Computer Science

Analyzing the Code Coverage of Android Apps using the Exerciser Monkey

Corné Spek

Supervisors:
Dr. Olga Gadyatskaya & Nathan Daniel Schiele

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

29/07/2021

Abstract

With a market share of 72.84% [1] in June 2021, Android is the worlds most used operating system for mobile devices. In order for developers to test the applications they create, random testing can be an attractive option. Additionally, when third-party applications have to be tested that the source code is not available for, automated testing is a good approach. Exerciser Monkey [2] is Google's official tool that uses a simple random approach in order to test Android apps. Prior studies show that Monkey's simple approach to testing beats more sophisticated tools. However, these types of tools almost never fully explore an application. Earlier work on Monkey's code coverage usually does not test many different parameter combinations. In order to maximize the code coverage of an application that gets covered by Monkey, in this paper, the code coverage obtained by the different parameters and runtimes of Monkey are investigated and compared, as well as the amount of applications that crash while being tested. The experiment is tested on a dataset of 50 applications with 10 different parameter options. It is shown that changing the parameters of Monkey can have a positive influence on code coverage and application crashes found.

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 Android Applications . . . 2
1.3 Application Activities . . . 3
1.4 Development Challenges . . . 3
1.5 Automated Testing . . . 4

2 Related Work **6**
2.1 Automated Testing . . . 6
2.2 Measuring Code Coverage . . . 7
2.3 Monkey Testing . . . 8

3 Methods **10**
3.1 Tools . . . 10
3.1.1 App Benchmark . . . 10
3.1.2 Emulator . . . 10
3.1.3 Monkey . . . 10
3.1.4 ACVTool . . . 11
3.2 Experiment Pipeline . . . 12
3.3 Report Processing . . . 13
3.4 Tested Monkey Parameters . . . 14

4 Results **16**
4.1 Code Coverage . . . 16
4.2 Crashes . . . 17

5 Discussion **19**
5.1 Code Coverage . . . 19
5.2 Crashes . . . 20
5.3 Limitations . . . 20

6 Conclusions and Further Research **21**

References **25**

A Application Benchmark **26**

1 Introduction

1.1 Overview

With over 6.3 billion active smartphone subscriptions in 2021 and an estimated 7.5 billion in 2026 [3], interest in creating applications for mobile devices is ever growing. According to Google, over 3 billion active Android devices are in circulation. Google Play [4], the largest place for Android users to download and install apps, offers more than 2.8 million [5] apps to its users, with an average of nearly 3000 apps published daily [6]. With consumer spending reaching numbers as high as 64.9 billion dollars in the first half of 2021 [7], this highlights the incredible stakes in the mobile app development world.

Figure 1: The Android Logo

Android is an open-source mobile operating system primarily designed for mobile devices with touchscreens. It was first shown in 2007 and was developed by Google and the Open Handset Alliance [8]. The most recent stable version of Android is Android 11. Currently Android is maintained and developed by Google and the Android Open Source Project (ASOP) [9].

The Android software stack contains a couple distinct major components, being the following (see Figure 2):

1. Linux Kernel
2. Hardware Abstraction Layer
3. Android Runtime
4. Native `C/C++` Libraries
5. Java API Framework
6. System Applications

The Android operating system runs on a version of the Linux kernel that has changes made by Google. The specific kernel depends on the device. The kernel handles things such as device drivers, security, resource accessing, power management, managing memory and more. The Hardware Abstraction Layer (HAL) is a layer that contains interfaces to connect hardware to the Java API framework, for example a Bluetooth module. Next is Android Runtime (ART), which was the Dalvik before Android version 5.0 or API 21, consisting of processes run by applications with each

having their own instance of the Android Runtime. The Native `C/C++` Libraries layer contains many core components that are used to build other parts such as ART and HAL. Part of these libraries can also be exposed to the Java framework APIs, allowing application developers to use them. The Java API Framework (Android Framework in Figure 2) houses the Android APIs, written in Java. These API functionalities contain many classes and interfaces used for application development and simplifies reuse of core components. The highest layer is the System Applications layer, holding the core applications as well as third party and developer applications.

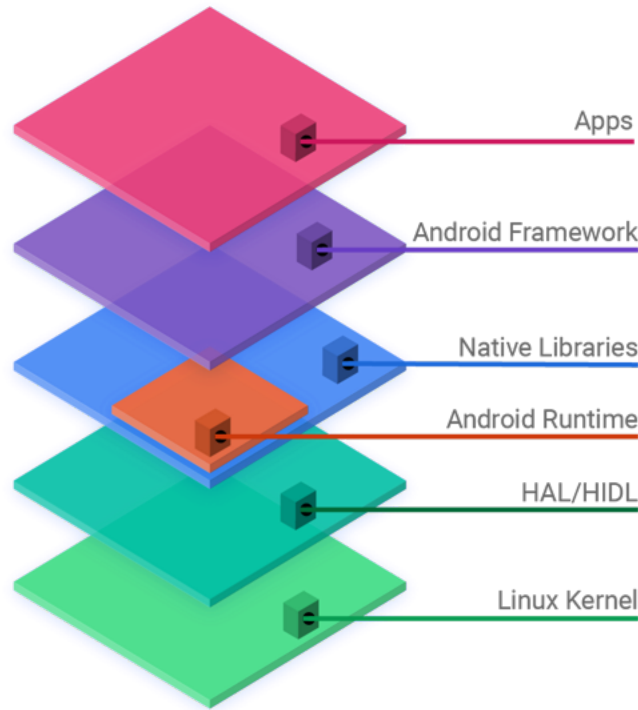

Figure 2: Android Software Stack. Source: [9]

1.2 Android Applications

Applications for the Android platform are mostly written in Java and Kotlin using the Android Software Development kit (SDK), with other programming languages also being supported. The SDK houses many features to support with the development of Android applications such as an emulator, testing tools, debugger, libraries, documentation and more. Third-party applications can be installed on a user's device by using an application store, with Google Play being the biggest, or via downloading and installing an Android Application Package (APK for short).

An APK can by created by compiling and packaging all parts of a program using Android Studio. Thus, an APK file contains every component of the program required for it to run. APK files contain the following components [10]:

- `META-INF/`: A directory containing the manifest file, the list of resources and the certificate/signature of the application.

- `lib/`: A directory that has the compiled code of the application which are split up into native libraries for specific device architectures.
- `res/`: A directory that has the resources that were not compiled into `resources.arsc`.
- `assets/`: A directory with raw resources and assets.
- `AndroidManifest.xml`: A file that that describes the name, version, access rights and contents of the APK file.
- `classes.dex`: The compiled classes of the application (in the .dex format) that are to be run on the device.
- `resources.arsc`: Pre-compiled resources that are used by the application, .arsc files.

