is why some black-box testing systems, e.g., [46, 16], use only open-source apps for experimental validation, where the source code coverage could be measured by popular tools developed for Java, such as EMMA [45] or JaCoCo [31].

In the absence of source code, code coverage is usually measured by instrumenting the bytecode of applications [36]. 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 [51, 36]. 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 [30], there have been so far only few attempts to track code coverage for Android applications at the bytecode level [62], and they all still have limitations. The most significant one is the *coarse granularity* of the provided code coverage metric. For example, ELLA [21], InsDal [40] and CovDroid [60] measure code coverage only at 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. [30] and Zhauniarovich et al. [64] 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., [40, 12, 60]).

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., [43, 48, 12]. 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 incurred 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 [30, 64] have only reported the number of successfully instrumented apps[3], we also verified whether apps can be successfully executed after instrumentation. We report that **96.9%** have been successfully executed on the Android emulator – it 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 also not prohibitive. With the benchmark PassMark application [47], the instrumentation code added by ACVTool introduces 27% of CPU overhead, while our evaluation of executions of original and repackaged app version by Monkey [24] show that there is **no significant runtime overhead** for real apps (mean difference of timing 0.12 sec).
  - We have evaluated whether ACVTool reliably measures the bytecode coverage by comparing its results with those reported by JaCoCo [31], a popular code coverage tool for Java that requires the source code. Our results show that the ACVTool results can be **trusted**, as code coverage statistics reported by ACVTool and JaCoCo are highly correlated.
  - By integrating ACVTool with Sapienz [43], an efficient automated testing framework for Android, we demonstrate that our tool can be **useful** as an integral part of an automated testing

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

or security analysis environment. We show that fine-grained bytecode coverage metric is better in revealing crashes, while activity coverage measured by Sapienz itself shows performance comparable to not using coverage at all. Furthermore, our experiments indicate that different levels of coverage granularity can be combined to achieve better results in automated testing.

- 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 at `https://github.com/pilgun/acvtool`.

ACVTool can be readily used with various dynamic analysis and automated testing tools, e.g., IntelliDroid [56], CopperDroid [50], Sapienz [43], Stoat [49], DynoDroid [42], CuriousDroid [14] and the like, to measure code coverage. This work extends our preliminary results reported in [44, 20].

This paper is structured as follows. We give some 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 instrumentation approach. In Section 5, we report on the experiments we performed to evaluate the effectiveness and efficiency of ACVTool, and to assess how compliant is the coverage data reported by ACVTool to the data measured by the JaCoCo system on the source code. Section 6 presents our results on integrating ACVTool with the Sapienz automated testing framework, and discusses the contribution of code coverage data to bug finding in Android apps. Then we discuss the limitations of our prototype and threats to validity for our empirical findings in Section 7, and we overview the 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 resources 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 [18]. 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 the most JVM implementations that are stack-based, DVM is a register-based virtual machine[4]. 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 method's register set.

Since raw Dalvik binaries are difficult for human understanding, several intermediate representations have been proposed that are more analyst-friendly: `smali` [32, 27] and `Jimple` [52]. In this work, we work with `smali`, which is a low-level programming language for the Android platform. `Smali` is supported by Google [27], and it can be viewed and manipulated using, e.g., the *smalidea* plugin for the IntelliJ IDEA/Android Studio [32].

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 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.

[4] We refer the interested reader to the official Android documentation about the Dalvik bytecode internals [25] and the presentation by Bornstein [10].

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) [63]. 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, such signatures cannot be used to verify the authenticity of the developer of an application being installed for the first time, 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 [3]. 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 [51], automatic test generation [43], and test prioritization [51]. In particular, security testing of Android apps falls under the black-box testing category, as the source code of third-party apps is rarely available: there is no requirement to submit the source code to Google Play. Still, Google tests all submitted apps to ensure that they meet the security standards[5]. It is important to understand how well a third-party app has been exercised in the black-box setting, and various Android app testing tools are routinely evaluated with respect to the achieved code coverage [30, 34, 17, 53].

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 [3]. 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 [31] and EMMA [45]. On the other hand, the Android community often uses the *activity* coverage metric, that counts the proportion of executed activities [43, 6, 62, 14] (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 [10]. 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 [51, 36]. 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 black-box testing scenario, and we resort to measuring the bytecode instruction coverage.

