black-box testing*, to emphasize the absence of source code and implementation details. This terminology is standard in the Android community [38].

³<https://www.android.com/security-center/>

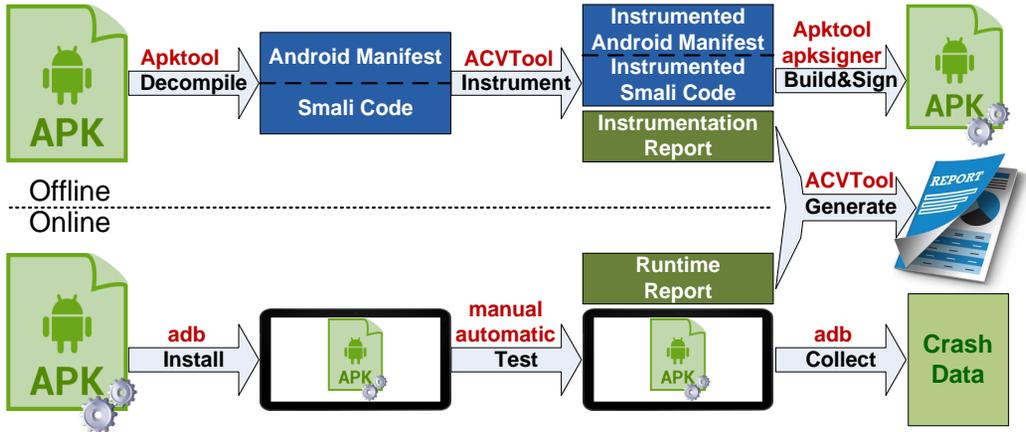


Fig. 1. ACVTool workflow

3 ACVTOOL DESIGN

ACVTool allows one to *measure* and *analyze* the degree to which the code of a *closed-source* Android app is executed during testing, and to *collect crash reports* occurred during this process. We have designed the tool to be self-contained by embedding all dependencies required to collect the runtime information into the application under test (AUT). Therefore, our tool does not require to install additional software components, allowing it to be effortlessly integrated into any existing testing or security analysis pipeline. For instance, we have tested ACVTool with the random input event generator Monkey [30], and we have integrated it with the Sapienz tool [51] to experiment with fine-grained coverage metrics (see details in Section 6). Furthermore, for instrumentation ACVTool uses only the instructions available on all current Android platforms. The instrumented app is thus compatible with all emulators and devices. We have tested whether the instrumented apps work using an Android emulator and a Google Nexus phone.

Figure 1 illustrates the workflow of ACVTool that consists of three phases: *offline*, *online* and *report generation*. At the time of the offline phase, the app is instrumented and prepared for running on a device or an emulator. During the online phase, ACVTool installs the instrumented app, runs it and collects its runtime information (coverage measurements and crashes). At the report generation phase, the runtime information of the app is extracted from the device and used to generate a coverage report. Below we describe these phases in detail.

3.1 Offline Phase

The offline phase of ACVTool is focused on app instrumentation. In a nutshell, this process consists of several steps depicted in the upper part of Figure 1. The original Android app is first decompiled using `apktool` [68]. Under the hood, `apktool` uses the `smali/backsmali` disassembler [10] to disassemble `.dex` files and transform them into `smali` representation. To track the execution of the original `smali` instructions, ACVTool inserts special *probe* instructions after each of them. These probes are invoked right after the corresponding original instructions, allowing us to precisely track their execution at runtime. After the instrumentation, ACVTool compiles the instrumented version of the app using `apktool` and signs it with `apksigner`. Thus, by relying on native Android tools and some well-supported

tools provided by the community, ACVTool is able to instrument almost every app. We present the details of our instrumentation process in Section 4.

In order to collect the runtime information, we used the approach proposed in [80] and developed a dedicated `Instrumentation` class. ACVTool embeds this class into the app code, allowing the tool to collect the runtime information. After the app has been tested, this class serializes the runtime information (represented as a set of boolean arrays) into a binary representation, and saves it to the external storage of an Android device. The `Instrumentation` class also collects and saves the data about crashes within the AUT, and registers a broadcast receiver. The receiver waits for a special event notifying that the process collecting the runtime information should be stopped. Therefore, various testing tools can use the standard Android broadcasting mechanism to control ACVTool externally.

ACVTool makes several changes to the Android manifest file (decompiled from binary to normal xml format by `apktool`). First, to write the runtime information to the external storage, we additionally request the `WRITE_EXTERNAL_STORAGE` permission. Second, we add a special `instrument` tag that registers our `Instrumentation` class as an instrumentation entry point.

After the instrumentation is finished, ACVTool assembles the instrumented package with `apktool`, re-signs and aligns it with standard Android utilities `apksigner` and `zipalign`. Thus, the offline phase yields an instrumented app that can be installed onto a device and executed.

It should be mentioned that we sign the application with a new signature. Therefore, if the application checks the validity of the signature at runtime, the instrumented application may fail or run with reduced functionality, e.g., it may show a message to the user that the application is repackaged and may not work properly.

Along with the instrumented apk file, the *offline* phase produces an *instrumentation report*. It is a serialized code representation saved into a binary file with the `pickle` extension that is used to map probe indices in a binary array to the corresponding original bytecode instructions. This data along with the runtime report (described in Section 3.2) is used during the *report generation* phase. By default, ACVTool instruments an application to collect instruction-, method- and class-level coverage information. It is also possible to instrument an app to collect only method- and class-level coverage data, in case only a coarser-grained coverage information is required.

3.2 Online Phase

During the online phase, ACVTool installs the instrumented app onto a device or an emulator using the `adb` utility, and initiates the process of collecting the runtime information by starting the `Instrumentation` class. This class is activated through a command issued to `adb`. Developers can then test the app manually, run a test suite, or interact with the app in any other way, e.g., by running tools, such as Monkey [30], IntelliDroid [70], or Sapienz [51]. ACVTool's data collection does not influence the app execution. If the `Instrumentation` class has been not activated, the app can still be run in a normal way.

After the testing is over, ACVTool generates a broadcast that instructs the `Instrumentation` class to stop the coverage data collection. Upon receiving the broadcast, the class consolidates the runtime information into a *runtime report* and stores it on the external storage of the testing device. Additionally, ACVTool keeps the information about all crashes of the AUT, including the timestamp of a crash, the name of the class that crashed, the corresponding error message and the full stack trace. By default, ACVTool is configured to catch all runtime

Element	Ratio	Cov.	Missed	Lines	Missed	Methods	Missed	Classes
 AndroidLauncher\$EUCountry.smali		98.48943%	5	331	1	5	0	1
 AndroidLauncher.smali		68.83117%	144	462	18	35	0	1
 BuildConfig.smali		0.00000%	1	1	1	1	1	1
 MyApplication\$TrackerName.smali		80.64516%	6	31	2	4	0	1
 MyApplication.smali		86.11111%	5	36	0	2	0	1
 R\$anim.smali		0.00000%	1	1	1	1	1	1
 SnakeGame.smali		55.55556%	16	36	0	2	0	1

Fig. 2. ACVTool *html* report

```

.method public constructor <init>(Lcom/gnsdm/snake/managers/ActionResolver;Z)V
    .locals 0
    .param p1, "resolver" # Lcom/gnsdm/snake/managers/ActionResolver;
    .param p2, "tabletSize" # Z

    invoke-direct {p0}, Lcom/badlogic/gdx/Game;->()V
    iput-boolean p2, p0, Lcom/gnsdm/snake/SnakeGame;->tabletSize:Z
    iput-object p1, p0, Lcom/gnsdm/snake/SnakeGame;->resolver:Lcom/gnsdm/snake/managers/ActionResolver;
    return-void
.end method

```

Fig. 3. Covered *smali* instructions highlighted by ACVTool

exceptions in an AUT without stopping its execution – this can be useful for collecting the code coverage information right after a crash happens, helping to pinpoint its location.

3.3 Report Generation Phase

The *runtime report* is a set of boolean vectors (with all elements initially set to **False**); each of these vectors corresponds to one class of the app. Every element of a vector maps to a probe that has been inserted into the class. Once a probe has been executed, the corresponding vector's element is set to **True**, meaning that the associated instruction has been covered. To build the *coverage report* that shows what original instructions have been executed during the testing, ACVTool uses data from the *runtime report*, showing what probes have been invoked at runtime, and from the *instrumentation report* that maps these probes to original instructions.

Currently, ACVTool generates reports in the *html* and *xml* formats. These reports have a structure similar to the reports produced by the JaCoCo tool [37]. While *html* reports are convenient for visual inspection, *xml* reports are more suitable for automated processing. Figure 2 shows an example of a *html* report. Analysts can browse this report and navigate the hyperlinks that direct to the *smali* code of individual files of the app, where the covered *smali* instructions are highlighted (as shown in Figure 3).

4 CODE INSTRUMENTATION

In this section, we describe the bytecode instrumentation approach used in ACVTool. In the literature, Huang et al. [35] propose two approaches for measuring bytecode coverage: (1) *direct instrumentation* by placing probes right after the instruction that has to be monitored for coverage (this requires using additional registers); (2) *indirect instrumentation* by wrapping probes into separate functions. The latter instrumentation approach introduces significant overhead in terms of added methods that could potentially lead to reaching the

upper limit of method references per `.dex` file (65536 methods, see [27]). Thus, we built ACVTool upon the former approach.

```

1 private void updateElements() {
2     boolean updated = false;
3     while (!updated) {
4         updated = updateAllElements();
5     }
6 }

```

Listing 1. *Java* code example.

```

1 .method private updateElements()V
2 .locals 1
3     const/4 v0, 0x0
4     .local v0, "updated":Z
5     :goto_0
6     if-nez v0, :cond_0
7     invoke-direct {p0}, Lcom/demo/Activity;-.>updateAllElements()Z
8     move-result v0
9     goto :goto_0
10    :cond_0
11    return-void
12 .end method

```

Listing 2. *Smali* representation of the original *Java* code example.

4.1 Bytecode representation

To instrument Android apps, ACVTool relies on the `apkil` library [72] that creates a tree-based structure of `smali` code. The tree generated by `apkil` contains classes, fields, methods, and instructions as nodes. It also maintains relations between instructions, labels, `try-catch` and `switch` blocks. We use this tool for two purposes: (1) `apkil` builds a structure representing the code that facilitates bytecode manipulations; (2) it maintains links to the inserted probes, allowing us to generate the code coverage report.

The original `apkil` library has not been maintained since 2013. Therefore, we adapted it to enable support for more recent versions of Android. In particular, we added annotation support for classes and methods, which has appeared in the Android API 19, and has been further extended in the API 22. Our modifications specify the `.annotation` word and its structure for classes and methods for the `apkil smali` parser. Other our additions to `apkil` contain 4 new instructions: `filled-new-array`, `invoke-custom`, `filled-new-array/range` and `invoke-custom/range`. We added them to the list of 35c and 3rc Dalvik instruction formats⁴. For parsing, `apkil` finds such instructions by name as they have a different format compared to other instructions [27]. Thus, the `apkil` library evolves according to ACVTool's needs, and it is maintained within the ACVTool project.