1.3 Application Activities

Android applications have multiple entry points, for example a user can launch an application simply by opening it from the application icon on the homescreen or by tapping a notification produced by the application, which then opens the application. This is why applications differ from more traditional desktop programs, opening the program or application does not always start from the same point. Applications can be launched from many different points, such as via other applications, and the previously mentioned ways. "An *activity* is the entry point for interacting with the user" [11]. These activities are launched when an application uses a different application, while also being the interaction point with the user of the application. Applications usually consist of multiple of these activities, that together form the whole application but are not strongly linked together. The main activity is generally the first screen one sees when opening an application, after which users can launch other activities and thus use the application. So, applications can be launched and entered via many different ways and points, including via other applications.

1.4 Development Challenges

Developers of Android applications face numerous challenges when creating their apps. Among the biggest challenges are [12, 13, 14]:

1. *Multiple platforms and devices*: Mobile devices and versions have differences that require developers to make platform and/or version specific changes.
2. *Testing of applications*: The majority of developers test their applications manually. Trust in automated test tools is not very high and thus manual testing is most common.
3. *Monitoring Tools*: Tools for monitoring the status of an application and it's device are insufficient.
4. *Problems with emulators*: The emulators that provide developers with a hardware environment for Android devices seem to lack some real features that makes testing on these more difficult.

With the many challenges that developers face along with time and budget concerns, automated testing plays an important role in lowering the workload and gaining confidence in the robustness and quality of an application.

1.5 Automated Testing

Effective stress testing and quality assurance of Android applications is becoming more and more important with the growth of the Android platform. Developers want to be sure that their developed applications are ready to be used and up to their expected level of quality. Most testing in the world of mobile development is still done manually. Due to the earlier challenges in development mentioned above, having to spend time manually testing applications on multiple devices and versions can quickly become very time and cost consuming.
Besides testing applications that are in development, testing and analyzing behaviour of third-party applications whose source code is not available is also important. Security is a big concern when downloading or working with an application from an untrustworthy source. Automated testing can be a solution to this problem, being able to execute an application and then analyzing it's behaviour. From these kinds of analyses, potentially malicious or otherwise unwanted behaviour can be identified. Important here is that automated testing tools should explore as much of an application as possible, in order to identify if those parts of the application are harmful.
Following these concerns, automating the test process for the development of mobile applications could solve many of these issues and lead to an increase in the overall quality of developed applications. Moreover, having an automated tool that can efficiently explore as much of an application as possible is important for analyzing third-party applications from a security standpoint, to produce accurate analyses and save time and resources.

Automated input testing is a method to test an application by generating inputs and events, such as screen touches or rotations, and then simulating these inputs and events to the application that is being tested. Ultimately, the goal of such a testing approach is to cover as much of the application's functionality and behaviour as possible, or to crash the application. Tools for automated testing can use a variety of strategies when deciding what type of events or inputs to send to the application that is being tested. Strategies can range from simple randomly chosen events and inputs to complex models that are generated based on an application's characteristics. A systematic overview/mapping of tools produced can be found at [15] and [16].

Applications that are to be tested can be looked at as black-box, white-box or grey-box. When testing an application in a black-box setting, no prior information about the internal workings of the application is available. The white-box setting has access to all information, including source code, of the application. In a grey-box setting, information about the internals of the application is available. The amount of information available can determine which tools can be used, or how these tools operate.

In order to quantify and measure how much of an application is actually explored by an automated testing tool, a code coverage measurement is used. Code coverage is a measurement that keeps track of how much of a program's source or byte code is executed during a test run. Coverage can be measured at multiple levels, such as the class, method or instruction level.
Measuring code coverage in a white-box setting is supported by Android Studio along with other tools. However, in order to measure coverage in a white-box setting, one must have access to the source code. When wanting to test a third-party application, this will not always be the case.
In a black-box setting, code coverage can also be measured. This is done by instrumenting the byte

code. This approach is more suited for third-party applications, since the source code is not always available. Several tools exist to measure code coverage in a black-box setting, more is explained in the related work and methods sections.

In this paper, Monkey [2] is analyzed. It is part of the Android SDK and is thus maintained by Google. Monkey is a tool that can automatically generate pseudo-random streams of user events. By default, Monkey runs on any emulator or device and is used to stress test applications. While Monkey runs it's events, it also analyzes the system, catching errors and other exceptions, as well as attempts to navigate to other packages installed on the device. Multiple parameters or options can also be used, such as event frequency and amounts. This is explained in more detail in the Methods section. Monkey follows a black-box approach, not needing any source code or other internal information about an application in order to function. Monkey is one of the easiest and most used tools for this purpose, thus making understanding it and exploring different ways to use Monkey important.

While there are multiple papers and studies that test the effectiveness of Monkey in terms of code coverage, few explore different combinations of parameters and other settings of the Monkey in order to achieve the best possible performance. In this thesis the performance in terms of code coverage and crashes of multiple different parameters and event distributions is evaluated on a benchmark of 50 applications.

Aside from the default event distribution, 10 different event distributions are tested on the application benchmark for 5 minutes per application and their code coverage is measured on the instruction, method and class level. The amount of applications that crash during the testing is also reported. Results show that changing the event distribution can increase code coverage as well as the amount of crashes during testing with Monkey.

A link to the code used for performing experiments and parsing results can be found at:
`https://github.com/kw-corne/monkey-coverage`

The rest of this thesis is structured as follows:

- The related work on what others have found and tried in the past is analyzed.
- Methods and tools used to setup the experiments are explained in detail.
- Findings and results are presented.
- Discussion about the results that were produced.
- Conclusions based on the findings are made clear.
- Future research is proposed.

2 Related Work

In this section, we look at related work when it comes to automated testing of Android apps in general, various ways to measure the code coverage in a black box setting and some studies on the Exerciser Monkey itself.

2.1 Automated Testing

A survey conducted by Vásquez et al. [17] on how Android developers test their applications found that although there are a lot of different tools for automated testing, they still have a low usage rate. 60.78% of the participants reported that they had no experience with using tools for random testing such as Monkey. Only 11 participants out of the surveyed 102 reported that they used Monkey in more than 10% of their testing efforts. According to the survey the tools still face issues such as the impact of the discovered bugs/crashes, difficulties in reproducing steps and a lack of expressiveness. The survey found that the majority of testing is still done manually.

Villanes et al. [18] analyzed questions asked on Stack Overflow regarding mobile testing. They found that Appium, Espresso, Monkey, and Robotium had increasing interest among developers on the Android platform.