3 ACVTool Design

ACVTool allows 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 the 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 UI event generator Monkey [24], and we have integrated it with the Sapienz tool [43] 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

[5] `https://www.android.com/security-center/`

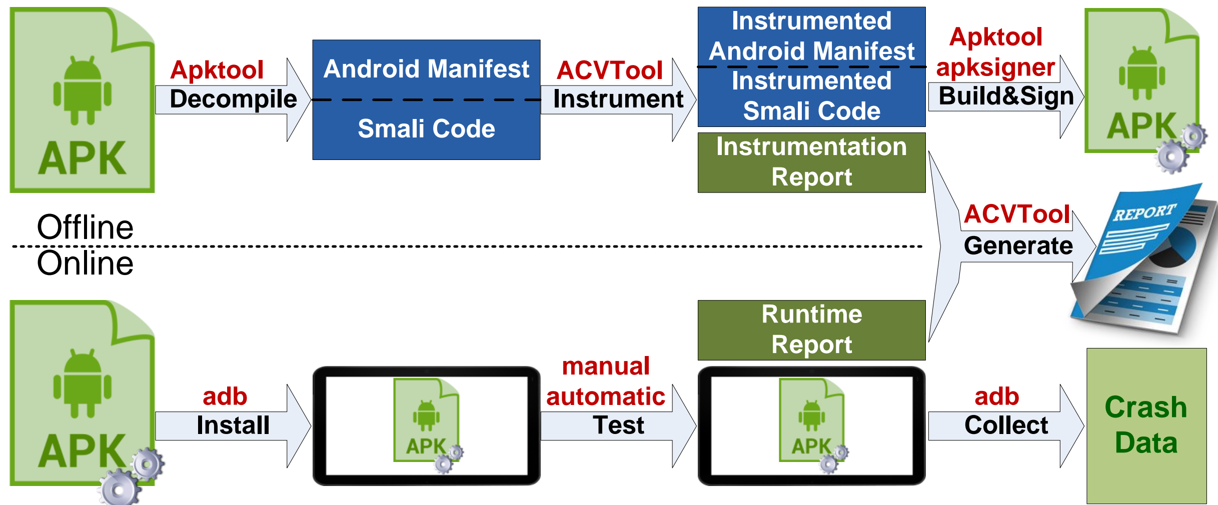


Figure 1: ACVTool workflow

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 the nutshell, this process consists of several steps depicted in the upper part of Figure 1. The original Android app is first decompiled using `apktool` [54]. Under the hood, `apktool` uses the `smali/backsmali` disassembler [32] to disassemble `.dex` files and transform them into `smali` representation. To track the execution of the original `smali` instructions, we insert 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 upon native Android tools and the 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 [64] by developing a special `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.

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

Figure 2: ACVTool *html* report

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. Currently, ACVTool can instrument an application to collect instruction-, method- and class-level coverage information.

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 the `adb shell am instrument` command. 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 [24], IntelliDroid [56], or Sapienz [43]. 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 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`), such that 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*, which 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 [31]. 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

Huang et al. [30] proposed two different 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

```
.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
```

Figure 3: Covered `smali` instructions highlighted by ACVTool

lead to reaching the upper limit of method references per `.dex` file (65536 methods, see [26]). 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: Original *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: Original *smali* code example.

4.1 Bytecode representation

To instrument Android apps, ACVTool relies on the `apkil` library [58] that creates a tree-based structure of `smali` code. The `apkil`'s tree 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.

Unfortunately, `apkil` has not been maintained since 2013. Therefore, we adapted it to enable support for more recent versions of Android. In particular, we added the annotation support for classes and methods, which has appeared in the Android API 19, and has been further extended in the API 22. We plan to support the new APIs in the future.