Tracking the bytecode coverage requires not only to insert the probes while keeping the bytecode valid, but also to maintain the references between the original and the instrumented bytecode. For this purpose, when we generate the `apkil` representation of the original bytecode, we annotate the nodes that represent the original bytecode instructions with additional information about the probes we inserted to track their execution. We then save

⁴See <https://source.android.com/devices/tech/dalvik/instruction-formats> for more details about instruction formats.

```

1 .method private updateElements()V
2 .locals 4
3   move-object/16 v1, p0
4   sget-object v2, Lcom/acvtool/StorageClass;-->Activity1267:Z
5   const/16 v3, 0x1
6   const/16 v4, 0x9
7   aput-boolean v3, v2, v4
8   const/4 v0, 0x0
9   goto/32 :goto_hack_4
10  :goto_hack_back_4
11  :goto_0
12  goto/32 :goto_hack_3
13  :goto_hack_back_3
14  if-nez v0, :cond_0
15  goto/32 :goto_hack_2
16  :goto_hack_back_2
17  invoke-direct {v1}, Lcom/demo/Activity;-->updateAllElements()Z
18  move-result v0
19  goto/32 :goto_hack_1
20  :goto_hack_back_1
21  goto :goto_0
22  :cond_0
23  goto/32 :goto_hack_0
24  :goto_hack_back_0
25  return-void
26  :goto_hack_0
27  const/16 v4, 0x4
28  aput-boolean v3, v2, v4
29  goto/32 :goto_hack_back_0
30  :goto_hack_1
31  const/16 v4, 0x5
32  aput-boolean v3, v2, v4
33  goto/32 :goto_hack_back_1
34  :goto_hack_2
35  const/16 v4, 0x6
36  aput-boolean v3, v2, v4
37  goto/32 :goto_hack_back_2
38  :goto_hack_3
39  const/16 v4, 0x7
40  aput-boolean v3, v2, v4
41  goto/32 :goto_hack_back_3
42  :goto_hack_4
43  const/16 v4, 0x8
44  aput-boolean v3, v2, v4
45  goto/32 :goto_hack_back_4
46 .end method

```

Listing 3. Instrumented *smali* code example. The highlighted lines mark the added instructions.

this annotated intermediate representation of the original bytecode into a separate serialized `.pickle` file as the instrumentation report.

4.2 Register management

To exemplify how our instrumentation works, Listing 1 gives an example of a Java code fragment, Listing 2 shows its *smali* representation, and Listing 3 illustrates the corresponding *smali* code instrumented by ACVTool.

The probe instructions that we insert are simple `aput-boolean` opcode instructions (e.g., Line 7 in Listing 3). These instructions put a boolean value (the first argument of the opcode instruction) into an array identified by a reference (the second argument), to a certain cell at an index (the third argument). Therefore, to store these arguments we need to allocate three additional registers per `aput-boolean` method.

The addition of these registers is not a trivial task. We cannot simply use the first three registers in the beginning of the stack because this will require modification of the remaining method code and changing the corresponding indices of the registers. Moreover, some instructions can address only 16 registers [27]. Therefore, the addition of new registers could make these instructions malformed. Similarly, we cannot easily use new registers at the end of the stack because method parameter registers must always be the last ones.

To overcome this issue, we use the following approach. We allocate three new registers, however, in the beginning of a method we copy the values of the argument registers to their corresponding places in the original method. For instance, in Listing 3 the instruction at Line 3 copies the value of the parameter `p0` into the register `v1` that has the same register position as in the original method (see Listing 2). Depending on the value type, we use different `move` instructions for copying: `move-object/16` for objects, `move-wide/16` for paired registers (Android uses register pairs for `long` and `double` types), `move/16` for others. Then we update all occurrences of parameter registers through the method body from `p` names to their `v` aliases (compare Line 7 in Listing 2 with Line 17 in Listing 3). Afterwards, the last 3 registers in the stack are safe to use for the probe arguments (for instance, see Lines 4-6 in Listing 3).

4.3 Probes insertion

Apart from moving the registers, there are other issues that must be addressed for inserting the probes correctly. First, it is impractical to insert probes after certain instructions that change the the execution flow of a program, namely `return`, `goto` (line 21 in Listing 3), and `throw`. If a probe was placed right after these instructions, it would never be reached during the program execution.

Second, some instructions come in pairs. For instance, the `invoke-*` opcodes, which are used to invoke a method, must be followed by the appropriate `move-result*` instruction to store the result of the method execution [27] (see Lines 17-18 in Listing 3). Therefore, we cannot insert a probe between them. Similarly, in case of an exception, the result must be immediately handled. Thus, a probe cannot be inserted between the `catch` label and the `move-exception` instruction.

These aspects of the Android bytecode mean that we insert probes after each instruction, but not after the ones modifying the execution flow, and not after the first command in the paired instructions. These excluded instructions are *untraceable* for our approach, and we do not consider them to be part of the resulting code coverage metric. Note that in case of a method invocation instruction, we log each invoked method, so that the computed method code coverage will not be affected by this.

The `VerifyChecker` component of the Android Runtime that checks the code validity at runtime poses additional challenges. For example, a Java `synchronized` block, which allows a particular code section to be executed by only one thread at a time, corresponds to a pair of the `monitor-enter` and `monitor-exit` instructions in the Dalvik bytecode. To ensure that the lock is eventually released, this instruction pair is wrapped with an implicit `try-catch` block, where the `catch` part contains an additional `monitor-exit` statement. Therefore, in case of an exception inside a lock, another `monitor-exit` instruction will unlock the thread. `VerifyChecker` ensures that the `monitor-exit` instruction will be executed only once, so it does not allow to add any instructions that may potentially raise an exception. To overcome this limitation, we insert the `goto/32` statement to redirect the flow to the tracking instruction, and a label to go back after the tracking instruction was executed. Since `VerifyChecker` examines the code sequentially, and the `goto/32` statement is not

considered as a statement that may throw exceptions, our approach allows the instrumented code to pass the code validity check.

5 EVALUATION

Our code coverage tracking approach modifies the app bytecode by adding probes and repackaging the original app. This approach could be deemed too intrusive to use with the majority of third-party applications. To prove the validity and the practical usefulness of our tool, we have performed an extensive empirical evaluation of ACVTool with respect to the following criteria:

Effectiveness. We report the instrumentation success rate of ACVTool, broken down in the following numbers:

- *Instrumentation success rate.* We report how many apps from our datasets have been successfully instrumented with ACVTool.
- *App health after instrumentation.* We measure the percentage of instrumented apps that can run on an emulator. We call these apps *healthy*⁵. To report this statistic, we installed the instrumented apps on the Android emulator and launched their main activity. If an app is able to run for 3 seconds without crashing, we count it as healthy.

Efficiency. We assess the following characteristics:

- *Instrumentation-time overhead.* Traditionally, the preparation of apps for testing is considered to be an *offline* activity that is not time-sensitive. Given that the testing process may be time-demanding (e.g., Sapienz [51] tests each application for hours), our goal is to ensure that the instrumentation time is insignificant in comparison to the testing time. Therefore, we have measured the time ACVTool requires to instrument apps in our datasets.
- *Runtime overhead.* Tracking instructions added into an app introduce their own runtime overhead, what may be a critical issue in testing. Therefore, we evaluate the impact of the ACVTool instrumentation on app performance and codebase size. We quantify the runtime overhead measured as the CPU utilization overhead on a subset of applications and on the benchmark PassMark application [59] by comparing executions of original and instrumented app versions. We also measure the increase in the `.dex` file size.

Compliance with other tools. We compare the coverage data reported by ACVTool with the coverage data measured by JaCoCo [37] that relies on the white-box approach and requires source code, and by Ella [23], which does not require source code, but measures coverage only at the method level. This comparison allows us to draw conclusions about the reliability of the coverage information collected by ACVTool.

To the best of our knowledge, this is the largest empirical evaluation of a code coverage tool for Android done so far. In the remainder of this section, after presenting the benchmark application sets used, we report on the results obtained in dedicated experiments for each of the above criteria. The experiments were executed on an Ubuntu server (Xeon 4114, 2.20GHz, 128GB RAM).

5.1 Benchmark

We downloaded 1000 apps from the Google Play sample of the AndroZoo dataset [2]. These apps were selected randomly among apps built after Android API 22 was released, i.e., after

⁵To the best of our knowledge, we are the first to report the percentage of instrumented apps that are healthy.

Table 1. ACVTool performance evaluation

Parameter	Google Play benchmark	F-Droid benchmark	Total
Total # healthy apps	832	446	1278
Instrumented apps	809 (97.2%)	442 (99.1%)	1251 (97.8%)
Healthy instrumented apps	799 (96.0%)	440 (98.7%)	1239 (96.9%)
Avg. instrumentation time	36.6 sec	27.4 sec	33.3 sec

November 2014. These are real third-party apps that may use obfuscation and anti-debugging techniques, and could be more difficult to instrument.

Among the 1000 Google Play apps, 168 could not be launched: 12 apps were missing a launchable activity, 1 had encoding problem, and 155 crashed upon startup. These crashes could appear due to some misconfigurations in the apps, but also due to the fact that we used an emulator. Android emulators lack many features present in real devices. We have used the emulator, because we subsequently test ACVTool together with Sapienz [51] (these experiments are reported in the next section). We excluded these unhealthy apps from our sample. In total, our **Google Play benchmark** contains **832** healthy apps. The apk sizes in this set range from 20KB to 51MB, with an average apk size of 9.2MB.

As one of our goals is to evaluate the reliability of the coverage data collected by ACVTool comparing to JaCoCo as a reference, we need to have some apps with the available source code. To collect such apps, we use the F-Droid⁶ dataset of open source Android apps (1330 application projects as of November 2017). We could `git clone` 1102 of those, and found that 868 apps used Gradle as a build system. We have successfully compiled 627 apps using 6 Gradle versions⁷.

To ensure that all of these 627 apps can be tested (*healthy apps*), we installed them on an Android emulator and launched their main activity for 3 seconds. In total, out of these 627 apps, we obtained **446** healthy apps that constitute our **F-Droid benchmark**. The size of the apps in this benchmark ranges from 8KB to 72.7MB, with an average size of 3.1MB.

5.2 Effectiveness

5.2.1 Instrumentation success rate. Table 1 summarizes the main statistics related to the instrumentation success rate of ACVTool.

Before instrumenting applications with ACVTool, we first reassembled, repackaged, rebuilt (with `apktool`, `zipalign`, and `apksigner`) and installed every healthy Google Play and F-Droid app on a device. From the Google Play sample, one repackaged app crashed upon startup, and `apktool` could not repackage 22 apps, raising `AndroLibException`. From the F-Droid sample, `apktool` was unable to repackage only one app. These apps were excluded from subsequent experiments, and we consider them as failures for ACVTool (even though ACVTool instrumentation did not cause these failures).

Besides the 24 apps that could not be repackaged in both app sets, ACVTool has instrumented all remaining apps from the Google Play benchmark. Yet, it failed to instrument 3 apps from the F-Droid set. The found issues were the following: in 2 cases `apktool` raised an exception `ExceptionWithContext` declaring an invalid instruction offset, in 1 case `apktool`

⁶<https://f-droid.org/>

⁷Gradle versions 2.3, 2.9, 2.13, 2.14.1, 3.3, 4.2.1 were used. Note that the apps that failed to build and launch correctly are not necessarily faulty, but they can, e.g., be built with other build systems or they may work on older Android versions. Investigating these issues is out of the scope of our study, so we did not follow up on the failed-to-build apps.

threw `ExceptionWithContext` stating that a register was invalid and must be between `v0` and `v255`.