Besides Monkey, many different tools have been developed for the automated testing of Android apps, such as for example: Sapienz [19], Stoat [20], WCTester [21] and Dynodroid [22], among others. Almeida et al. [16] carried out a systematic mapping of 80 of these tools and found that 42% of them use model-based testing, and another 40% implement GUI testing. Model based testing is a technique where a model is first generated based on information about an application, after which this model is used to generate inputs and events. GUI testing is a method whereby an application's GUI is dynamically analyzed in order to generate inputs and events. They also mapped out the differences between some tools when it comes to the various techniques and methods these tools use.

Choudhary et al. [23] compared the effectiveness of multiple automated testing tools on four metrics: code coverage, fault detection, ability to work on multiple platforms and their ease of use. Their conclusion was that even though more sophisticated exploration techniques exist, they were still outperformed by the random exploration techniques from Monkey and Dynodroid. Based on the four criteria mentioned earlier they concluded that Monkey is the winner among the tools they evaluated, due it achieving the best coverage and reporting the most failures. Moreover they also found it was very easy to use and available on all platforms. But they also thought that because every tool has a strong point that one should consider which tool to use in which case.

Wang et al. [24] found that among multiple state-of-the-art tools for generating tests on real-world industrial apps Monkey still achieved the highest method and activity coverage on over half of the apps that were tested. They advise to use a combination of tools to achieve the highest coverage rates and reported crashes.

From the above findings and papers, we can see that Monkey remains one of the most user

friendly and widely used tools when it comes to automated testing.

2.2 Measuring Code Coverage

In order to measure the code coverage achieved by an automated testing tool, we need a tool that can measure code coverage in a black-box setting, so without any source code available. Multiple tools have been created for this purpose, such as: ACVTool [25], COSMO [26], ELLA [27], BBoxTester [28] and InsDal [29].

ACVTool [25] measures code coverage at the instruction level, using the `smali` representation of Android bytecode. According to the authors ACVTool can successfully instrument and execute 96.9% of Android applications. This also makes it suitable for large scale testing, as it does not add too much overhead or long instrumentation times. ACVTool generates reports by combining runtime reports and instrumentation reports to map probes to their original instructions. Reports generated are available in `html` and `xml` formats making them useful for visual inspection as well as automated report inspection and processing.

COSMO [26] is an automated instrumenter that works on Gradle-based [30] as well as on compiled apps. A compiled app is first instrumented by instrumentng the Java bytecode with JaCoCo [31]. The instrumented version is converted back to Dalvik and added to the original APK file. Once the app has been installed and explored a report can be generated. The authors report that the main difference between ACVTool and COSMO for the compiled app process is that developers do not need to grant additional permissions to the instrumented apps. COSMO succesfully instruments 86.9% of apps and 71.6% run without errors.

ELLA [27] is another tool that can be used to measure code coverage for Android apps. It instruments apps at the method level, by inserting probes and tracking their execution. It can also track the trace of executed methods and the values of the arguments passed at call-sites and more. Sadly, ELLA is no longer supported. Reports are generated containing a list of method signatures and a list of method identifiers that were executed. Like ELLA, InsDal [29] also measures code coverage at the method level.

BboxTester [28], proposed by Zhauniarovich et al., is a tool that can be used to generate code coverage reports and other coverage metrics for apps that the source is not available of. BBoxTester instruments `jar` files using EMMA [32], which are then assembled back into a new APK. BBoxTester instruments 65% of apps succesfully.

Considering the different tools, ACVTool is used in the experiments that are performed for its high instrumentation rate, low instrumentation times and low overhead. ACVTool is also suited for lage scale testing and easy to be integrated into a testing/experiment pipeline. The reports are also clear and easy to parse.

2.3 Monkey Testing

As previously discussed, Choudhary et al. [23] compared multiple automated testing tools and found Monkey to be the best performing. Their application benchmark consisted of 68 applications retrieved from papers about automated testing tools. They used the Monkey with a 200 millisecond delay between actions. From their experiments they reported the mean coverage over a 60 minute period to be about 45%. From their data we can also see that after 5 minutes, code coverage stagnates heavily for the tools they tested. No different parameter settings using Monkey were tested in their experiments.

Patel et al. [33] studied how Monkey's parameters affect code coverage on the line, block, method and class levels. Applications tested were 15 very famous applications, and 64 popular applications. They also compared code coverage achieved by Monkey against code coverage achieved by manual exploration. They found that changing Monkey's event distribution did not lead to any significant increases in achieved coverage. From their testing they also found that Monkey is on average 2-3% behind manual exploration when it comes to code coverage. However, when testing different event distributions they only tested increasing the touch, motion, trackball, navigation and major navigation events at 75%. The apps were tested for 5 minutes.

An empirical study conducted by S. Paydar [34] on how effective Monkey is in testing Android applications on robustness and responsiveness found that Monkey is a very effective and highly recommended tool for these purposes. The study tested Monkey on real-world applications and fault-injected applications, so applications where the errors are known before testing. 150 applications from F-Droid and 150 Iranian applications were tested for the real-world application part. They defined some metrics, being: crash rate, ANR-rate, failure rate, crash delay, ANR-delay and failure delay. 67% of the apps in first dataset (F-droid applications) did not crash, 52% of apps in their second dataset (Iranian applications) did not crash. Thus, Monkey seems very reliable in detecting robustness errors in real-world applications. They also found that the default Monkey in the Android Studio IDE has a low focus of 5.5%, and that the Monkey is less sensitive to faults that cause a delay lower than 5 seconds. As future work they proposed more experiments on the distribution of events generated by Monkey. They did however not test any different parameters of the Monkey.

Mohammed et al. [35] studied the difference between human and Monkey testing of Android applications. They investigated dynamic event traces of five Android apps and compared these against traces generated by eight users who tested the applications manually. Five different testing times for Monkey were used (5, 20, 25, 30, and 60 minutes), and the authors did not find a correlation between the testing time and Monkey's event coverage (such as UI events and lifecycle events, see [36]). Event coverage is a metric to identify how many of these events are triggered. The same held true for human testing of the same applications. Further, it was found that human testing and Monkey testing worked indistinguishably differently when it came to event coverage. From their testing they found that system events were the most generated events. Furthermore, Monkey most of the time generated a lower percentage of UI events compared to human testing. For 3 of the applications they investigated Monkey could mimic human testing behaviour. They concluded that Monkey works best when testing simple applications with a lot of operable widgets and no

required ordering of operations. Monkey does not work as well when testing complex applications that require the user to have domain knowledge.