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

```
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 yellow lines highlight the added instructions.

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 app 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 [26], therefore the addition of new registers could make them malformed. Similarly, we cannot easily use new registers in the end of the stack because method parameters registers must always be the last ones.

To overcome this issue, we use the following hack. 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 the 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 [26] (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 the first command in the paired instructions. These excluded instructions are *untraceable* for our approach, and we do not consider them to be a 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, the 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 percentage of the instrumented apps that can run on an emulator. We call these apps *healthy*[6]. 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 black-box testing may be time-demanding (e.g., Sapienz [43] tests each application for hours), our goal is to ensure that the instru-

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

mentation 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 runtime overhead by using the benchmark PassMark application [47], by comparing executions of original and instrumented app versions, and by measuring the increase in `.dex` file size.

Compliance with other tools. We compare the coverage data reported by ACVTool with the coverage data measured by JaCoCo [31] which relies upon white-box approach and requires source code. 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 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 that crashed upon startup. These crashes could be 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 [43] (these experiments are reported in the next section). We excluded these unhealthy apps from the consideration. In total, our **Google Play benchmark** contains **832** healthy apps. The apk sizes in this set range from 20KB to 51MB, with the average apk size 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-Droid[7] dataset of open source Android apps (1330 application projects as of November 2017). We managed to `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 versions[8].

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 the 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 reassembled, repackaged, rebuilt (with `apktool`, `zipalign`, and `apksigner`) and installed every healthy Google Play and F-Droid app on a device. In Google Play set, one repackaged app had crashed upon startup, and `apktool` could not repackage 22 apps, raising `AndrolibException`. In the F-Droid set, `apktool` could not 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).

[7] `https://f-droid.org/`

[8] Gradle 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.

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

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` threw `ExceptionWithContext` stating that a register is 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 absence of Retrofit annotation (four `IllegalStateException` and one `IllegalArgumentException`), 1 case – `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 [64] (65%) and the tool by Huang et al. [30] (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 comparing to the testing time usual in the black-box setting that could easily reach several hours.

5.3.2 Runtime overhead

Running two copies with Monkey To assess the runtime overhead induced by our instrumentation in a real world setting, we ran the original and instrumented versions of 50 apps randomly chosen from our dataset with Monkey [24] (same seed, 50ms throttle, 250 events), and timed the executions. This experiment showed that our runtime overhead is insignificant: mean difference of timing was 0.12 sec, standard deviation 0.84 sec. While this experiment does not quantify the overhead precisely, it shows that our overhead is not prohibitive in a real-world test-case scenario. Furthermore, we have not observed any significant discrepancies in execution times, indicating that instrumented apps' behaviour was not drastically different from the original ones' behaviour, and there were no unexpected crashes. Note that in some cases two executions of the same app with the same Monkey script can still diverge due to the reactive nature of Android programs, but we have not observed such cases in our experiments.

Table 2: PassMark overhead evaluation

Granularity of instrumentation	Overhead	
	CPU	`.dex` size
Only class and method	+17%	+11%
Class, method, and instruction	+27%	+249%

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%)

PassMark overhead To further estimate the runtime overhead we used a benchmark application called PassMark [47]. 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. Research community has previously used this app to benchmark their Android related-tools (e.g., [7]).

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 version to collect full coverage information at the class, method and instruction level; and another version to log only class and method-level coverage.

Table 2 summarizes the performance degradation of the instrumented PassMark version in comparison to the original app. When instrumented, the size of Passmark `.dex` file increased from 159KB (the original version) to 178KB (method granularity instrumentation), and to 556KB (instruction granularity instrumentation). We have run 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, 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.