5.2.2 App health after instrumentation. From all successfully instrumented Google Play apps, 10 applications crashed at launch and generated runtime exceptions, i.e., they became unhealthy after instrumentation with ACVTool (see the third row in Table 1). Five cases were due to the absence of the Retrofit annotation (four `IllegalStateException` and one `IllegalArgumentException`), 3 cases – `ExceptionInInitializerError`, 1 case – `NullPointerException`, 1 case – `RuntimeException` in a background service. In the F-Droid dataset, 2 apps became unhealthy due to the absence of Retrofit annotation, raising `IllegalArgumentException`.

Upon investigation of the issues, we suspect that they could be due to faults in the ACVTool implementation. We are working to properly identify and fix the bugs, or to identify a limitation in our instrumentation approach that leads to a fault for some type of apps.

Conclusion: we can conclude that ACVTool is able to process the vast majority of apps in our dataset, i.e., it is effective for measuring code coverage of third-party Android apps. For our total combined dataset of 1278 originally healthy apps, ACVTool has instrumented 1251, what constitutes 97.8%. From the instrumented apps, 1239 are still healthy after instrumentation. This gives us the instrumentation survival rate of 99%, and the total instrumentation success rate of 96.9% (of the originally healthy population). The instrumentation success rate of ACVTool is much better than the instrumentation rates of the closest competitors BBoxTester [80] (65%) and the tool by Huang et al. [35] (36%).

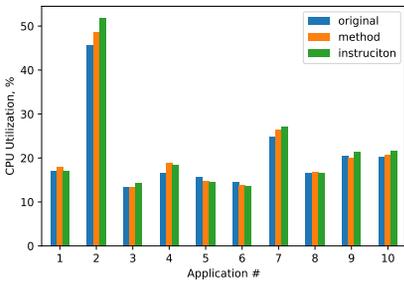
5.3 Efficiency

5.3.1 Instrumentation-time overhead. Table 1 presents the average instrumentation time required for apps from our datasets. It shows that ACVTool generally requires less time for instrumenting the F-Droid apps (on average, 27.4 seconds per app) than the Google Play apps (on average, 36.6 seconds). This difference is due to the smaller size of apps, and, in particular, the size of their `.dex` files. For our total combined dataset the average instrumentation time is 33.3 seconds per app. This time is negligible compared to the testing time usual in the black-box setting that could easily reach several hours.

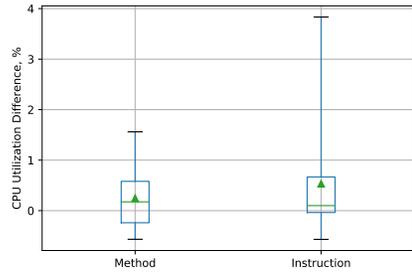
5.3.2 Runtime overhead.

CPU utilization overhead. To assess the runtime overhead induced by our instrumentation in a real world setting, we ran the original and instrumented versions of 10 apps (size range 1–32MB, 10MB mean APK size) randomly chosen from our dataset with Monkey [30], a random input event generator from Google (same seed for reproducibility, 1 second throttle, 500 events), and measured CPU utilization with the Qualcomm Snapdragon Profiler [55].

We provide performance measurement at the best precision we could achieve on Android, taking into account its reactive nature. Our fully automated pipeline works as follows. First, we reboot the Android device (we use Nexus 5) and install a new app. Then the profiler is launched starting the CPU utilization measurements. Monkey starts and exercises the app, while the profiler saves the data. Once the testing is finished, the app is uninstalled. We test every app 5 times for each of its 3 versions: original, instrumented at the method level and instrumented at the instruction level. Finally, we calculated the average CPU utilization for every app version and logic processor (since the CPU on our device has 4 logic processors).



(a) Average CPU utilization measured in 10 applications.



(b) Boxplot of the CPU utilization difference for instrumented versions of applications (instrumented versions at the method-only and at the instruction level compared to the original app versions).

Fig. 4. Performance measurement results in a realistic app testing scenario

Table 2. PassMark overhead evaluation

Granularity of instrumentation	Overhead	
	CPU	.dex size
Method	+17%	+11%
Instruction	+27%	+249%

Figure 4a shows that CPU utilization of instrumented apps slightly differs from the CPU utilization by their original versions. However, as seen from Figure 4b, the difference is insignificant: the mean difference of CPU utilization is 0.25% and 0.53% for the method- and instruction-instrumented versions, respectively. This experiment shows that the runtime overhead introduced by ACVTool is not prohibitive in a user-like application testing scenario.

PassMark overhead. To further estimate the runtime overhead we used a benchmark application called PassMark [59]. Benchmark applications are designed to assess performance of mobile devices. The PassMark app is freely available on Google Play, and it contains a number of test benchmarks related to assessing CPU and memory access performance, speed of writing to and reading from internal and external drives, graphic subsystem performance, etc. These tests do not require user interaction. The research community has previously used this app to benchmark their Android related-tools (e.g., [8]).

For our experiment, we used the PassMark app version 2.0 from September 2017. This version of the app is the latest that runs tests in the managed runtime (Dalvik and ART) rather than on a bare metal using native libraries. We have prepared two versions of the PassMark app instrumented with ACVTool: one instrumented at the method level, and another instrumented at the instruction level.

Table 2 summarizes the performance degradation of the instrumented PassMark version in comparison to the original app. When instrumented, the size of the Passmark .dex file increased from 159KB (the original version) to 178KB (method granularity instrumentation), and to 556KB (instruction granularity instrumentation). We have run the Passmark application 10 times for each level of instrumentation granularity against the original version of the app. In the CPU tests that utilize high-intensity computations, Passmark slows down,

Table 3. Increase of .dex files for the Google Play benchmark

Summary statistics	Original file size	Size of instrumented file	
		Method	Instruction
Minimum	4.9KB	17.6KB (+258%)	19.9KB (+304%)
Median	2.8MB	3.1MB (+10%)	7.7MB (+173%)
Mean	3.5MB	3.9MB (+11%)	9.0MB (+157%)
Maximum	18.8MB	20MB (+7%)	33.6MB (+78%)

on average, by 17% and 27% when instrumented at the method and instruction levels, respectively. Other subsystem benchmarks did not show significant changes in numbers.

Evaluation with PassMark is artificial for a common app testing scenario, as the PassMark app stress-tests the device. However, from this evaluation we can conclude that performance degradation under the ACVTool instrumentation is not prohibitive, especially if it is used with modern hardware.

Dex size inflation. As another metric for overhead, we analysed how much ACVTool enlarges Android apps. We measured the size of .dex files in both instrumented and original apps for the Google Play benchmark apps. As shown in Table 3, the .dex file increases on average by 157% when instrumented at the instruction level, and by 11% at the method level. Among already existing tools for code coverage measurement, InsDal [46] has introduced .dex size increase of 18.2% (on a dataset of 10 apks; average .dex size 3.6MB), when instrumenting apps for method-level coverage. Thus, ACVTool shows smaller code size inflation in comparison to the InsDal tool.

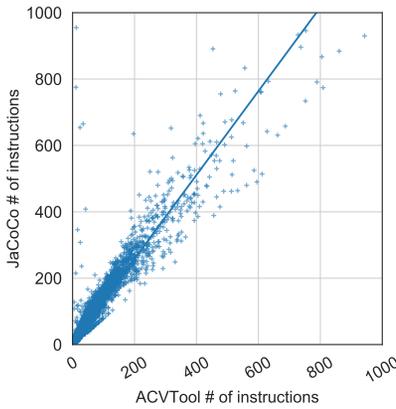
Conclusion: ACVTool introduces an off-line instrumentation overhead that is acceptable for common testing scenarios. The run-time overhead (measured as CPU utilization) in live testing with Monkey is negligible. When stress-testing with the benchmark PassMark app, ACVTool introduces 27% overhead in CPU. The increase in code base size introduced by the instrumentation instructions, while significant, is not prohibitive. Thus, we can conclude that ACVTool is efficient for measuring code coverage in Android app testing pipelines.

5.4 Compliance with other coverage tools

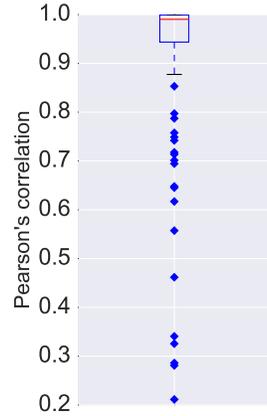
5.4.1 Instruction coverage measurement. When the source code is available, developers can log code coverage of Android apps using the JaCoCo library [37] that could be integrated into the development pipeline via the Gradle plugin. We used the coverage data reported by this library to evaluate the correctness of code coverage metrics reported by ACVTool.

For this experiment, we used only the F-Droid benchmark because it contains open-source applications. We put the new `jacocoTestReport` task in the Gradle configuration file and added our `Instrumentation` class into the app source code. In this way we avoid creating app-specific tests, and can run any automatic testing tool. Due to the diversity of project structures and versioning of Gradle, there were many faulty builds. We obtained 141 apks instrumented with JaCoCo, i.e., we could generate JaCoCo reports for them. Two of these apks were further excluded, as the JaCoCo reports for them generated incorrectly (coverage always would be zero). Thus, totally we used 139 apps in this experiment.

First, we analyze this app population in terms of instructions. Indeed, `smali` code and Java bytecode are organized differently. Figure 5a shows a scatterplot of the number of method instructions in `smali` code (measured by ACVTool, including the “untrackable” instructions) and in Java code (measured by JaCoCo). Each point in this Figure corresponds



(a) Scatterplot of the number of instructions in app methods, as computed by ACVTool and JaCoCo.



(b) Boxplot of correlation for code coverage data computed by ACVTool and JaCoCo.

Fig. 5. Compliance of coverage data reported by ACVTool and JaCoCo.

to an individual method of one of the apks in our benchmark. The line in the Figure is the linear regression line. The data shape demonstrates that the number of instructions in the `smali` code is usually slightly smaller than the number of instructions in the Java bytecode.

Figure 5a also shows that there are some outliers, i.e., methods that have low instruction numbers in `smali`, but many instructions in Java bytecode. We have manually inspected all these methods and found that outliers were constructor methods that contain declarations of arrays. `Smali` (and Dalvik VM) allocates such arrays with only one pseudo-instruction (`.array-data`), while Java bytecode is much longer [12]. Given these differences in the code organization, we can expect that, generally, there will be discrepancies in the coverage measured by ACVTool and JaCoCo.

Discrepancies in code coverage measurements can appear also due to the fact that some instructions are not tracked by ACVTool, as mentioned in Section 4. It is our choice to not count those instructions towards `covered`. In our F-Droid dataset, about half of the methods consist of 7 `smali` instructions or less. For such small methods, if they contain untraceable instructions, code coverage measurements by ACVTool and JaCoCo can differ substantially.