From a study performed by [37] that compared automated exploration tools we can also see that coverage stops increasing by much at around the 5 minute mark. In their study they compared four different tools and found Monkey to be the second best performing when it came to method coverage, the best being TestLab [38].

A lot of studies have measured and compared the effectiveness of Monkey in coverage related metrics, but almost none test the different parameters of Monkey when performing their experiments. Most studies show that Monkey performs very well against other similar tools and remains among the most used.

Application benchmarks used are usually famous real world applications or application gathered from sources such as F-Droid. The amount of applications tested is usually around 50-100 applications, so this is comparable to the application benchmark used in this thesis.

Because multiple studies show that coverage stops increasing by much at around the 5 minute mark we test the applications in our experiments for 5 minutes.

3 Methods

3.1 Tools

This section explains components of the experiment pipeline in detail.

3.1.1 App Benchmark

In order to measure the code coverage and number of application crashes achieved by Monkey with various parameters, first a collection of APKs/applications is needed in order to be able to benchmark the performance of Monkey. A number of applications were collected from F-droid [39], a catalogue of open-source Android applications. After downloading some amount of applications from each category on F-droid during May and July 2021 and after removing applications that did not work properly with the selected tools or crashed when opened in the Android emulator, 50 applications were chosen from those that worked properly with the tools used for conducting the experiments and did not crash right away when opened in the Android emulator. A full list of the applications as well as some statistics about the applications used can be found in Appendix A. The applications are instrumented for coverage using ACVTool (see 3.1.4).

3.1.2 Emulator

The applications that are being tested by Monkey are installed on the emulator of the Android SDK. Experiments were performed on Ubuntu 20 operating system, with a 64-bit, 8 GB RAM machine having an Intel i7-8550U 4 core CPU. For every app, the emulator is reset to it's default state (all data wiped) to ensure that all applications have the same starting point and do not potentially influence each other. Exact specifications of the emulator can be found in Table 1.

Spec	**Value**
Device	Pixel 3a
Release Name	Oreo
API Level	26
Target	Android 8.0 (Google APIs)
CPU/ABI	x86

Table 1: Emulator specs used for performing experiments.

3.1.3 Monkey

Monkey has a number of parameters that can be chosen, changing the behaviour of the Monkey in various ways, such as the distribution of events or delay between events. A description of relevant parameters and events can be found in Table 2.

Parameter/Event	Description
seed	The random seed that determines the sequence of events.
throttle	Fixed delay between events. Defaults to sending events as fast as possible.
nav	Up/down/left/right inputs from a directional device
touch	Down-up event in a single place
majornav	Major events like center button or menu button
trackball	Random movements, can be followed by click
anyevent	Evens that do not fit in any other category
motion	Down event, random movements, up event
syskeys	Keys reserved for the system like Home or volume control
appswitch	Percentage of activity launches
pinchzoom	Pinch or zoom movements
rotation	Screen rotations

Table 2: Command line arguments for event types and frequencies [2].

Event	Percentage
nav	25.0%
touch	15.0%
majornav	15.0%
trackball	15.0%
anyevent	13.0%
motion	10.0%
syskeys	2.0%
appswitch	2.0%
pinchzoom	2.0%
rotation	0.0%

Table 3: Default distribution of Monkey events

3.1.4 ACVTool

ACVTool [25] can instrument Android APK files and generate a code coverage report, without needing access to the APK's original source code. ACVTool's workflow for one APK boils down to the following 6 steps:

1. Instrument
2. Install
3. Start
4. Test
5. Stop
6. Report

The steps are explained in detail below.

Instrument: ACVTool measures coverage by inserting probes after each `smali` instruction of the decompiled APK. Apktool [40] is used to decompile as well as reassemble the application, along with apksigner [41] and zipalign [42] to produce a working application that can be installed on a real device or emulator. An instrumented version of the APK is produced that can be used in the next step. An instrumentation report is also created which is used to map probes to their `smali` instructions.
Install: The instrumented version of the APK that was produced in the previous step is installed on the emulator or device, using `adb` [43], ready to be opened.
Start: The instrumentation process in the emulator is started, prompting the process of runtime information collection.
Test: During this step the application is tested and exercised, either manually or by a tool. In this case we use Monkey to generate the inputs and events on the application that is being tested.
Stop: The process of runtime information collection is halted and the information collected during the runtime is stored in a report on the device or emulator.
Report: During this phase, the runtime report and the instrumentation report are used to map probes back to their instructions, in order to generate the coverage report. The coverage report is available in both `xml` and `html` format, the former being more suited towards automated processing and the latter to visual inspection. Within the report, coverage on the class, method and instruction levels.

3.2 Experiment Pipeline

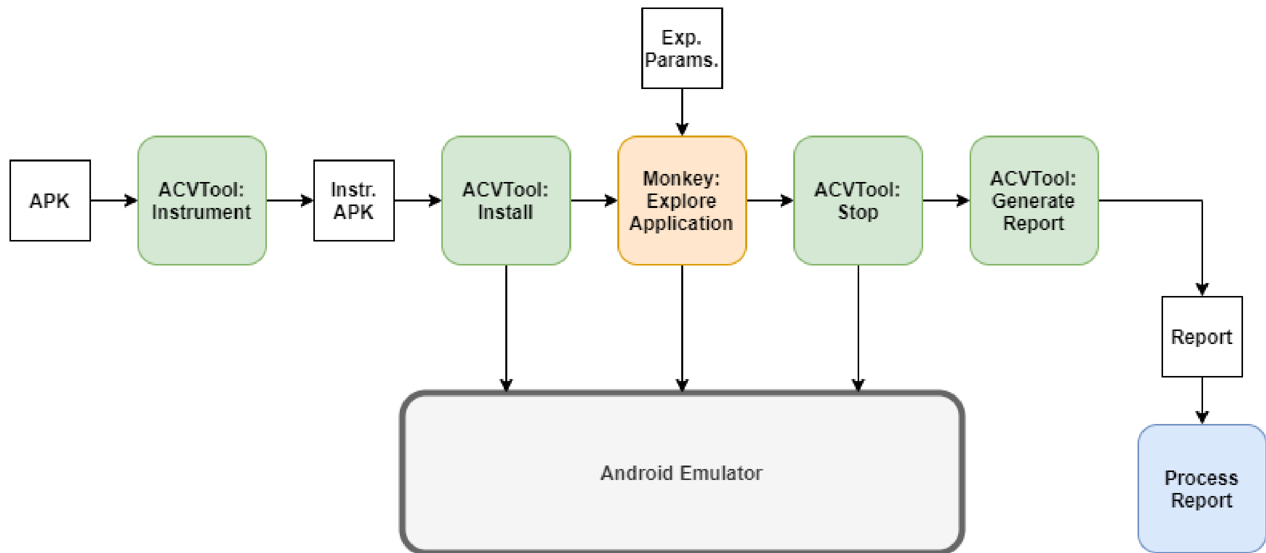

Figure 3: Experiment process for 1 APK
Green = ACVTool, Orange = Monkey, Blue = Custom report processing program

Figure 3 shows the structure of the experiment process for a single APK. This process is repeated for all 50 APKs in the application benchmark.