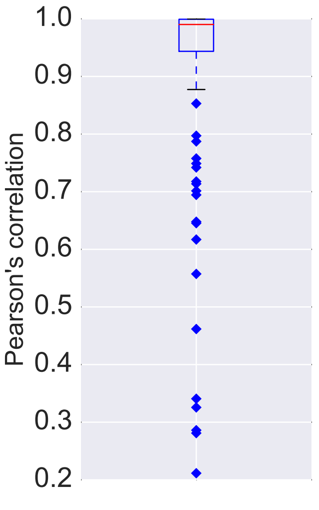
Evaluation with PassMark is artificial for a common app testing scenario, as the PassMark app stress-tests the device. However, from this evaluation we can conclude that performance degradation under the ACVTool instrumentation is not prohibitive, especially if it is used with modern hardware.

Dex size inflation As another metrics 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 [40] 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.

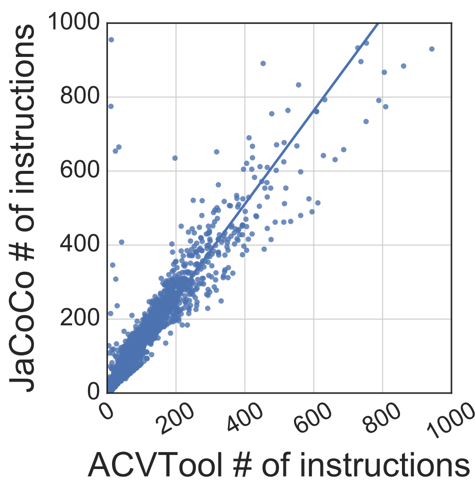
Conclusion: ACVTool introduces an off-line instrumentation overhead that is negligible considering the total duration of testing, which can last hours. The run-time overhead in live testing with Monkey is negligible. In the stress-testing with the benchmark PassMark app, ACVTool introduces 27% overhead 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 JaCoCo

When the source code is available, developers can log code coverage of Android apps using the JaCoCo library [31] that could be integrated into the development pipeline via the Gradle plugin. We used the



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



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

Figure 4: Compliance of coverage data reported by ACVTool and JaCoCo.

coverage data reported by this library to evaluate the correctness of code coverage metrics reported by ACVTool.

For this experiment, we used only 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 avoided creating app-specific tests, and could run any automatic testing tool. Due to the diversity of project structures and versioning of Gradle, there were many faulty builds. In total, we obtained 141 apks correctly instrumented with JaCoCo, i.e., we could generate JaCoCo reports for them.

We ran two copies of each app (instrumented with ACVTool and with JaCoCo) on the Android emulator using the same Monkey [24] scripts for both versions. Figure 4a shows the boxplot of 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. Overall, the boxplot demonstrates that code coverage logged by ACVTool is strongly correlated with code coverage logged by JaCoCo.

The boxplot in Figure 4a contains a number of outliers. Discrepancies in code coverage measurement have appeared due to several reasons. First of all, as mentioned in Section 4, ACVTool does not track some instructions. It is our choice to not count those instructions towards *covered*. In our F-Droid dataset, about half of app methods consist of 7 *smali* instructions or less. Evidently, the correlation of covered instructions for such small methods can be perturbed by these untraceable instructions.

The second reason for the slightly different code coverage reported is the differences in the `smali` code and Java bytecode. Figure 4b shows a scatterplot of method instruction numbers in `smali` code (measured by ACVTool, including the "untrackable" instructions) and in Java code (measured by JaCoCo). Each point in this Figure corresponds to an individual method of one of the apks. The line in the Figure is the linear regression line. The data shape demonstrates that the number of instructions in `smali` code is usually slightly smaller than the number of instructions in Java bytecode.

Figure 4b 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 are 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 [10].

Table 4: Crashes found by Sapienz in 799 apps

Coverage metrics	# unique crashes	# faulty apps	# crash types
Activity coverage	547 (47%)	381	30
Method coverage	578 (50%)	417	31
Instruction coverage	612 (53%)	429	30
Without coverage	559 (48%)	396	32
Total	1151	574	37

Conclusion: overall, we can summarize that code coverage data reported by ACVTool generally agree 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.

6 Contribution of Code Coverage Data to Bug Finding

To assess the usefulness of ACVTool in practical black-box testing and analysis scenarios, we integrated ACVTool with Sapienz [43] – 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) ensure that ACVTool fits into a real automated testing/analysis pipeline; (2) evaluate whether 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 [45] (reports source code statement coverage); ELLA [21] (reports method coverage); and its own 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 [43] has not evaluated the impact of code coverage metric used on the discovered crashes population.

Our previously reported experiment with JaCoCo suggests 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 integrates capability to measure 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.

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.

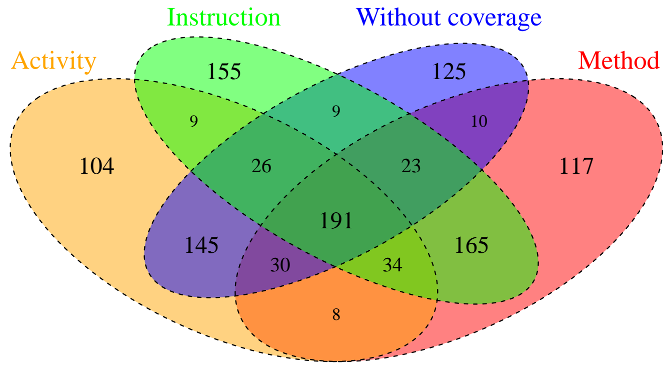
On average, each app has been tested by Sapienz for 3 hours for each coverage metric. After each run, we collected the crash information (if any), which included the components of apps that crashed and Java exception stack traces.

In this section we report on the results of crash detection with different coverage metrics and draw conclusions about how different coverage metrics contribute to bug detection.

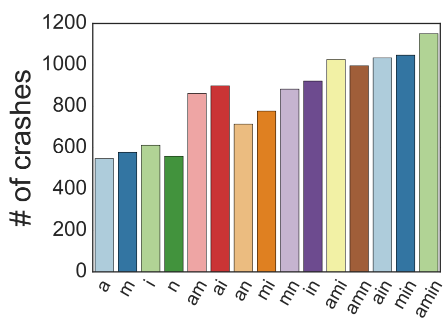
6.1 Descriptive statistics of crashes

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 unique 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 574 apps out of 799 to be faulty (at least one crash detected), and it has logged 1151 unique crashes with the four coverage conditions. Figure 5a summarizes the crash distribution for the coverage metrics. The intersection of all code coverage conditions' results contains 191 unique crashes (18% of total crash population). Individual coverage metrics have found 53% (instruction coverage), 50% (method coverage), 48% (without coverage), and 47% (activity coverage) of the total found crashes.