To compare the measured coverage data, we ran two copies of each app (instrumented with ACVTool and with JaCoCo) on the Android emulator using the same Monkey scripts for both versions. Figure 5b shows a boxplot of the correlation of code coverage measured by ACVTool and JaCoCo. Each data point corresponds to one application, and its value is the Pearson correlation coefficient between percentage of executed code, for all methods included in the app. The minimal correlation is 0.21, the first quartile is 0.94, median is 0.99, and maximal is 1.00. This means that for more than 75% of apps in the tested applications, their code coverage measurements have correlation equal to 0.94 or higher, i.e., they are strongly correlated. The boxplot in Figure 5b contains a number of outliers that appear due to the reasons explained above. Still, overall, the boxplot demonstrates that code coverage logged by ACVTool is strongly correlated with code coverage logged by JaCoCo.

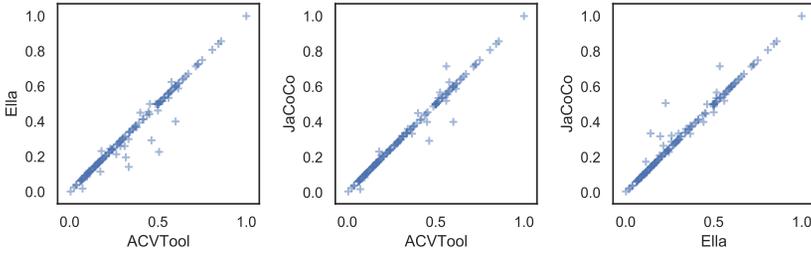


Fig. 6. Code coverage measurements at the method level: pair-wise coverage comparisons for ACVTool, ELLA and JaCoCo.

5.4.2 Method-level coverage measurements. When the application source code is not available, testers cannot use JaCoCo to measure code coverage. In this situation researchers and practitioners frequently use the ELLA library [23] to measure the method coverage [51, 67]. As ELLA is no longer maintained, ACVTool can be now used by testers to measure code coverage at the method level, if such need arises.

To provide evidence that ACVTool measures method-level code coverage reliably, we compare its results with the method coverage data reported by ELLA (no source code) and JaCoCo (white-box coverage).

For this experiment, we use the same 139 F-Droid apps mentioned above. We have instrumented them with the ACVTool at the method level. We have also instrumented them with ELLA, and we took the apps already pre-compiled with JaCoCo. For all these app versions, we run Monkey in the same setting.

Figure 6 shows scatterplots of method coverage measurements for pair-wise comparison of data from the three coverage tools; each data point corresponds to an application. This Figure demonstrates that the vast majority of data points lie on the symmetry line from (0,0) to (1,1), i.e., the tools report practically identical method coverage results for most of the apps in this set. The deviations from the main line are results of possible differences in app behavior (further elaborated in 6.3).

In this experiment, correlation of method coverage measurements for ACVTool and ELLA is 0.9829; for ACVTool and JaCoCo is 0.9912; and for ELLA and JaCoCo is 0.9858. This demonstrates very high compliance of ACVTool measurements to results obtained by the other independent tools.

Class-level compliance. As the previous experiment has shown that the method code coverage measured by ACVTool agrees with the measurements at the same level by ELLA and JaCoCo, we can consider the class-level coverage to be compliant with the other tools as well. This is an implication of our instrumentation implementation for classes: class-level coverage requires method-level instrumentation, and a class is considered covered if at least one of its methods was called.

Conclusion: overall, we can summarize that code coverage data reported by ACVTool generally agrees with data computed by JaCoCo. The discrepancies in code coverage appear due to the different approaches that the tools use, and the inherent differences in the Dalvik and Java bytecodes. At the method level, the measurements by ACVTool are highly compliant with the measurements taken by ELLA and JaCoCo.

6 USEFULNESS OF ACVTOOL IN TESTING WITH SAPIENZ

To assess the usefulness of ACVTool in practical black-box testing and analysis scenarios, we integrated ACVTool with Sapienz [51] – a state-of-art automated Android search-based testing tool. Its fitness function looks for Pareto-optimal solutions using three criteria: code coverage, number of found crashes and the length of a test suite. This experiment had two main goals: (1) demonstrate that ACVTool fits into a real automated testing/analysis pipeline; (2) evaluate whether the fine-grained code coverage measure provided by ACVTool can be useful to automatically uncover diverse types of crashes with black-box testing strategy.

Sapienz integrates three approaches to measure code coverage achieved by a test suite: EMMA [56] (reports source code statement coverage); ELLA [23] (reports method coverage); and its native plugin to measure coverage in terms of launched Android activities. EMMA does not work without the source code of apps, and thus in the black-box setting only ELLA and own Sapienz plugin could be used. The original Sapienz paper [51] did not evaluate the impact of the code coverage metric used on the discovered crashes population, because it was not possible to compare results for the same group of applications.

Our previously reported experiments with JaCoCo suggest that ACVTool can be used to replace EMMA, as the coverage data reported for Java instructions and `smali` instructions are highly correlated and comparable. Furthermore, ACVTool measures coverage in terms of classes and methods, and thus it can also replace ELLA within the Sapienz framework. Note that the code coverage measurement itself does not interfere with the search algorithms used by Sapienz. Thus, ACVTool allows us to compare the coverage granularities performance with respect to bug finding with Sapienz.

As our dataset, we use the healthy instrumented apks from the Google Play dataset described in the previous section. We have run Sapienz against each of these 799 apps, using its default parameters. Each app has been tested using the activity coverage provided by Sapienz, and the method and instruction coverage supplied by ACVTool. Furthermore, we also ran Sapienz without coverage data, i.e., substituting coverage for each test suite as 0.

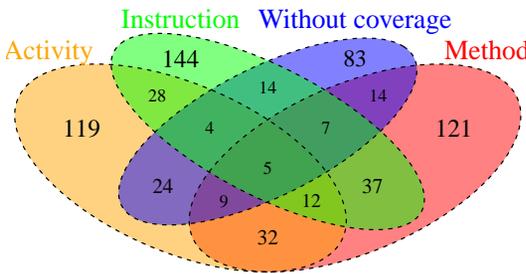
Each app has been tested by Sapienz under the default settings for 3 hours for each coverage metric. After each run, we collected the crash information (if any), which included the components of apps that crashed and Java exception stack traces. In the remainder of this section, we report on the results of crash detection with different coverage metrics and draw conclusions about whether the choice of a coverage metric contributes to bug detection.

Like many other automated testing tools for Android, Sapienz is non-deterministic. Thus, we apply statistical tests to analyze significance of all reported findings. Throughout this section, we will evaluate how effectively Sapienz finds bugs with each coverage metric in a particular app. For each app population, we obtain records of found crashes in each application, and we compare performance of each coverage metric *per each app record*. This gives us paired measurements for all coverage metrics, which are not necessarily normally distributed. Thus, to evaluate statistical significance of the results, we use the non-parametric Wilcoxon signed-rank test [69] that is appropriate in this setting. The *null-hypothesis* for the Wilcoxon test is that there is no difference which metric to use in Sapienz. *Alternative hypothesis* is that Sapienz with one coverage condition will consistently find more crashes than Sapienz with another coverage condition.

To measure the effect size we use the Vargha-Delaney A12 statistics [66] that was applied in the original Sapienz paper [51].

Table 4. Crashes found by Sapienz in 799 apps

Coverage metrics	# unique crashes	# faulty apps	# crash types
Activity coverage	233 (36%)	154	22
Method coverage	237 (36%)	142	21
Instruction coverage	251 (38%)	147	25
Without coverage	160 (24%)	102	22
Total	653	353	35



(a) Crashes found with Sapienz using different coverage metrics in 799 apps.

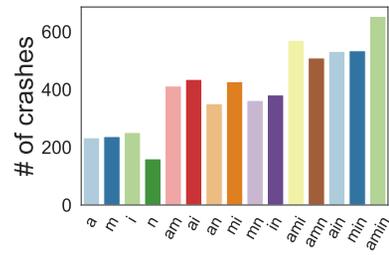
(b) Barplot of crashes found by coverage metrics individually and jointly (*a* stands for activity, *m* for method, *i* for instruction coverage, and *n* for no coverage).

Fig. 7. Crashes found by Sapienz.

6.1 Descriptive statistics of crashes

Table 4 shows the numbers of crashes grouped by coverage metrics that Sapienz has found in the 799 apps. We consider a *unique crash* as a distinctive combination of an application, its component where a crash occurred, the line of code that triggered an exception, and a specific Java exception type.

In total, Sapienz has found 353 apps out of 799 to be faulty (at least one crash detected), and it has logged 653 unique crashes with the four coverage conditions. Figure 7a summarizes the crash distribution for the coverage metrics. The intersection of the results for all code coverage conditions contains only 5 unique crashes. Individual coverage metrics have found 38% (instruction coverage), 36% (method coverage), 36% (activity coverage), and 24% (without coverage) of the total number of found crashes. These results suggest that coverage metrics at different granularities find distinct crashes.

In these experiments, instruction coverage has shown slightly better performance in bug finding as it found more crashes on the dataset. However, when comparing chances to find a bug in a particular app, its edge over the activity and method coverage is not statistically significant according to the Wilcoxon signed-rank test [69]. On the other hand, all valid coverage metrics outperform testing without coverage data in a statistically significant way (p -values $\leq 10^{-4}$). Still, Vargha-Delaney effect sizes [66] are very small: 0.52 for method and instruction coverage (compared to no coverage), and 0.53 for activity coverage. Thus, we can

conclude that it is likely that Sapienz with coverage performs better than without coverage data. However, the practical importance of coverage data used in Sapienz may be limited.

We now set out to investigate how multiple runs affect detected crashes, and whether a combination of coverage metrics could detect more crashes than a single metric.

6.2 Evaluating bug finding efficiency on multiple runs

We now look at assessing the impact of randomness on Sapienz' results. As we mentioned, our findings may be affected by the non-determinism in Sapienz. To determine the impact of coverage metrics in finding crashes *on average*, we need to investigate how crash detection behaves in multiple runs. Thus, we have performed the following two experiments on a set of 150 apks randomly selected from the 799 healthy instrumented apks.

6.2.1 Performance in 5 runs. We have run Sapienz for 5 times with each coverage metric and without coverage data, for 3 hours per each of 150 apps. This gives us two crash populations: \mathcal{P}_1 that contains unique crashes detected in the 150 apps during the first experiment, and \mathcal{P}_5 that contains unique crashes detected in the same apps running Sapienz 5 times. Table 5 summarizes the populations of crashes found by Sapienz with each of the coverage metrics and without coverage.

As expected, running Sapienz multiple times increases the amount of found crashes. In this experiment, we are interested in the proportion of crashes contributed by coverage metrics individually. If coverage metrics are interchangeable, i.e., they do not differ in capabilities of finding crashes, and they will, eventually, find the same crashes, the proportion of crashes found by individual metrics to the total crashes population can be expected to significantly increase: each metric, given more attempts, will find a larger proportion of the total crash population.

As shown in Table 5, the activity coverage has found a larger proportion of total crash population (38% from 32%). Sapienz without coverage data also shows better performance over multiple runs (36% from 31%), while the instruction coverage has increased performance from 36% to 40%. The method coverage has achieved the best improvement (45% from 34%). For all coverage metrics, the increases in the found crashes populations due to repeated testing are statistically significant according to the Wilcoxon signed-rank test (p -values $\leq 10^{-5}$), but the Vargha-Delaney effect sizes [66] are small: all in the range (0.61, 0.64). Thus, repeating Sapienz test executions improves chances to find a crash in an app, but not a lot. The edge of method coverage over other metrics in repeated experiments is not statistically significant.

