PhD-FSTM-2020-64
The Faculty of Sciences, Technology and Medicine

DISSERTATION

Defence held on 06/11/2020 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Aleksandr PILGUN
Born on 18 March 1990 in Minsk (Belarus)

INSTRUCTION COVERAGE FOR ANDROID APP TESTING AND TUNING

Dissertation defence committee

Dr Yves Le Traon, Chair
Professor, Université du Luxembourg

Dr Olga Gadyatskaya, Vice-Chair
Leiden University, Netherlands

Dr Sjouke Mauw, dissertation supervisor
Professor, Université du Luxembourg

Dr Pascal Bouvry
Professor, Université du Luxembourg

Dr Yang Liu
Professor, Nanyang Technological University, Singapore

PhD-FSTM-2020-64
The Faculty of Science, Technology and Medicine

DISSERTATION

Presented on 06/11/2020 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Aleksandr PILGUN
Born on 18 March 1990 in Minsk (Belarus)

INSTRUCTION COVERAGE FOR ANDROID APP TESTING AND TUNING

Supported by the Luxembourg National Research Fund (FNR)
AFR-PhD-11289380-DroidMod.

Abstract

For many people, mobile apps have already become an indispensable part of modern life. Apps entertain, educate, assist us in our daily routines and help us connect with others. However, the advanced capabilities of modern devices running the apps and sensitive user data make mobile devices also an attractive attack target. To get access to sensitive data, adversaries tend to conceal malicious functionality in freely distributed legitimately-looking apps.

The problem of low-quality and malicious apps, spreading at an enormous scale, is especially relevant for one of the biggest software repositories — Google Play. The Android apps distributed through this platform undergo a validation process by Google. However, that is insufficient to confirm their good nature. To identify dangerous apps, novel frameworks for testing and app analysis are being developed by the Android community.

Code coverage is one of the most common metrics for evaluating the effectiveness of these frameworks, and it is used as an internal metric to guide code exploration in some of them. However, when analyzing apps without source code, the Android community relies mostly on method coverage since there are no reliable tools for measuring finer-grained code coverage in 3rd-party Android app testing.

Another stumbling block for testing frameworks is the inability to test an app exhaustively. While code coverage measurement can indicate an improvement in testing, it is neither possible to reach 100% coverage nor to identify the maximum reachable coverage value for the app. Despite testing, the app still contains high amounts of not executed code, which makes it impossible to confirm the absence of potentially malicious code in the part of the app that has not been tested. The existing static debloating approaches aim at app size minimization rather than security and simply debloat not reachable code. However, there is currently no approach to debloat apps based on dynamic analysis information, i.e. to cut out not-executed code.

In this dissertation, we solve these two problems by, first, proposing an efficient approach and a tool to measure code coverage at the instruction level, and second, a dynamic binary shrinking methodology for deleting not executed code from the app. We support our solutions by the following contributions:

- An instrumentation approach to measure code coverage at the instruction level. Our technique instruments `smali` representation of Android bytecode to allow code coverage measurement at the finest level.
- An implementation of the instrumentation approach. ACVTool is a self-contained package containing 4K lines of Python code. It is publicly available and can be integrated into different testing frameworks.
- An extensive empirical evaluation that shows the high reliability and versatility of our approach. ACVTool successfully executes on 96.9% of apps from our dataset, introduces a negligible instrumentation time and runtime overheads, and its results are complaint to the results of JaCoCo (source code coverage) and Ella (method coverage) tools.
- A detailed study on the influence of code coverage metric granularity on automated testing. We demonstrate the usefulness of ACVTool for automated testing techniques that rely on code coverage data in their operation.
- A dynamic debloating approach based on ACVTool instruction coverage. We propose Dynamic Binary Shrinking System, a novel methodology created to shrink 3rd-party Android apps towards observed benign functionality on executed code.
- An implementation of the dynamic debloating technique incorporated into the ACVCut tool. The tool demonstrates the viability of the Dynamic Shrinking System on two examples. It allows us to cut out not executed code and, thus, provide 100% instruction coverage on explored app behaviors.