(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).

Figure 5: Crashes found by Sapienz.

Our empirical results suggest that coverage metrics at different granularities can find distinct crashes. Particularly, we note the tendencies for instruction and method coverage, and for activity coverage and Sapienz without coverage data to find similar crashes. This result indicates that activity coverage could be too coarse-grained and comparable in effect to ignoring coverage data at all. Instruction and method-level coverage metrics, on the contrary, are relatively fine-grained and are able to drive the genetic algorithms in Sapienz towards different crash populations. Therefore, it is possible that a combination of a coarse-grained metric (or some executions without coverage data) and a fine-grained metric measured by ACVTool could provide better results in testing with Sapienz.

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 behavior on multiple runs

Before assessing whether a combination of metrics could be beneficial for evolutionary testing, we look at assessing the impact of randomness on Sapienz' results. Like many other automated testing tools for Android, Sapienz is non-deterministic. Our findings may be affected by this. 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 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 significantly larger proportion of total crash population (52% from 42%). Sapienz without coverage data also shows better performance over multiple runs (51% from 43%). Yet, the instruction coverage has only slightly increased performance (54% from 50%), while the

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	86 (42%)	184 (52%)
Method coverage	104 (51%)	174 (49%)
Instruction coverage	103 (50%)	190 (54%)
No coverage	89 (43%)	180 (51%)
Total	203	351

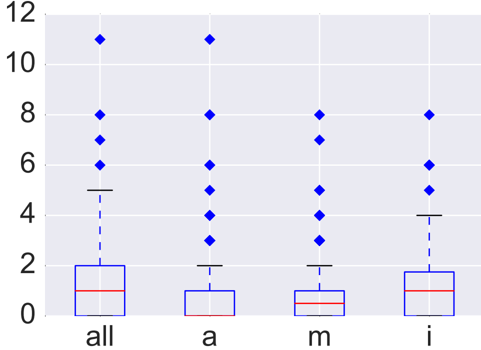


Figure 6: Boxplots of crashes detected per app (a stands for activity, m for method, and i for instruction, respectively).

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

Statistics	1 run × 3 metrics	3 runs × 1 metric		
		activity	method	instruction
Min	0	0	0	0
1st. Quartile	0	0	0	0
Mean	1.20	1.02	0.97	1.08
Median	1	0	0.5	1
3rd. Quartile	2	1	1	1.75
Max	11	11	8	8

method coverage has fared worse (49% from 51%). These findings suggest that the coverage metrics are not interchangeable, and even with 5 repetitions they are not able to find all crashes that were detected by other metrics. Our findings in this experiment are consistent with the previously reported smaller experiment that involved only 100 apps (see [20] for more details).

6.2.2 The Wilcoxon signed-rank test

The previous experiment indicates that even repeating the runs multiple times does not allow any of the code coverage metrics to find the same amount of bugs as all metrics together. The instruction coverage seems to perform slightly better than the rest in the repeating runs, but not a lot. We now fix the time that Sapienz spends on each apk[9], and we want to establish whether the amount of crashes that Sapienz can find in an apk with 3 metrics is greater than the amount of crashes found with just one metric but with 3 attempts. This will suggest that the combination of 3 metrics is more effective 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 6. The summary statistics show that Sapienz equipped with 3 coverage metrics has found,

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

on average, more crashes per apk than Sapienz using only one metric but executed 3 times. To verify this, we apply the Wilcoxon signed-rank test [55]. This statistical test is appropriate, as our data is paired but not necessarily normally distributed.

The *null-hypothesis* for the Wilcoxon test is that there is no difference which metric to use in Sapienz. Then, on average, Sapienz with 3 metrics will find the same amount of crashes in an app as Sapienz with 1 metric but run 3 times. *Alternative hypothesis* is that Sapienz with several coverage metrics will consistently find more crashes. Considering the standard significance level of 5%, on our data, the results of the Wilcoxon test rejected the null-hypothesis for the activity and method coverage metrics (p-values 0.002 and 0.004, respectively), but not for the instruction coverage (p-value 0.09, which is more than the threshold 0.05). Cohen's d for effect size are equal, respectively, 0.297, 0.282 and 0.168 for each of the metrics.

These results show that it is likely that a combination of coverage metrics achieves better results than only activity or method coverage. The outcome for the instruction coverage is inconclusive. We posit that a larger-size experiment could provide a more clear picture.

Indeed, our findings from this last experiment are also not fully consistent with the previously reported experiment on a smaller set of 100 apps [20]. The difference could be explained by two factors. First, we have used only healthy instrumented apps in this experiment (the ones that did not crash upon installation). The experiment reported in [20] did not involved 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 the instruction coverage was able to find more bugs in them.

6.3 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. Furthermore, our results establish that a fine-grained code coverage measurement tool, such as ACVTool, can be helpful in improving automated testing tools that rely on code coverage. Indeed, in our experiment with 799 Google Play apps, the instruction-level coverage has been able to identify more faults in a single Sapienz run than other considered coverage metrics. Moreover, we compared three coverage metrics executed once with individual metrics executed 3 times, and the method and activity coverage metrics were found less effective by the Wilcoxon test of means, while the instruction coverage alone could be reasonably effective in finding bugs.