These findings suggest that even with 5 repetitions a single coverage metric is not able to find all crashes that were detected by other metrics. Our results in this experiment are consistent with a previously reported smaller experiment that involved only 100 apps (see [22] for more details).

6.2.2 Evaluating a combination of metrics. The previous experiment indicates that even repeating the runs multiple times does not allow any of the code coverage metrics to find the same number of bugs as all metrics together. We now fix the time that Sapienz spends on each apk⁸, and we want to establish whether the number of crashes that Sapienz can find in an apk with 3 metrics is greater than the number of crashes found with just one metric but with 3 attempts. This would suggest that the combination of 3 metrics is more effective

⁸In these testing scenarios, Sapienz spends the same amount of time per app (3 runs), but the coverage conditions are different.

Table 5. Crashes found in 150 apps with 1 and 5 runs.

Coverage metrics	Crashes	
	\mathcal{P}_1 : 1 run	\mathcal{P}_5 : 5 runs
Activity coverage	40 (32%)	101 (38%)
Method coverage	43 (34%)	119 (45%)
Instruction coverage	46 (36%)	106 (40%)
No coverage	39 (31%)	95 (36%)
Total	126	263

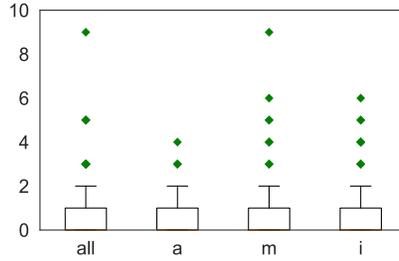
Fig. 8. Boxplots of crashes detected per app (*a* stands for activity, *m* for method, and *i* for instruction, respectively).

Table 6. Summary statistics for crashes found per apk, in 150 apk

Statistics	1 run \times 3 metrics	3 runs \times 1 metric		
		activity	method	instruction
Min	0	0	0	0
1st. Quartile	0	0	0	0
Mean	0.65	0.48	0.64	0.58
Median	0	0	0	0
3rd. Quartile	1	1	1	1
Max	9	4	9	6

in finding crashes than each individual metric. For each apk from the chosen 150 apps, we compute the number of crashes detected by Sapienz with each of the three coverage metrics executed once. We then have executed Sapienz 3 times against each apk with each coverage metric individually.

Table 6 summarizes the basic statistics for the apk crash numbers data, and the data shapes are shown as boxplots in Figure 8. The summary statistics show that Sapienz equipped with 3 coverage metrics has found, on average, slightly more crashes per apk than Sapienz using only one metric but executed 3 times. To verify this, we apply the Wilcoxon signed-rank test [69].

The results of the Wilcoxon test did not reject the null-hypotheses for all coverage metrics (p -values 0.43 and 0.58, and 0.51 for activity, method and instruction coverage, respectively). This can be interpreted as a high probability that the crashes data have been drawn from similarly distributed populations.

To confirm this negative result, we apply the Vargha-Delaney A12 statistic to measure the effect size. The A12 effect sizes for differences in the crash population found by 3 metrics

jointly and the population found by the activity, method and instruction coverage are, respectively, 0.515, 0.516 and 0.513, which all correspond to a negligible effect.

Our findings from this experiment are not fully consistent with the previously reported experiment on a smaller set of 100 apps [22]. The difference could be explained by the following factors. First, we have used only healthy instrumented apps in this experiment (the ones that did not crash upon installation). The experiment reported in [22] did not involve the check for healthiness, and the crashing apps could have affected the picture. In the unhealthy app case, Sapienz always reports one single crash for it, irrespectively of which coverage metrics is used. Note that in our Google Play sample approximately 17% are unhealthy, i.e., they cannot be executed on an emulator, as required by Sapienz. Second, the new apps tested in this experiment could have behaved slightly differently than the previously tested cohort. And, finally, in these experiments we used more recent releases of the testing environment components, including the Android SDK, that are more stable and have less compatibility issues.

6.3 Impact of ACVTool on Sapienz

Instrumentation and repackaging of the app's codebase may introduce differences in runtime behavior and additional faults. Such deviations can make parts of the app unreachable, which may impact further testing. Despite our positive evaluation demonstrated in Section 5, automated testing tools, such as Sapienz, look deeper into the app and can be more significantly impacted by issues raised by instrumentation. Here we analyze how much does ACVTool interfere with the Sapienz testing process. We consider two main aspects.

- *Preserving the original behavior.* We compare the behavior of instrumented apps against their original versions and report the differences.
- *Fault analysis.* We analyse what crashes Sapienz found with and without ACVTool and report on ACVTool-specific crashes.

6.3.1 Preserving behavior. To evaluate the ACVTool impact on app behavior we designed the following experiment. For every app from the 150 apps subset mentioned in Section 6.2 we took the most evolutionary developed Sapienz test suite and ran it on two versions of the app: original and instrumented at instruction level. After every triggered event we saved a screenshot of the UI state and its XML layout (with the help of UIAutomator [31]). Then we removed the status bar (around 90px at the image top) from every image and performed an automated comparison of images and XML files for the original and instrumented versions in Beyond Compare [58].

The publicly available version of Sapienz produces sequences containing mostly atomic Monkey [30] events, with one exception. An event named `GUIGen` in the sequence produces up to 12 random events on Android. Thus, in this experiment we excluded the `GUIGen` line from all the sequences to achieve sequence reproducibility. Moreover, we kept 1 second pause between the events to make sure that content loading and app animation have lower impact on the produced screenshots.

In this experiment, 50% out of the total 33938 automatically compared image and XML pairs were found to be identical. We manually inspected the other pairs and found that the differences could be attributed to the following main reasons.

- **Pop-ups:** One of the apps in the pair in some cases fires a pop-up related to the Android OS state or the app itself. This happens to both instrumented and original apps. In this case we re-run the test and it solves the problem.

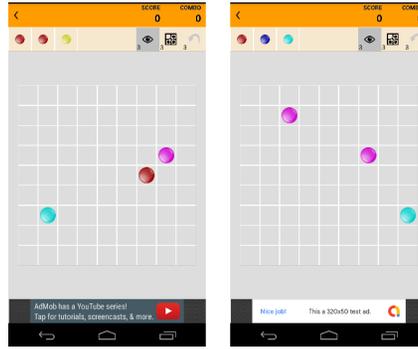


Fig. 9. Example of a justified difference in behavior between the original (left) and the instrumented (right) app versions: the screenshots are different due to the dynamic nature of the app itself and the loaded ad content.

- Advertisement: Apps frequently load ads, which may look differently each time, as shown in Figure 9).
- Dynamic content: Some apps may display their UI each time differently or download completely new content, as exemplified in Figure 9.
- PNG artefacts: Apktool sometimes breaks PNG files or changes the color and transparency properties during decoding. Therefore, parts of the app may look different.

In this experiment, we assume that two versions of an app behave identically if their GUI states stay equal.

Taking into account the above-mentioned factors, we consider the app behavior unmodified if the GUI states are totally identical, or they are different due to the four reasons specified above, which we call *justifiable discrepancies*. Such discrepancies do not correspond to functional differences in app states.

In total, 145 out of 150 apps in the dataset behaved justifiably the same, while 26 of them behaved completely the same. In these 145 apps we did not observe behavioral differences caused by ACVTool.

Two apps behaved differently because the test interacted with an ad, which expanded to the full screen mode. Three apps could not load maps, which made the apps to malfunction. They threw the *Google Maps Android API: Authorization failure* error in `logcat`. ACVTool caused this error because it re-signs the app with its own signature, while the Google Maps API requires the original signature.

6.3.2 Fault analysis. Figure 10 demonstrates the distribution of crashes found by Sapienz with respect to the code coverage metric applied. The 653 unique crashes found on the set of 799 apps were caused by 35 exception types. We note that our crash type distribution resembles the results reported in the original Sapienz paper for the main crash types on the Google Play subjects [51]. The most prevalent Java exception types in Figure 10 also generally agree with the statistics of Java exceptions in open source Android projects reported by [20]. As Android bug finding is not the core goal of this paper, we limit ourselves to reporting the general crash distribution as provided by Sapienz, and we do not focus here on attributing the root causes of crashes as in, e.g., [44].

It is interesting yet very challenging to evaluate whether ACVTool introduces new app crashes due to the instrumentation process. There are two approaches to confirm if ACVTool

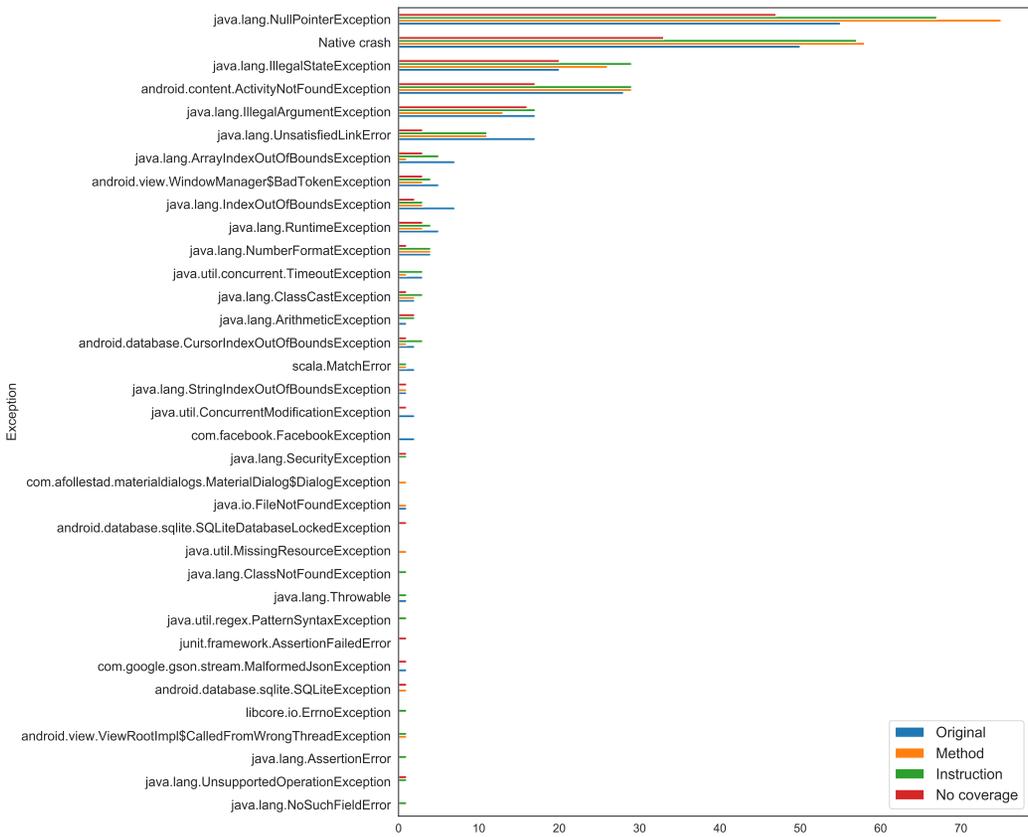


Fig. 10. Distribution of the crash types (exceptions) found by Sapienz with different code coverage granularities. Exceptions are sorted in the descending order with respect to the total number of unique found crashes with this exception type.

introduces new faults in the experiment with Sapienz. First, from a crash description and the corresponding stack trace we can find the evidence that the bug is introduced by ACVTool. Second, crashes introduced by ACVTool and thus detected only on instrumented apps should not reproduce on the original app versions.