At the start, a fresh emulator is started that does not contain any prior data or state. Once the emulator has finished loading, the next step starts. First of all, an APK is selected from the application benchmark. ACVTool then proceeds to instrument this APK, producing an instrumented APK version. After this the instrumented APK is installed on the emulator. Following this, Monkey will explore the application according to the test parameters. After the time expires, Monkey stops and ACVTool stops the runtime information collection. Next, ACVTool generates the report which is then collected by the report processing program, see section 3.3. After the report is collected, the process starts again for the next APK.

3.3 Report Processing

After instrumentation, the `xml` coverage report generated by ACVTool after testing the application has to be parsed. ACVTool reports the coverage on the instruction, method and class levels. In order to extract coverage metrics from the `xml` report, we use the ElementTree XML API [44] from the Python standard library [45].

The `xml` report is structured as can be seen in Figure 4. At the root of the tree is the `report` element. Below the `report` node are all the `packages` of the application, below the `package` nodes are the `class` nodes. Contained in the `class` nodes are the `method` nodes. Summarizing, the order goes:
`report` → `package` → `class` → `method`.
In the `method` nodes, `counter` elements containing information about the amount of covered and missed instructions/methods are located. The `class` elements also contain `counter` elements about the overall covered and missed instructions/methods of that `class`.

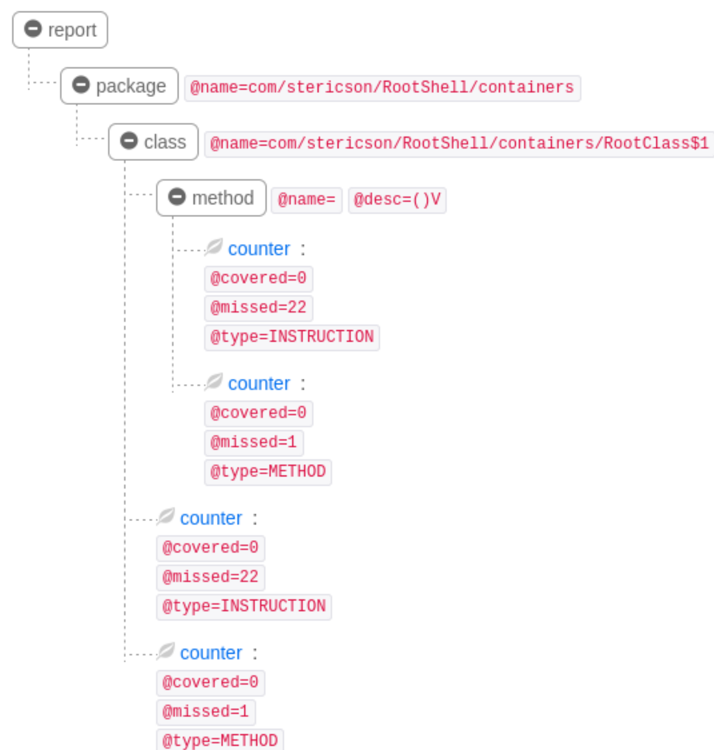

Figure 4: Example of the structure of an XML-report generated by ACVTool.

Exact coverage metrics determined from these `xml` reports are the percentage of covered lines, percentage of covered methods and the percentage of covered classes. The total amount of lines, methods and classes is also extracted.

Instruction and method coverage is calculated by summing up all the `counter` elements information, that are direct children of `class` nodes, and calculating the percentage of covered lines/methods by dividing the amount of lines by the total amount of lines/methods.

Class coverage is determined by checking if the `covered` element of the `method counter` node, that is a direct child of the `class` node is bigger than zero. The total amount of classes is then divided by the amount of covered classes to produce the percentage of covered classes.

All these calculated percentages along with the total amount of lines, method and classes are stored in a `csv` file along with the corresponding package name of the APK.

3.4 Tested Monkey Parameters

In order to better understand how much each parameter affects the measured code coverage, 10 different event distributions are tested and evaluated on the application benchmark. To compare these results, the default distribution of events is also tested at two different execution times.

First, the four most frequent events of the default distribution are tested at 100.0% for 5 minutes to see how much these events cover on their own. This produces the following 4 distributions.

1. 100.0% touch event
2. 100.0% nav event
3. 100.0% majornav event
4. 100.0% trackball event

Second, the 6 most frequent events of the default distribution are set to 50.0%, with the remaining 50.0% being divided to the other events according to the default distribution. Thus the following distributions are tested for 5 minutes each:

1. 50.0% touch event
2. 50.0% nav event
3. 50.0% majornav event
4. 50.0% trackball event
5. 50.0% anyevent event
6. 50.0% motion event

Lastly, the default distribution of Monkey's events is tested at **1 minute** of execution time and **5 minutes** of execution time.

4 Results

4.1 Code Coverage

In this section, the results of the previously described experiments to determine the coverage and number of crashes are presented.

After instrumenting the applications in the benchmark using ACVTool, each instrumented application is installed on a fresh emulator and tested for 5 minutes (except for the 1 minute default event distribution) by Monkey with the specified event distribution. Afterwards the coverage report is generated by ACVTool and the average coverage metrics over the application benchmark are calculated. It is also tracked whether the application crashed during the testing process.

Figure 5 shows the average instruction, method and class coverage achieved by the different event distributions. Tables 4, 5 and 6 display the average instruction, method and class coverage per event distribution ordered from highest to lowest.