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 [15]. In our experiment with 799 apps, Sapienz without coverage data has shown results most comparable to Sapienz equipped with activity coverage. This finding could indicate that activity coverage is too coarse-grained for being a useful fitness-function in automated Android testing. On the other hand, our experiment with repeating executions 5 times shows that no coverage metrics is able to find the vast majority of the total found crash population. This result indicates that different granularities of coverage are not directly interchangeable. Further investigation of these aspects could be a promising line of research.

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. An inherent limitation of our approach is that the 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 instrumented. Furthermore, apps can employ various means to prevent repackaging, e.g., they can check signature at the start, and stop executing in case of a failed signature check. This limitation is common to all tools that instrument applications (e.g., [64, 21, 30, 60, 40]). 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., [30, 64]). Therefore, ACVTool is highly practical and reliable. We examine the related work and compare ACVTool to the available tools in the subsequent Section 8.

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 by 27% in the instruction-level instrumented version. This is a critical scenario, and the overhead for an average app will be much smaller, what is confirmed by our experiments with Monkey.

We assessed that the code coverage data from ACVTool is compliant to the measurements from the well-known JaCoCo [31] tool. 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.

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 right 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.

Another limitation of our current approach is the limit of 255 registers per method. 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 [23]. We plan this extension as the future work.

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, what 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 prototype 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%, what is still much higher than other tools.

Also, the current implementation still has a few issues (3.3% of apps have not survived instrumentation) that 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 different granularities of code coverage could be combined for achieving time-efficient and effective bug finding.

At this point, it is not yet clear which coverage metric works best. However, the fine-grained instruction-level coverage provided with ACVTool has been able to uncover more bugs than other coverage metrics, on our sample. Further investigation of this topic is required to better understand exactly how granularity of

code coverage affects the results, and whether there are other confounding factors that may influence the performance of Sapienz and other similar tools.

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 amount 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 [24] as the underlying test engine, and thus it is not deterministic. It is possible that this indeterminism may have influence on our current results, and the results obtained with the different coverage metrics could converge on many runs. Our experiments with repeating executions 5 times indicate that this is unlikely. However, the success of Sapienz in finding bugs without coverage information shows that Monkey is powerful enough to explore significant proportions of code, even without evolution of test suites following a coverage-based fitness function. This threat warrants further investigation.

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.

The last but not least, we acknowledge that the 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. We have tried to address this threat to validity in two ways. First, we have manually inspected a selection of crashes to evaluate whether they appear due to instrumentation. We have not found such evidence. Second, we have run original and instrumented apks with Monkey to assess run-time overhead (as reported in Section 5), and we have not seen discrepancies in executions of these apps. If the instrumented version would crash unexpectedly, but the original one would continue running under that same Monkey test, this would be evident from the timings.

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 Stoat [49], AimDroid [29] or QBE [35] 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 also to investigate this issue.

To conclude, our experiment with Sapienz [43] has demonstrated that well-known search-based testing algorithms that rely on code coverage metrics can benefit from the fine-grained code coverage provided by ACVTool. Automated testing tools may be further improved by including several code coverage metrics with different levels of measurement granularity. This finding confirms the downstream value of our tool.

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 [42] 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 [48]. Recently, the survey by Choudhary et al. [17] has compared the most prominent testing tools that automatically generate app input events in terms of efficiency, including code coverage and fault detection. Two recent surveys, by Linares et al. [39] and by Kong et al. [34], 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 [28]. Supported coverage libraries are JaCoCo [31], EMMA [45], and the IntelliJ IDEA coverage tracker [33]. 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