As we described in Section 3.1, during the offline phase ACVTool embeds the custom **Instrumentation** class and probes into apps. We carefully analyzed the stack traces of all exceptions obtained in our experiment with Sapienz and did not find any evidence of ACVTool methods there. However, the probes we embed do not use method calls, but rather directly change binary array values corresponding to original bytecode instructions at runtime. Thus, if any fault happens due to these probes we will not see any probe-specific symptoms in the stack trace. Therefore, the first approach cannot completely confirm the absence of crashes introduced by ACVTool.

The applicability of the second approach is hurdled by flaky tests. Flaky tests are those that do not reliably fail even in identical circumstances. The authors of the follow-up paper about Sapienz [3] admit that the tool is subject to flaky tests, but they do not provide estimates about how many tests are flaky. It is mentioned only that “it is safer to assume

that we live in a world where all tests are flaky”, what may indicate that the flaky tests proportion is high.

To prove this we ran an experiment where we used the crash-leading test suites found by Sapienz with the default settings on the original, non-modified apps. In this experiment, only 37% of the faults found in the original apps were reproduced on the same apps.

The main reason for this low reproducibility of faults is the asynchronous nature of Android [3]. Depending on the wait time, an asynchronous call to a service may produce different results. When the service returns a value in time, this value is used, but if a value is not returned, the default value is used. This may lead to completely different execution paths.

We should also mention that the default throttle setting that Sapienz uses for Monkey is quite aggressive. It intensively bombards an app with events irrespective of the app’s state and its animation. Since Monkey’s throttle parameter significantly affects crash detection [53], satisfactory crash reproducibility on Sapienz may be achieved with a proper throttle value (e.g., as we set in Section 6.3.1). However, the consequence and a huge disadvantage would be a dramatic slowing down of Sapienz in finding new faults. Still, even this approach cannot guarantee full reproducibility of the crashes. Thus, with this approach we cannot confirm that ACVTool does not introduce new faults, because these faults could be due to flaky tests.

Thus, we can confidently confirm only the crashes described in Section 5.2.2 as caused by the ACVTool instrumentation phase. However, out of the 5 exception types found when filtering healthy apps in Section 5.2.2, 3 types – `IllegalStateException`, `IllegalArgumentException`, `NullPointerException` – appear prevalently in the found crashes distribution. We expect that ACVTool could have contributed to at least some of these exceptions.

6.4 Analysis of results

Our experiments show that ACVTool can be integrated into an automated testing pipeline and it can be used in conjunction with available testing tools such as Sapienz. Our experiments demonstrate that ACVTool does not impact app behavior in testing with Sapienz for the majority of the tested apps. However, as we expected, the repackaging process breaks the original signature, and some app code parts may become unavailable due to the failing signature checks, as happens, e.g., with the Google Maps Android API.

We can also conclude that better investigation and integration of different coverage granularities is warranted in the automated Android testing domain, just like in software testing in general [16]. Our crash data analysis and the experiment with repeating executions 5 times show that no coverage metric is able to find the vast majority of the total found crash population. Sapienz without coverage finds fewer bugs than with coverage data (160 crashes on the total app population versus, e.g., 233 crashes found with the activity coverage), yet it is still able to uncover a significant crash population. Further investigation of these aspects could be a promising line of research. Our open-source ACVTool can be helpful in these studies.

7 DISCUSSION

ACVTool addresses the important problem of measuring code coverage of closed-source Android apps. Our experiments show that the proposed instrumentation approach works for the majority of Android apps, the measured code coverage is reliable, and the tool can be integrated with security analysis and testing tools. We have already shown that integration of the coverage feed produced by our tool into an automated testing framework can help to

uncover more application faults. Our tool can further be used, for example, to compare code coverage achieved by dynamic analysis tools and to find suspicious code regions.

In this section, we discuss limitations of the tool design and current implementation, and summarize the directions in which the tool can be further enhanced. We also review threats to validity regarding the conclusions we make from the Sapienz experiments.

7.1 Limitations of ACVTool

ACVTool design and implementation have several limitations that we discuss in this section.

Limitations of the ACVTool design. An inherent limitation of our approach is that apps must be first instrumented before their code coverage can be measured. Indeed, in our experiments, there was a fraction of apps that could not be repackaged and instrumented. Furthermore, apps can employ various means to prevent repackaging, e.g., they can check their signature at the start, and stop executing in case of a failed signature check. Moreover, as shown by the experiment reported in Section 6.3, the repackaging step may inhibit the usage of Google APIs. Still, this limitation is common to all tools that instrument applications (e.g., [23, 35, 46, 74, 80]). Considering this, ACVTool has successfully instrumented 96.9% of our total original dataset selected randomly from F-Droid and Google Play. Our instrumentation success rates are significantly higher than any of the related work, where this aspect has been reported (e.g., [35, 80]). Therefore, ACVTool is practical and reliable. We examine the related work and compare ACVTool to the available tools in the subsequent Section 8.

While being an important part of the ACVTool workflow, the decompilation and repackaging part are not the focus of this study. Therefore, we do not investigate possible errors in `apktool`, which is currently the best Android reverse engineering tool that integrates a decompiler to `smali`. It is also well-maintained, and new improved versions are released regularly.

We assessed that the code coverage data from ACVTool is compliant to the measurements from the well-known JaCoCo [37] and ELLA [23] tools. We have found that, even though there could be slight discrepancies in the number of instructions measured by JaCoCo and ACVTool, the coverage data obtained by both tools is highly correlated and commensurable. Therefore, the fact that ACVTool does not require the source code makes it, in contrast to JaCoCo, a very promising tool for simplifying the work of Android developers, testers, and security specialists.

Limitations of our instrumentation approach. One of the reasons for the slight difference in the JaCoCo and ACVTool measurements of the number of instructions is the fact that we do not track several instructions, as specified in Section 4. Technically, nothing precludes us from adding probes right before the “untraceable” instructions. However, we consider this solution to be inconsistent from the methodological perspective, because we deem the right place for a probe to be immediately after the executed instruction. In the future we plan to extend our approach to compute also basic block coverage, and then the “untraceable” instruction aspect will be fully and consistently eliminated. Alternatively, ACVTool can be enhanced by introducing a lightweight static analysis at the `smali` code level for a control flow graph-aware instrumentation [34].

Another limitation of our current approach is the constraint of 256 registers per method. Our instrumentation approach introduces 3 new registers. This register manipulation technique is safe as long as the total number of registers in the original `smali` method is less than or equal to 256. The only problematic instruction in this respect is `aput-boolean`, which

can access up to 256 registers. While this limitation could potentially affect the success rate of ACVTool, we have encountered only one app, in which this limit was exceeded after the instrumentation. This limitation can be addressed either by switching to another instrumentation approach, whereby inserting probes as specific method calls, or by splitting big methods. Both of the approaches may require to reassemble an app that has more than 64K methods into a multidex apk [26]. We plan this extension as future work.

Taken to extremes, insertion of probes may potentially lead to issues. It is not clear what is the limit to the amount of instructions in a single app method and whether this limit can be reached by increasing the total number of instructions by a factor of 4. We have not encountered such cases, but this aspect may be worthy of further investigation in case of testing very complex applications [77].

We investigated the runtime overhead introduced due to our instrumentation, which could be another potential limitation. Our results show that ACVTool does not introduce a prohibitive runtime overhead. For example, the very resource-intensive computations performed by the PassMark benchmark app degrade the CPU utilization by 27% in the instruction-level instrumented version. This is a critical scenario, and the overhead for an average app will be much smaller, which is confirmed by our experiments on real apps.

Limitations of the current ACVTool implementation. Our current ACVTool prototype does not fully support multidex apps. It is possible to improve the prototype by adding full support for multidex files, as the instrumentation approach itself is extensible to multiple dex files. In our dataset, we have 46 multidex apps, which constitutes 3.5% of the total population. In particular, in the Google Play benchmark there were 35 apks with 2 dex files, and 9 apks containing from 3 to 9 dex files (overall, 44 multidex apps). In the F-Droid benchmark, there were two multidex apps that contained 2 and 42 dex files, respectively. The current version of ACVTool is able to instrument multidex apks and log coverage data for them, but coverage will be reported only for one dex file. While we considered the multidex apks, if instrumented correctly, as a success for ACVTool, after excluding them, the total instrumentation success rate will become 93.1%, which is still much higher than other tools.

Also, the current implementation still has a few issues (3.3% of apps have not survived instrumentation), which we plan to fix in subsequent releases.

7.2 Threats to validity

Our experiments with Sapienz reported in Section 6 allow us to conclude that black-box code coverage measurement provided by ACVTool is useful for state-of-art automated testing frameworks. Furthermore, these experiments suggest that it is necessary to better study the impact of coverage data for achieving time-efficient and effective bug finding.

At this point, it is not yet clear if there is a coverage metric that works best. Further investigation of this topic is required to better understand exactly how granularity of code coverage affects the results, and what are other confounding factors that may influence the performance of Sapienz and other similar tools.

Our findings from these experiments are negative, as our data does not indicate prevalence of a particular coverage granularity. We now discuss the threats to validity for the conclusions we draw from our experiments. These threats to validity could potentially be eliminated by a larger-scale experiment.

Internal validity. Threats to internal validity concern the experiment's aspects that may affect validity of the findings. First, our preliminary experiment involved only a sample of 799 Android apps. It is, in theory, possible that on a larger dataset we will obtain different results

in terms of number of unique crashes and their types. A significantly larger experiment involving thousands of apps could lead to more robust results.

Second, Sapienz relies on the random input event generator Monkey [30] as the underlying test engine, and thus it is nondeterministic. It is possible that this randomness may have influence on our current results, and the results obtained in another experiment will show a clear edge for some coverage granularity.

Third, we perform our experiment using the default parameters of Sapienz. It is possible that their values, e.g., the length of a test sequence, may also have an impact on the results. In our future work, we plan to investigate this threat further.

We acknowledge that tools measuring code coverage may introduce some additional bugs during the instrumentation process. In our experiments, results for the method and instruction-level coverage have been collected from app versions instrumented with ACVTool, while data for the activity coverage and without coverage were gathered for the original apk versions. If ACVTool introduces bugs during instrumentation, this difference may explain why the corresponding populations of crashes for instrumented (method and instruction coverage) and original (activity coverage and no coverage) apps tend to be close.

As reported in Section 6.3, we have tried to address this threat by comparing application behaviors on original and instrumented app versions, and by investigating the crashes. We have shown that ACVTool does not change the app behavior, as visible in the GUI. However, we are yet not able to automatically confirm that ACVTool does not introduce crashes at runtime. Unfortunately, the publicly available Sapienz version does not support crash reproducibility. In the future, we consider to systematically evaluate reproducibility of found crashes across the original and instrumented app versions using tools like RecDroid [78], CrashScope [52] or Paladin [48].

Finally, our findings may be affected by the experimental set-up. We run Sapienz with Android emulators, which are not fully representative of real devices and may introduce some stability issues that can result in app crashes [3].