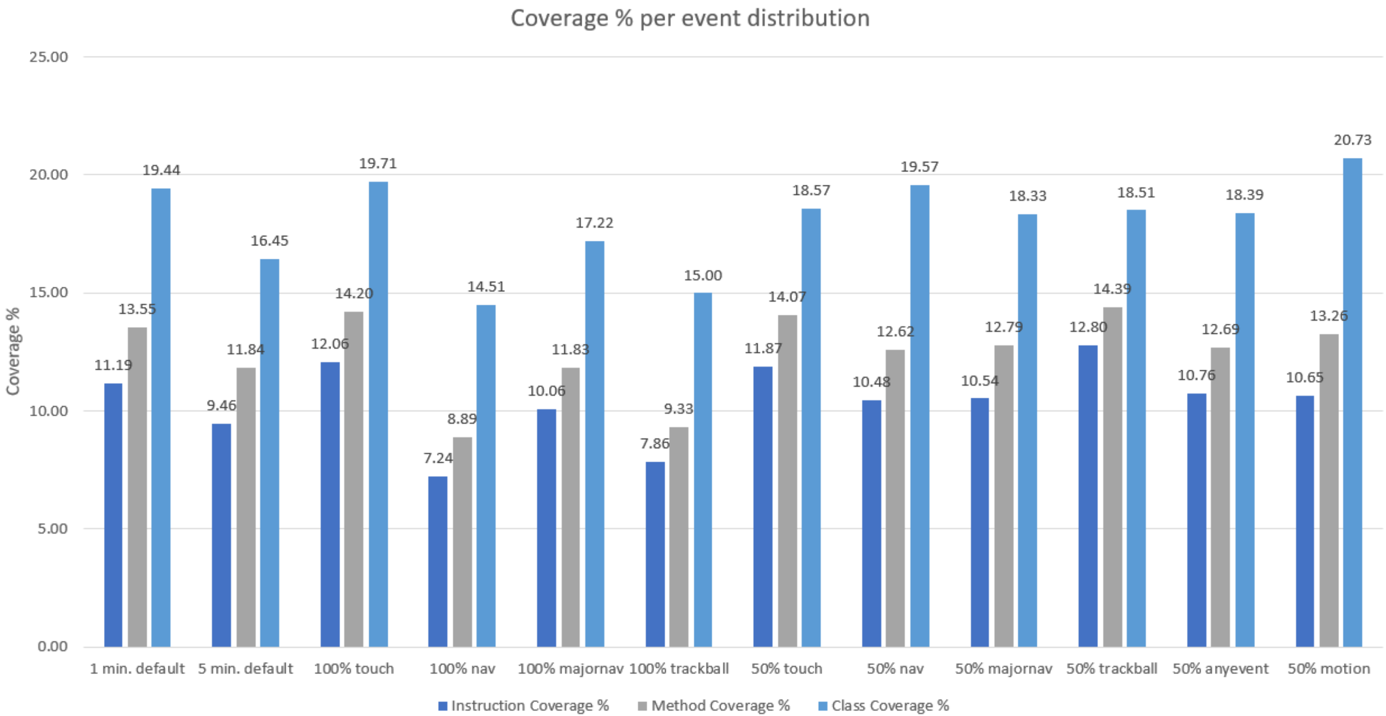

Figure 5: Average instruction, method and class coverage per event distribution in %, each tested for 5 minutes per application, except for 1 minute default.

Parameter	Instruction Coverage %
50% trackball	12.80
100% touch	12.06
50% touch	11.87
1 min. default	11.19
50% anyevent	10.76
50% motion	10.65
50% majornav	10.54
50% nav	10.48
100% majornav	10.06
5 min. default	9.46
100% trackball	7.86
100% nav	7.24

Table 4: Instruction coverage ordered from highest to lowest

Parameter	Method Coverage %
50% trackball	14.39
100% touch	14.20
50% touch	14.07
1 min. default	13.55
50% motion	13.26
50% majornav	12.79
50% anyevent	12.69
50% nav	12.62
5 min. default	11.84
100% majornav	11.83
100% trackball	9.33
100% nav	8.89

Table 5: Method coverage ordered from highest to lowest

Parameter	Class Coverage %
50% motion	20.73
100% touch	19.71
50% nav	19.57
1 min. default	19.44
50% touch	18.57
50% trackball	18.51
50% anyevent	18.39
50% majornav	18.33
100% majornav	17.22
5 min. default	16.45
100% trackball	15.00
100% nav	14.51

Table 6: Class coverage ordered from highest to lowest

4.2 Crashes

While testing the applications from the benchmark, the amount of crashes was also recorded, as visible in Figure 6. In Table 7, the number of crashes ordered from high to low can be seen. A crash in this case is defined as an exception that does not get handled and causes the application to exit. This means the number of crashes can be seen as the amount of applications that were unable to continue working while being tested by Monkey during the experiment's allocated time. Discussion about the results is found in the next section.

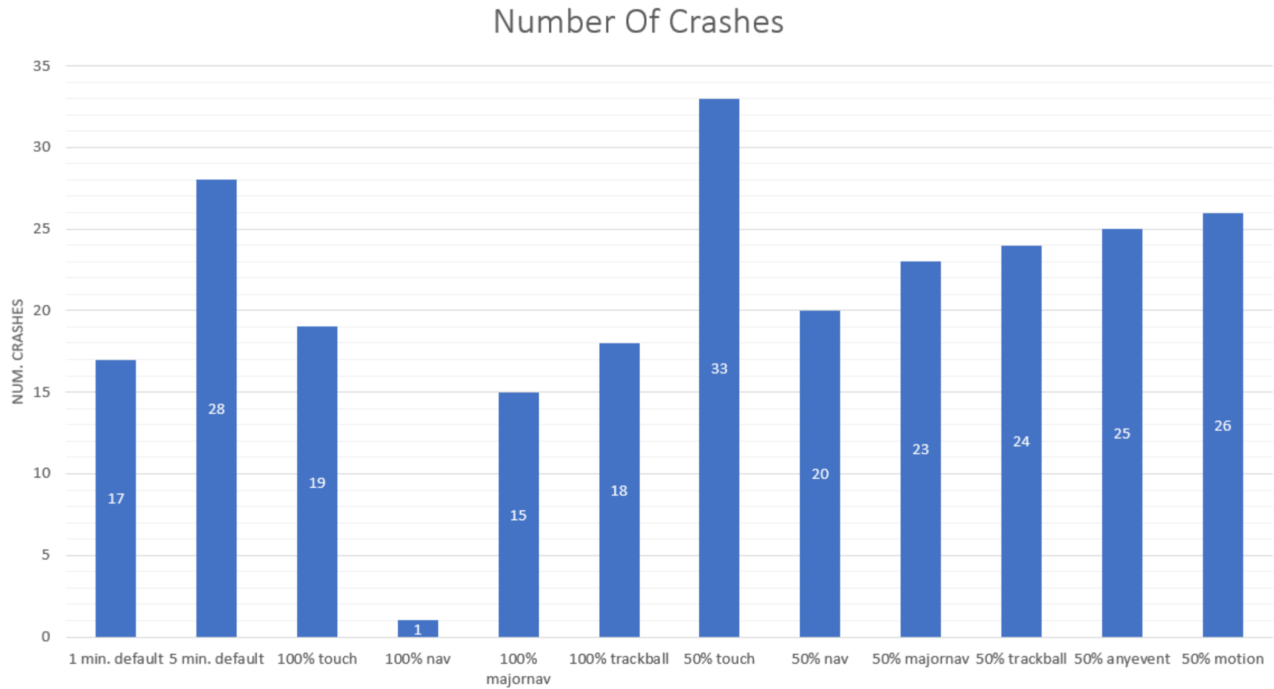

Figure 6: Amount of crashes per event distribution

Parameter	**Num. Crashes**
50% touch	33
5 min. default	28
50% motion	26
50% anyevent	25
50% trackball	24
50% majornav	23
50% nav	20
100% touch	19
100% trackball	18
1 min. default	17
100% majornav	15
100% nav	1

Table 7: Amount of crashes ordered from high to low

5 Discussion

In this section, the results presented in the previous section are discussed and explained.