					Instrumented	Executed	Instr. time (sec/app)	Run time (%)	
ELLA [21, 53]	method	Dalvik bytecode	Y	68 [53]	60% [53]	60% [53]	N/A	N/A	N/A
Huang et al. [30]	class, method, basic block, instruction	Dalvik bytecode (`smali`)	N	90	36%	N/A	N/A	N/A	Y
BBoxTester [64]	class, method, basic block	Java bytecode	Y	91	65%	N/A	15.5	N/A	N
Asc [48]	basic block, instruction	Jimple	Y	35	N/A	N/A	N/A	N/A	N
InsDal [40, 57, 41]	class, method	Dalvik bytecode	N	10	N/A	N/A	1.5	N/A	N
Sapienz [43]	activity	Dalvik bytecode	Y	1112	N/A	N/A	N/A	N/A	N
DroidFax [12, 11, 13]	instruction	Jimple	Y	195	N/A	N/A	N/A	N/A	N
AndroCov [9, 37]	method, instruction	Jimple	Y	17	N/A	N/A	N/A	N/A	N
CovDroid [60]	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 PassMark	**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 [21], InsDal [40], and CovDroid [60] measure code coverage only at the method level.

ELLA [21] is arguably one of the most popular tools to measure Android code coverage in the black-box setting, however, it is no longer supported. An empirical study by Wang et al. [53] has evaluated performance of Monkey [24], Sapienz [43], Stoat [49], and WCTester [62] 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.

Huang et al. [30] 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 much higher instrumentation rate, evaluated against much larger dataset.

BBoxTester [64] 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 [45], 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 [40] 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 [57], and for logging the number of method invocations in measuring the energy consumption of apps [41]. 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 average instrumentation time overhead of 1.5 sec per app, and average instrumentation code overhead of 18.2% of `dex` file size. ACVTool introduces smaller code size overhead of 11%, on average, but requires more time to instrument an app. On our dataset, 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 [60], another black-box code coverage measurement system for Android apps, transforms apk code into `smali`-representation using the `smali` disassembler [32] 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. [8] worked on instrumenting Android apps for improving their privacy and security via translation to Java bytecode. Zhauniarovich et al. [64] translated Dalvik into Java bytecode in order to use EMMA's code coverage measurement functionality. However, the limitation of such approaches, as reported in [64], 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 [48], DroidFax [12], and AndroCov [37, 9]. Jimple is a suitable representation for subsequent analysis with Soot [5], yet, unlike `smali`, it does not belong to the "core" Android technologies maintained by Google. Moreover, Arnatovich et al. [4] 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 [43, 48, 12, 38, 14] 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. [30], InsDal [40] and CovDroid [60]. 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 [19] and SwiftHand [16] 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., [30] and [64]. 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 [21] and BBoxTester[64]) or publicly unavailable tools ([30, 60]).

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. Also, as an interesting line of future work, we consider 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 the granularity of coverage is important, when used as a component of the fitness function in the black-box app testing. The second line of the future work for us is to expand our experiments to more apps and more testing tools, thus establishing better guidelines on which coverage metric is more effective and efficient in bug finding.

Acknowledgements

This work has been partially supported by Luxembourg National Research Fund through grants C15/IS/10404933/COMMA and AFR-PhD-11289380-DroidMod.

References

[1] dex2jar, 2017.

[2] K. Allix, T. F. Bissyande, J. Klein, and Y. L. Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471, May 2016.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2 edition, 2016.

[4] Yauhen Leanidavich Arnatovich, Hee Beng Kuan Tan, and Lwin Khin Shar. Empirical comparison of intermediate representations for android applications. In *26th International Conference on Software Engineering and Knowledge Engineering*, 2014.

[5] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 13–24, Piscataway, NJ, USA, 2017. IEEE Press.

[6] Tanzirul Azim and Iulian Neamtiu. 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*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[7] M. Backes, S. Bugiel, O. Schranz, P. v. Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 481–495, April 2017.

[8] Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, and Yves Le Traon. In-vivo bytecode instrumentation for improving privacy on android smartphones in uncertain environments, 2012.

[9] Nataniel P. Borges, Jr., Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, pages 133–143, New York, NY, USA, 2018. ACM.

[10] D. Bornstein. Google I/O 2008 - Dalvik Virtual Machine Internals, 2008.

[11] H. Cai, N. Meng, B. Ryder, and D. Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, pages 1–1, 2018.

[12] H. Cai and B. G. Ryder. Droidfax: A toolkit for systematic characterization of android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 643–647, Sep. 2017.

[13] H. Cai and B. G. Ryder. Understanding android application programming and security: A dynamic study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 364–375, Sep. 2017.