External validity. Threats to external validity concern the generalization of our findings. To test the viability of our hypothesis, we have experimented with only one automated test design tool. It could be possible that other similar tools that rely upon code coverage metrics such as Stoa [62], AimDroid [32] or QBE [39] would not obtain better results when using the fine-grained instruction-level coverage. We plan to investigate this further by extending our experiments to include more automated testing tools that rely on code coverage.

It should be also stressed that we used apps from the Google Play for our experiment. While preparing a delivery of an app to this market, developers usually apply different post-processing tools, e.g., obfuscators and packers, to prevent potential reverse-engineering. Some crashes in our experiment may be introduced by these tools. In addition, obfuscators may introduce some additional dead code and alter the control flow of apps. These features may also impact the code coverage measurement, especially in case of more fine-grained metrics. Therefore, in our future work we plan to also investigate this issue.

8 RELATED WORK

8.1 Android app testing

Automated testing of Android applications is a very prolific research area. Today, there are many frameworks that combine UI and system events generation, striving to achieve better code coverage and fault detection. E.g., Dynodroid [49] is able to randomly generate both UI and system events. Interaction with the system components via callbacks is another

facet, which is addressed by, e.g., EHBdroid [60]. The survey by Choudhary et al. [18] has compared the most prominent testing tools that automatically generate app input events in terms of efficiency, including code coverage and fault detection. Three recent surveys, by Zein, Salleh and Grundy [76], by Linares-Vásquez et al. [45], and by Kong et al. [38], summarize the main efforts and challenges in the automated Android app testing area.

8.2 Coverage measurement tools in Android

White-box coverage measurement. Tools for white-box code coverage measurement are included into the Android SDK maintained by Google [29]. Supported coverage libraries are JaCoCo [37], EMMA [56], and the IntelliJ IDEA coverage tracker [61]. These tools are capable of measuring fine-grained code coverage, but require that the source code of an app is available. This makes them suitable only for testing apps at the development stage.

Table 7. Summary of black-box coverage measuring tools

Tool	Tool details			Results of empirical evaluation					
	Coverage granularity	Target representation	Code available	Sample size	Instrumentation success rate (%)		Overhead		Compliance evaluated
					Instru-mented	Executed	Instr. time (sec/app)	Run time (%)	
ELLA [23, 67]	method	Dalvik bytecode	Y	68 [67]; 1278 (this paper)	60% [67]; 95.9% (this paper)	60% [67]; 91.1% (this paper)	15.7 (this paper)	N/A	Y (this paper)
Huang et al. [35]	class, method, basic block, instruction	Dalvik bytecode (smali)	N	90	36%	N/A	N/A	N/A	Y
BBoxTester [80]	class, method, basic block	Java bytecode	Y	91	65%	N/A	15.5	N/A	N
Asc [60]	basic block, instruction	Jimple	Y	35	N/A	N/A	N/A	N/A	N
ABCA [36]	class, method, instruction	Jimple	N	6	N/A	N/A	N/A	9-86% of system time	N
Horvath et al. [33]	method	Java bytecode	N	10	N/A	N/A	N/A	N/A	N
InsDal [46, 47, 71]	class, method	Dalvik bytecode	N	10	N/A	N/A	1.5	N/A	N
Sapienz [51]	activity	Dalvik bytecode	Y	1112	N/A	N/A	N/A	N/A	N
DroidFax [13?, 14]	instruction	Jimple	Y	195	N/A	N/A	N/A	N/A	N
AndroCov [11, 42]	method, instruction	Jimple	Y	17	N/A	N/A	N/A	N/A	N
CovDroid [74]	method	Dalvik bytecode (smali)	N	1	N/A	N/A	N/A	N/A	N
ACVTool (this paper)	class, method, instruction	Dalvik bytecode (smali)	Y	1278	97.8%	96.9%	33.3	up to 27% on Pass-Mark	Y

Black-box coverage measurement. Several frameworks for measuring black-box code coverage of Android apps already exist, however they are inferior to ACVTool. Notably, these frameworks often measure code coverage at coarser granularity. For example, ELLA [23], InsDal [46], CovDroid [74], and the tool by Horvath et al. [33] measure code coverage only at the method level.

ELLA [23] is arguably one of the most popular tools to measure Android code coverage in the black-box setting, however, it is no longer supported. ELLA relies on the same approach to app instrumentation as ACVTool (at the method level): it inserts probes at the beginning of methods, manipulates registers and tracks probe execution [23]. The difference in coverage measurement approaches appears in the reporting procedure. While executed, an app instrumented by ELLA sends identifiers of executed methods via a socket to the ELLA server.

An empirical study by Wang et al. [67] has evaluated performance of Monkey [30], Sapienz [51], Stoa [62], and WCTestter [77] automated testing tools on large and popular industry-scale apps, such as Facebook, Instagram and Google. They have used ELLA to measure method code coverage, and they reported the total success rate of ELLA at 60% (41 apps) on their sample of 68 apps.

In our own experiment with ELLA reported in Section 5.4.2, ELLA has nearly the same instrumentation success rates as ACVTool: the same 98.7% apps in the F-Droid dataset, and 94.4% in Google Play dataset (against 97.8% for ACVTool). After the ELLA instrumentation, in total, 91.1% out of 1278 apps are healthy (against 96.9% for ACVTool). While success rates are similar between the tools (ACVTool performs slightly better), ACVTool does more sophisticated instrumentation at the instruction level and, therefore, takes twice as much time as compared to ELLA.

Huang et al. [35] proposed an approach to measure code coverage for dynamic analysis tools for Android apps. Their high-level approach is similar to ours: an app is decompiled into `smali` files, and these files are instrumented by placing probes at every class, method and basic block to track their execution. However, the authors report a low instrumentation success rate of 36%, and only 90 apps have been used for evaluation. Unfortunately, the tool is not publicly available, and we were unable to obtain it or the dataset by contacting the authors. Because of this, we cannot compare its performance with ACVTool, although we report a much higher instrumentation rate, evaluated against a much larger dataset.

BBoxTester [80] is another tool for measuring black-box code coverage. Its workflow includes app disassembling with `apktool` and decompilation of the `dex` files into Java `jar` files using `dex2jar` [1]. The `jar` files are instrumented using EMMA [56], and assembled back into an apk. The empirical evaluation of BBoxTester showed the successful repackaging rate of 65%, and the instrumentation time has been reported to be 15 seconds per app. We were able to obtain the original BBoxTester dataset. Out of 91 apps, ACVTool failed to instrument just one. This error was not due to our own instrumentation code: `apktool` could not repackage this app. Therefore, ACVTool successfully instrumented 99% of this dataset, against 65% of BBoxTester.

The InsDal tool [46] instruments apps for class and method-level coverage logging by inserting probes in the `smali` code, and its workflow is similar to ACVTool. The tool has been applied for measuring code coverage in the black-box setting with the AppTag tool [71], and for logging the number of method invocations in measuring the energy consumption of apps [47]. The information about instrumentation success rate is not available for InsDal, and it has been evaluated on a limited dataset of 10 apps. The authors have reported an average instrumentation time overhead of 1.5 sec per app, and an average instrumentation code overhead of 18.2% of `dex` file size. ACVTool introduces a smaller code size overhead of 11%, on average, but requires more time to instrument an app. On our dataset, the average instrumentation time is 24.1 seconds per app, when instrumenting at the method level only. It is worth noting that half of this time is spent on repackaging with `apktool`.

CovDroid [74], another black-box code coverage measurement system for Android apps, transforms apk code into `smali`-representation using the `smali` disassembler [10] and inserts probes at the method level. The coverage data is collected using an execution monitor, and the tool is able to collect timestamps for executed methods. While the instrumentation approach of ACVTool is similar in nature to that of CovDroid, the latter tool has been evaluated on a single application only.

Alternative approaches to Dalvik instrumentation focus on performing detours via other languages, e.g., Java or Jimple. For example, Bartel et al. [9] worked on instrumenting Android apps for improving their privacy and security via translation to Java bytecode. Zhauniarovich et al. [80] translated Dalvik into Java bytecode in order to use EMMA’s code coverage measurement functionality, while Horvath et al. [33] used translation into Java bytecode to use their own `JInstrumenter` library for `jar` files instrumentation. The limitation of such approaches, as reported in [80], is that not all apps can be retargeted into Java bytecode.

The instrumentation of apps translated into the Jimple representation has been used in, e.g., Asc [60], DroidFax [13], ABCA [36], and AndroCov [11, 42]. Jimple is a suitable representation for subsequent analysis with Soot [6], yet, unlike `smali`, it does not belong to the “core” Android technologies maintained by Google. Moreover, Arnatovich et al. [5] in their comparison of different intermediate representations for Dalvik bytecode advocate that `smali` is the most accurate alternative to the original Java source code and therefore is the most suitable for security testing.

Remarkably, in the absence of reliable fine-grained code coverage reporting tools, some frameworks [13, 15, 40, 43, 48, 51, 60] integrate their own black-box coverage measurement libraries. Many of these papers do note that they have to design their own code coverage measurement means in the absence of a reliable tool. ACVTool addresses this need of the community. As the coverage measurement is not the core contribution of these works, the authors have not provided enough information about the rates of successful instrumentation, and other details related to the performance of these libraries, so we are not able to compare them with ACVTool.

App instrumentation. Among the Android application instrumentation approaches, the most relevant for us are the techniques discussed by Huang et al. [35], InsDal [46] and CovDroid [74]. ACVTool shows much better instrumentation success rate, because our instrumentation approach deals with many peculiarities of the Dalvik bytecode. A similar instrumentation approach has been also used in the DroidLogger [21] and SwiftHand [17] frameworks, which do not report their instrumentation success rates.

Summary. Table 7 summarizes the performance of ACVTool and code coverage granularities that it supports in comparison to other state-of-the-art tools. ACVTool significantly outperforms any other tool that measures black-box code coverage of Android apps. Our tool has been extensively tested with real-life applications, and it has excellent instrumentation success rate, in contrast to other tools, e.g., [35] and [80]. We attribute the reliable performance of ACVTool to the very detailed investigation of `smali` instructions we have done, that is missing in the literature. ACVTool is available as open-source to share our insights with the community, and to replace the outdated tools (ELLA [23] and BBoxTester[80]) or publicly unavailable tools ([35, 74]).

9 CONCLUSIONS

In this paper, we presented an instrumentation technique for Android apps. We incorporated this technique into ACVTool – an effective and efficient tool for measuring precise code coverage of Android apps. We were able to instrument and execute 96.9% out of 1278 apps used for the evaluation, showing that ACVTool is practical and reliable.

The empirical evaluation that we have performed allows us to conclude that ACVTool will be useful for both researchers who are building testing, program analysis, and security assessment tools for Android, and practitioners in industry who need reliable and accurate coverage information.

To enable better support for automated testing community, we are working to add support for multidex apps, extend the set of available coverage metrics to branch coverage, and to alleviate the limitation caused by the fixed amount of registers in a method. We will also investigate an option to store counters for each executed instruction, what will allow identifying most and least executed code locations. As another promising line of future work, we will investigate on-the-fly dex file instrumentation that will make ACVTool even more useful in the context of analyzing highly complex applications and malware.