5.1 Code Coverage

From the code coverage metrics in Figure 5 and Tables 4, 5 and 6 we can see that the default event distribution of Monkey achieves an instruction coverage of 9.46%, a method coverage of 11.84% and a class coverage of 16.45%. Compared to these numbers from the 5 minute default event distribution, only the 100% trackball and 100% nav event distributions perform worse. Interestingly, the 1 minute default distribution does better than the 5 minute default distribution, the explanation for this is that due to Monkey's random nature, different applications crashed at different moments causing the average code coverage to go down. ACVTool also does not generate an intermediate report after the application crashes which can cause the 1 minute distribution to do better in code coverage than the 5 minute distribution.

The best performing event distribution when it came to the instruction coverage metric was the 50% trackball distribution. For method coverage 50% trackball was also the best performing. When it came to class coverage, 50% motion did the best.

Between instruction coverage and method coverage, the order of best performing remains relatively similar. However when it comes to class coverage, the order is changed somewhat. This indicates that certain event distributions can cover more classes than others.

From the data we can see that changing the event distribution can have a positive effect on the code coverage achieved by Monkey. Although the effect is not incredibly large, compared to 5 minutes of the default settings, the biggest positive difference in instruction coverage was about 3.3%. But compared to 1 minute of default settings the difference was only 1.6%. These differences are about the same when comparing instruction coverage to method coverage and class coverage. An overall instruction coverage of 12.8% (the highest achieved from the tested event distributions) is still not very high. Exploring only about 12.8% of an application is definitely not enough to rely on automated testing for testing an entire application. Adding to that, from a security standpoint it also leaves way too much room for uncertainty regarding what the code of the application does that was not covered.

Contrary to the study by Patel et al. [33] which tested the code coverage achieved by changing 5 of Monkey's parameters to 75% event distribution and did not find it to lead to an average coverage increase, the event distributions that were tested in this thesis *did* lead to an increase in coverage. This means that altering the parameters of Monkey can lead to an increase in average code coverage on a wider range of applications and not just in isolated cases as stated by Patel et al.

5.2 Crashes

Figure 6 and Table 7 show the amount of applications that crashed per event distribution, with the table showing the same data as the figure but ordered from high to low. The default Monkey event distribution at 5 minutes of runtime crashed 28 out of the 50 applications, over half of them.

The event distribution that crashed the most applications was 50% touch, with 33 crashes. 100% nav did the worst, with 1 application crashed. 100% nav also did the worst when it came to code coverage, which could explain why it crashed so little applications compared to the other distributions.

The 5 minute runtime default event distribution of Monkey crashed a lot more applications than the 1 minute version, 28 compared to 17. The 5 minute runtime did however achieve a lower code coverage than the 1 minute runtime, which could be due to the amount of crashes.

From Table 7 we can see that changing the event distribution of Monkey can increase the amount of crashes, but the default distribution already does very well in this regard. This means that Monkey is very effective at crashing applications, making it useful for stress testing applications.

In line with [34] and [33], the experiments performed in this thesis also show that Monkey is very good at crashing applications. The default distribution crashed more than half of the applications in the benchmark with the 50% touch doing even better than that. Monkey could thus be very useful in helping developers save time by automatically having their applications stress tested.

5.3 Limitations

There are some limitations that came with testing the effectiveness of the different Monkey event distributions on the application benchmark. First of all, it takes a lot of time to test applications for 5 minutes each, it does not simply take 5 minutes to test one application, there is also added overhead from opening/wiping the emulator, installing applications, etc. If more time and resources would have been available, the application benchmark could have been larger, allowing for more reliable data to be gathered.

The amount of different distributions tested could also have been larger if more time and resources were available. Testing more event distributions could give more insight into how the different events affect code coverage and crashes.

6 Conclusions and Further Research

In this thesis, a study was conducted on the effectiveness of the automated testing tool Monkey when comparing different parameters and event distributions. Experiments were performed on an application benchmark consisting of 50 applications and testing each for 5 minutes. It was found that changing the event distribution of Monkey can lead to an increase in code coverage as well as an increase in crashes when testing applications. The 50% trackball event distribution achieved the highest code coverage of 12.80%, with the 50% touch distribution crashing the most applications, 33 in total. Changing the event distribution can have a positive influence in achieving a higher average code coverage on a large amount of applications. This shows that automated testing still has a long way to go when it comes to code coverage. In a black-box setting covering only 12.80% of instructions is not enough to make informed decisions about the nature of an application in a security context. This number is also too low to give developers confidence the entirety of their application is being tested. However, the high amount of crashes that Monkey triggers proves it's usefulness in stress testing applications and finding potential faults.

In this thesis 10 event distributions of Monkey were tested that were not the default distribution on 50 applications. If more time and resources could be used, more event distributions and parameters of Monkey could be evaluated on a larger dataset of applications. Since the difference in code coverage rates is not very large, it is hard to say what to use in which situation. As future work, the scale of the experiments could be made way larger to gain more data on the effect of the event distributions as well as gain more trust in the produced data.

References

[1] Mobile operating systems' market share worldwide from january 2012 to june 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. Accessed: July 2021.

[2] Android. Exerciser monkey. https://developer.android.com/studio/test/monkey. Accessed: July 2021.

[3] Number of smartphone subscriptions worldwide from 2016 to 2026. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Accessed: July 2021.

[4] Google play. https://play.google.com/. Accessed: July 2021.

[5] Number of android apps on google play. https://www.appbrain.com/stats/number-of-android-apps. Accessed: July 2021.

[6] Google play statistics and trends. https://42matters.com/google-play-statistics-and-trends. Accessed: July 2021.