[14] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. Curiousdroid: Automated user interface interaction for android application analysis sandboxes. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, pages 231–249, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[15] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 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*, ICSE '17, pages 597–608, Piscataway, NJ, USA, 2017. IEEE Press.

[16] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[17] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[18] Mike Cleron. Android Announces Support for Kotlin, May 2017.

[19] Shuaifu Dai, Tao Wei, and Wei Zou. Droidlogger: Reveal suspicious behavior of android applications via instrumentation. In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pages 550–555, Dec 2012.

[20] Stanislav Dashevskyi, Olga Gadyatskaya, Aleksandr Pilgun, and Yury Zhauniarovich. 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*, CCS '18, pages 2216–2218, New York, NY, USA, 2018. ACM.

[21] ELLA. A tool for binary instrumentation of Android apps, 2016.

[22] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Trans. Softw. Eng. Methodol.*, 24(4):22:1–22:33, September 2015.

[23] Google. Enable Multidex for Apps with Over 64K Methods.

[24] Google. UI/Application Exerciser Monkey.

[25] Google. Dalvik Executable format, 2017.

[26] Google. Dalvik bytecode, 2018.

[27] Google. smali, 2018.

[28] Google. Test your app, 2018.

[29] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. LÃij. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 103–114, Sep. 2017.

[30] C. Huang, C. Chiu, C. Lin, and H. Tzeng. Code coverage measurement for android dynamic analysis tools. In *2015 IEEE International Conference on Mobile Services*, pages 209–216, June 2015.

[31] JaCoCo. Java code coverage library, 2018.

[32] JesusFreke. smali/backsmali.

[33] JetBrains. Code coverage, 2017.

[34] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyande, and J. Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, pages 1–22, 2018.

[35] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. Qbe: Qlearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115, April 2018.

[36] N. Li, X. Meng, J. Offutt, and L. Deng. 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)*, pages 380–389, Nov 2013.

[37] Y. Li. AndroCov. measure test coverage without source code, 2016.

[38] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26, May 2017.

[39] M. Linares-VÃąsquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, Sep. 2017.

[40] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. Insdal: A safe and extensible instrumentation tool on dalvik byte-code for android applications. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–506, Feb 2017.

[41] Q. Lu, T. Wu, J. Yan, J. Yan, F. Ma, and F. Zhang. Lightweight method-level energy consumption estimation for android applications. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 144–151, July 2016.

[42] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[43] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[44] Aleksandr Pilgun, Olga Gadyatskaya, Stanislav Dashevskyi, Yury Zhauniarovich, and Artsiom Kushniarou. An effective android code coverage tool. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2189–2191, New York, NY, USA, 2018. ACM.

[45] V. Rubtsov. Emma: Java code coverage tool, 2006.

[46] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. Patdroid: Permission-aware gui testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 220–232, New York, NY, USA, 2017. ACM.

[47] PassMark Software. Passmark. interpreting your results from performancetest, 2018.

[48] Wei Song, Xiangxing Qian, and Jeff Huang. Ehbdroid: Beyond gui testing for android applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 27–37, Piscataway, NJ, USA, 2017. IEEE Press.

[49] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 245–256, New York, NY, USA, 2017. ACM.

[50] K. Tam, S. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[51] D. Tengeri, F. HorvÃąth, ÃĄ. BeszÃľdes, T. Gergely, and T. GyimÃşthy. Negative effects of bytecode instrumentation on java source code coverage. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 225–235, March 2016.

[52] R. Vallee-rai and L. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. 2004.

[53] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 738–748, New York, NY, USA, 2018. ACM.

[54] R. Wiśniewski and C. Tumbleson. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps, 2017.

[55] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated, 2012.

[56] M. Y Wong and D. Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[57] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. Target directed event sequence generation for android applications, 2016.

[58] K. Yang. APK Instrumentation library, 2018.

[59] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, Aug 2009.

[60] C. Yeh and S. Huang. Covdroid: A black-box testing coverage system for android. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 447–452, July 2015.

[61] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[62] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 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*, FSE 2016, pages 987–992, New York, NY, USA, 2016. ACM.

[63] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: fast detection of repackaged applications. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 130–145. Springer, 2014.

[64] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards Black Box Testing of Android Apps. In *2015 Tenth International Conference on Availability, Reliability and Security*, pages 501–510, August 2015.