Furthermore, our experiments with Sapienz have produced interesting conclusions that no coverage granularity is able to find all crashes, even in repeated experiments. We have also found negative results on the importance of coverage granularity, when used as a component of the fitness function in the black-box app testing. ACVTool that works with most of the apps has uniquely enabled us to perform this coverage comparison study. The second line of future work for us is to expand our experiments to more apps and more testing tools, thus establishing better guidelines on which coverage metric(s) is more effective and efficient in bug finding.

Acknowledgements

This work was supported by the Luxembourg National Research Fund (FNR) through grants AFR-PhD-11289380-DroidMod and C15/IS/10404933/COMMA. We thank the anonymous reviewers for their useful comments.

REFERENCES

- [1] 2017. *dex2jar*. <https://github.com/pxb1988/dex2jar>
- [2] K. Allix, T. F. Bissyande, J. Klein, and Y. L. Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 468–471. <https://doi.org/10.1109/MSR.2016.056>
- [3] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *International Symposium on Search Based Software Engineering*. Springer, 3–45.
- [4] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2 ed.). Cambridge University Press. <https://doi.org/10.1017/9781316771273>
- [5] Yauhen Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaiping Liu, and Lwin Khin Shar. 2014. Empirical Comparison of Intermediate Representations for Android Applications.. In *SEKE*. 205–210.
- [6] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2017. The Soot-based Toolchain for Analyzing Android Apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems* (Buenos Aires, Argentina) (*MOBILESoft '17*). IEEE Press, Piscataway, NJ, USA, 13–24. <https://doi.org/10.1109/MOBILESoft.2017.2>
- [7] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) (*OOPSLA '13*). ACM, New York, NY, USA, 641–660. <https://doi.org/10.1145/2509136.2509549>

- [8] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. 2017. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 481–495.
- [9] Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, and Yves Le Traon. 2012. In-Vivo Bytecode Instrumentation for Improving Privacy on Android Smartphones in Uncertain Environments. arXiv:1208.4536 [cs.CR]
- [10] Ben Gruver. 2018. *Smali/Baksmali Tool*. Retrieved 23/11/2018 from <https://github.com/JesusFreke/smali>
- [11] Nataniel P. Borges, Jr., Maria Gómez, and Andreas Zeller. 2018. Guiding App Testing with Mined Interaction Models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (Gothenburg, Sweden) (MOBILESoft '18)*. ACM, New York, NY, USA, 133–143. <https://doi.org/10.1145/3197231.3197243>
- [12] D. Bornstein. 2008. *Google I/O 2008 - Dalvik Virtual Machine Internals*. Retrieved 24/01/2018 from <https://sites.google.com/site/io/dalvik-vm-internals>
- [13] H. Cai and B. G. Ryder. 2017. DroidFax: A Toolkit for Systematic Characterization of Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 643–647. <https://doi.org/10.1109/ICSME.2017.35>
- [14] H. Cai and B. G. Ryder. 2017. Understanding Android Application Programming and Security: A Dynamic Study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 364–375. <https://doi.org/10.1109/ICSME.2017.31>
- [15] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2017. Curious-Droid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Financial Cryptography and Data Security*, Jens Grossklags and Bart Preneel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 231–249.
- [16] Thierry Titchou Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *Proceedings of the 39th International Conference on Software Engineering (Buenos Aires, Argentina) (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 597–608. <https://doi.org/10.1109/ICSE.2017.61>
- [17] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.
- [18] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.
- [19] Mike Cleron. 2017. *Android Announces Support for Kotlin*. Retrieved 23/11/2017 from <https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html>
- [20] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. 2017. Exception handling bug hazards in Android. *Empirical Software Engineering* 22, 3 (2017), 1264–1304.
- [21] Shuaifu Dai, Tao Wei, and Wei Zou. 2012. DroidLogger: Reveal suspicious behavior of Android applications via instrumentation. In *2012 7th International Conference on Computing and Convergence Technology (ICCT)*. 550–555.
- [22] Stanislav Dashevskiy, Olga Gadyatskaya, Aleksandr Pilgun, and Yury Zhauniarovich. 2018. The Influence of Code Coverage Metrics on Automated Testing Efficiency in Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. ACM, New York, NY, USA, 2216–2218. <https://doi.org/10.1145/3243734.3278524>
- [23] ELLA. 2016. *A Tool for Binary Instrumentation of Android Apps*. <https://github.com/saswatanand/ella>
- [24] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Trans. Softw. Eng. Methodol.* 24, 4, Article 22 (Sept. 2015), 33 pages. <https://doi.org/10.1145/2660767>
- [25] Google. 2017. *Dalvik Executable format*. Retrieved 23/11/2017 from <https://source.android.com/devices/tech/dalvik/dex-format>
- [26] Google. 2017. *Enable Multidex for Apps with Over 64K Methods*. Retrieved 23/11/2017 from <https://developer.android.com/studio/build/multidex.html>
- [27] Google. 2018. *Dalvik bytecode*. <https://source.android.com/devices/tech/dalvik/dalvik-bytecode>
- [28] Google. 2018. *smali*. Retrieved 02/12/2018 from <https://android.googlesource.com/platform/external/smali/>
- [29] Google. 2018. *Test your app*. <https://developer.android.com/studio/test/index.html>

- [30] Google. 2018. *UI/Application Exerciser Monkey*. Retrieved 23/11/2018 from <https://developer.android.com/studio/test/monkey>
- [31] Inc. Google. 2019. *UI Automator*. Retrieved 23/09/2019 from <https://developer.android.com/training/testing/ui-automator>
- [32] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü. 2017. AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 103–114. <https://doi.org/10.1109/ICSME.2017.72>
- [33] F. Horváth, S. Bognar, T. Gergely, R. Racz, A. Beszedes, and V. Marinkovic. 2014. Code coverage measurement framework for Android devices. *Acta Cybernetica* 21, 3 (2014), 439–458.
- [34] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*.
- [35] C. Huang, C. Chiu, C. Lin, and H. Tzeng. 2015. Code Coverage Measurement for Android Dynamic Analysis Tools. In *2015 IEEE International Conference on Mobile Services*. 209–216. <https://doi.org/10.1109/MobServ.2015.38>
- [36] Shang-Yi Huang, Chia-Hao Yeh, Farn Wang, and Chung-Hao Huang. 2015. ABCA: Android Black-box Coverage Analyzer of mobile app without source code. In *Progress in Informatics and Computing (PIC), 2015 IEEE International Conference on*. IEEE, 399–403.
- [37] JaCoCo. 2018. *Java Code Coverage Library*. <http://www.jacoco.org/>
- [38] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.
- [39] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. 2018. QBE: QLearning-Based Exploration of Android Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 105–115. <https://doi.org/10.1109/ICST.2018.00020>
- [40] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for Android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–127.
- [41] N. Li, X. Meng, J. Offutt, and L. Deng. 2013. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report). In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 380–389. <https://doi.org/10.1109/ISSRE.2013.6698891>
- [42] Y. Li. 2016. *AndroCov. Measure test coverage without source code*. <https://github.com/ylimit/androcov>
- [43] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 23–26. <https://doi.org/10.1109/ICSE-C.2017.8>
- [44] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 233–244.
- [45] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 399–410. <https://doi.org/10.1109/ICSME.2017.27>
- [46] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. 2017. InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 502–506. <https://doi.org/10.1109/SANER.2017.7884662>
- [47] Q. Lu, T. Wu, J. Yan, J. Yan, F. Ma, and F. Zhang. 2016. Lightweight Method-Level Energy Consumption Estimation for Android Applications. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 144–151. <https://doi.org/10.1109/TASE.2016.27>
- [48] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. 2019. Paladin: Automated generation of reproducible test cases for Android apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. 99–104.
- [49] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (Saint Petersburg, Russia) (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. <https://doi.org/10.1145/2491411.2491450>

- [50] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. 2012. A whitebox approach for automated security testing of Android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE press, 22–28.
- [51] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. <https://doi.org/10.1145/2931037.2931054>
- [52] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. Crashescope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 15–18.
- [53] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test*. ACM, 34–37.
- [54] Aleksandr Pilgun, Olga Gadyatskaya, Stanislav Dashevskiy, Yuri Zhauniarovich, and Artsiom Kushniarou. 2018. An Effective Android Code Coverage Tool. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. ACM, New York, NY, USA, 2189–2191. <https://doi.org/10.1145/3243734.3278484>
- [55] Qualcomm Technologies. 2019. *Snapdragon Profiler*. Retrieved 19/09/2019 from <https://developer.qualcomm.com/software/snapdragon-profiler>
- [56] V. Rubtsov. 2006. *EMMA: Java Code Coverage Tool*. <http://emma.sourceforge.net/>
- [57] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: Permission-aware GUI Testing of Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (Paderborn, Germany) (ESEC/FSE 2017)*. ACM, New York, NY, USA, 220–232. <https://doi.org/10.1145/3106237.3106250>
- [58] Scooter Software. 2019. *Beyond Compare*. Retrieved 23/09/2019 from <https://www.scootersoftware.com>
- [59] PassMark Software. 2018. *Passmark. Interpreting your Results from PerformanceTest*. https://www.passmark.com/support/performance-test/interpreting_test_results.htm
- [60] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBdroid: Beyond GUI Testing for Android Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (Urbana-Champaign, IL, USA) (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 27–37. <http://dl.acm.org/citation.cfm?id=3155562.3155570>
- [61] JetBrains s.r.o. 2017. Code Coverage. <https://www.jetbrains.com/help/idea/2017.1/code-coverage.html>
- [62] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (Paderborn, Germany) (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256. <https://doi.org/10.1145/3106237.3106298>
- [63] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *NDSS*.
- [64] Dávid Tengeti, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. 2016. Negative effects of bytecode instrumentation on Java source code coverage. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 225–235.
- [65] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).
- [66] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
- [67] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) (ASE 2018)*. ACM, New York, NY, USA, 738–748. <https://doi.org/10.1145/3238147.3240465>
- [68] R. Wiśniewski and C. Tumbleson. 2017. *Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps*. <https://ibotpeaches.github.io/Apktool/>
- [69] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
- [70] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

- [71] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. 2016. Target Directed Event Sequence Generation for Android Applications. arXiv:1607.03258 [cs.SE]
- [72] K. Yang. 2018. *APK Instrumentation Library*. Retrieved 06/02/2018 from <https://github.com/kelwin/apkil>
- [73] Q. Yang, J. J. Li, and D. M. Weiss. 2009. A Survey of Coverage-Based Testing Tools. *Comput. J.* 52, 5 (Aug 2009), 589–597. <https://doi.org/10.1093/comjnl/bxm021>
- [74] C. Yeh and S. Huang. 2015. CovDroid: A Black-Box Testing Coverage System for Android. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. 447–452. <https://doi.org/10.1109/COMPSAC.2015.125>
- [75] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (March 2012), 67–120. <https://doi.org/10.1002/stv.430>
- [76] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 117 (2016), 334–356.
- [77] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for android: Are we really there yet in an industrial case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 987–992.
- [78] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.
- [79] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. 2014. Fsquadra: fast detection of repackaged applications. In *IFIP Annual Conference on Data and Applications Security and Privacy XXVIII*, Vol. 8566. Springer, 130–145.
- [80] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. 2015. Towards Black Box Testing of Android Apps. In *The Tenth International Conference on Availability, Reliability and Security*. IEEE, 501–510.