[7] Global app spending approached $65 billion in the first half of 2021, up more than 24% year-over-year. https://sensortower.com/blog/app-revenue-and-downloads-1h-2021. Accessed: July 2021.

[8] Open handset alliance. https://www.openhandsetalliance.com/. Accessed: July 2021.

[9] Android open source project. https://source.android.com/. Accessed: July 2021.

[10] .apk file extension. https://fileinfo.com/extension/apk. Accessed: July 2021.

[11] Application fundementals. https://developer.android.com/guide/components/fundamentals. Accessed: July 2021.

[12] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, 2013.

[13] Abhinav Kathuria1 and Anu Gupta. Challenges in android application development: A case study. In *International Journal of Computer Science and Mobile Computing*, volume 4, pages 294–299, 2015.

[14] Naila Kousar, Muhammad Sheraz, Arshad Malik, Aramghan Sarwar, Burhan Mohy-ud din, and Ayesha Shahid. Software engineering: Challenges and their solution in mobile app development. *International Journal of Advanced Computer Science and Applications*, 9(1):200–203, 2018.

[15] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, PP:1–22, 09 2018.

[16] Diego R. Almeida, Patrícia D. L. Machado, and Wilkerson L. Andrade. Testing tools for android context-aware applications: a systematic mapping. *Journal of the Brazilian Computer Society*, 25(1):12, Dec 2019.

[17] Mario Linares-Vásquez, Cárlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, 2017.

[18] Isabel K. Villanes, Silvia M. Ascate, Josias Gomes, and Arilo Claudio Dias-Neto. What are software engineers asking about android testing on stack overflow? In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, page 104–113, New York, NY, USA, 2017. Association for Computing Machinery.

[19] 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, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[20] 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, page 245–256, New York, NY, USA, 2017. Association for Computing Machinery.

[21] 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, page 987–992, New York, NY, USA, 2016. Association for Computing Machinery.

[22] 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, page 224–234, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[24] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 738–748, 2018.

[25] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. Fine-grained code coverage measurement in automated black-box android testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–35, 2020.

[26] Andrea Romdhana, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella. Cosmo: Code coverage made easier for android. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 417–423, 2021.

[27] ELLA, a tool for binary instrumentation of android apps. https://github.com/saswatanand/ella. Accessed: July 2021.

[28] 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 (ARES)*, pages 501–510, August 2015.

[29] Jierui Liu, Tianyong Wu, Xi Deng, Jun Yan, and Jian 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, 2017.

[30] Grade build tool. https://gradle.org/. Accessed: July 2021.

[31] JaCoCo java code coverage library. https://www.eclemma.org/jacoco/. Accessed: July 2021.

[32] V. Rubtsov. EMMA: A free java code coverage tool. http://emma.sourceforge.net/. Accessed: July 2021.

[33] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. On the effectiveness of random testing for android: Or how i learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test*, AST '18, page 34–37, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Samad Paydar. An empirical study on the effectiveness of monkey testing for android applications. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, 44(2):1013–1029, Jun 2020.

[35] Mostafa Mohammed, Haipeng Cai, and Na Meng. An empirical comparison between monkey testing and human testing (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, page 188–192, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Input events overview. https://developer.android.com/guide/topics/ui/ui-events. Accessed: July 2021.

[37] Michael Osoro-Riano and Mario Linares-Vasquez. Comparison and analysis between automatic exploration tools for android applications. 2020.

[38] Firebase test lab. https://firebase.google.com/docs/test-lab. Accessed: July 2021.

[39] F-droid. https://www.f-droid.org/. Accessed: July 2021.

[40] apktool. https://ibotpeaches.github.io/Apktool/. Accessed: July 2021.

[41] apksigner. https://developer.android.com/studio/command-line/apksigner. Accessed: July 2021.

[42] zipalign. https://developer.android.com/studio/command-line/zipalign. Accessed: July 2021.

[43] Android debug bridge (adb). https://developer.android.com/studio/command-line/adb. Accessed: July 2021.

[44] Elementtree xml api. https://docs.python.org/3/library/xml.etree.elementtree.html. Accessed: July 2021.

[45] Python standard library. https://docs.python.org/3/library/. Accessed: July 2021.

Appendix

A Application Benchmark

Package Name	File Size (MB)
com.smartpack.kernelprofiler	1.8
com.metallic.chiaki	8.4
dummydomain.yetanothercallblocker	8.0
com.allansimon.verbisteandroid	11.0
net.moasdawiki.app	1.8
com.fgrim.msnake	0.1
org.strawberryforum.pollywog	1.3
de.markusfisch.android.shadereditor	1.6
me.blog.korn123.easydiary	21.2
eu.veldsoft.ithaka.board.game	4.2
com.tnibler.cryptocam	6.1
com.nicobrailo.pianoli	3.6
org.ninthfloor.copperpdf	0.4
org.woheller69.weather	11.5
com.smartpack.smartflasher	2.0
digital.selfdefense.lucia	2.5
fr.ralala.hexviewer	1.7
de.jbservices.nc_passwords_app	27.5
com.maxistar.textpad	1.5
org.getdisconnected.libreipsum	2.1
com.DartChecker	2.5
jp.takke.cpustats	2.1
taco.scoop	1.6
de.k3b.android.intentintercept	1.2
de.msal.muzei.nationalgeographic	3.2
com.concept1tech.instalate	3.6
com.releasestandard.scriptmanager	3.4
ca.cmetcalfe.locationshare	0.9
org.segin.bfinterpreter	1.3
de.bulling.barcodebuddyscanner	3.7
com.appmindlab.nano	3.4
com.tunjid.fingergestures	2.5
com.aragaer.jtt	0.2
com.github.ashutoshgngwr.tenbitclockwidget	1.1
com.zorinos.zorin_connect	6.4
dev.corruptedark.openchaoschess	1.1
org.epstudios.morbidmeter	1.7
ch.hgdev.toposuite	2.4
com.dosse.airpods	1.4
com.dozingcatsoftware.kumquats	19.0
com.chessclock.android	0.1
de.eidottermihi.raspicheck	3.8
com.sunilpaulmathew.debloater	1.9
net.tjado.usbgadget	2.2
com.adonai.manman	2.9
com.nathaniel.motus.umlclasseditor	2.7
net.sourceforge.x11basic	3.1
de.dennisguse.opentracks	21.6
com.smartpack.scriptmanager	1.8
io.github.lufte.lona	1.9

Table 8: Application benchmark used in experiments

Average Size	4.5
Median Size	2.3
Min Size	0.1
Max Size	27.5
Standard Deviation	5.9

Table 9: Statistics about the sizes in MB of the application benchmark