Acknowledgments

First of all, I would like to thank Prof. Dr. Sjouke Mauw for letting me join the SaToSS research group, his sincere and fair comments, and overall positive influence on me. I especially appreciate giving me the opportunity to independently conclude my research with the final paper.

This journey, however, would not have been possible without a great support of Dr. Olga Gadyatskaya and Dr. Yury Zhauniarovich, whose research experience from one hand and the knowledge of the Android platform on the other hand were a great boost to me. I am sincere grateful to them also for arranging my coming to Luxembourg, being responsible and always ready to pass me the hand of support.

Thanks to the members of my defence committee, Prof. Dr. Yves Le Traon, Prof. Dr. Pascal Bouvry, and Prof. Dr. Yang Liu for reviewing my thesis and joining the defence committee.

At the TAROT summer school, I had a great luck to meet Dr. Manuel Rigger, who reflects me a role model of a classic researcher. Thanks to him I ended up at a Google private event, where I met other world top Software Engineering PhD researchers and received inspiration for the latest work.

Thanks to my close friends, colleagues and especially PhDs, who were creating an atmosphere of fraternity and mutual aid. Thanks in particular to Jorge and Zach – for being bright examples of PhD researchers. It was fun to share the Kirchberg office with you. Sevdenur, Sema & Dasha – for mindful conversations and unconditional support. Zhiqiang – for being extra positive. Alena and her beautiful cats (Busya & Zefirka) – for several wonderful years of life-changing experience. Thanks to my parents and my sister for their unconditional love and support.

Thanks to my talented students, who trusted me to guide them in their bachelor and master projects: Pavel, Rudrani, Davit and Roman. I wish them a good future.

Finally, I would like to express my gratitude and full support to the people of Belarus who are currently changing my country to a rightful and prosperous place for everyone. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the reader into the subject of this dissertation. We discuss the role of code coverage measurement in automated testing of Android apps and the possibility to shrink apps basing on the code coverage information. We formulate research questions to be answered and list our contributions to this dissertation.

Contents

1.1 Android Apps

In the last ten years, the software development community supported by the hardware manufacturers brought up a new generation of software — applications (apps for short) for mobile devices. The mobile apps quickly became a permanent part of modern life, wherein people not only communicate and entertain themselves but also perform their work tasks from mobile devices [LL19]. Not surprisingly, mobile traffic has more than tripled since 2013, while from the beginning of 2017 mobile devices consistently generate around 50% of web traffic [Cle20a].

Nowadays, an average user spends a significant amount of time during the day with a smartphone in his hand. The popularity of devices also changed the way traditional retail companies work. People buy products on the fly, consume virtual content (including advertisements) and buy virtual services using mobile devices. Companies actively integrate new technologies such as NFC, GPS and geofencing to improve their services.

Currently, Android and iOS are the main competitive operating systems on the mobile market. While iOS accounts for 14% of the market, Android takes remaining 86% [IDC20]. Both mobile systems have central market places for apps — App Store and Google Play, respectively, where any app developer can publish their app. To develop an app, the developer relies on sophisticated documentation, tools and SDKs explicitly created for these operating systems. In the Android OS, moreover, a user can benefit from alternative app markets such as Amazon Appstore and F-Droid and even install an app just downloaded from the Internet.

No doubt, the apps that daily help people, make mobile devices so important for our society. Google Play is filled with millions of apps that billions of people use every day. Shortly after its release, Android has gained popularity and a large user base. However, it has also quickly become a target for attackers.

Anyone can create an Android app and publish it on Google Play, but Google that controls this marketplace has no access to source code of submitted apps and no control over the development process. Considering the huge number of submitted apps, it is very challenging for Google to comprehensively analyse and completely eliminate malicious third-party applications. A famous example of Android malware spread through Google Play is one of the largest mobile botnet campaigns called NotCompatible.C [Str14]. It comprised the possibility to run spam campaigns, bulk ticket purchasing, bruteforce attacks on Wordpress and C99 shell control. The malware infected over four million Android devices. There are and have been many more malware families targeting Android, as discussed by Tam et al. [TFA+17].

Still, Google is making a serious effort to protect the market from low

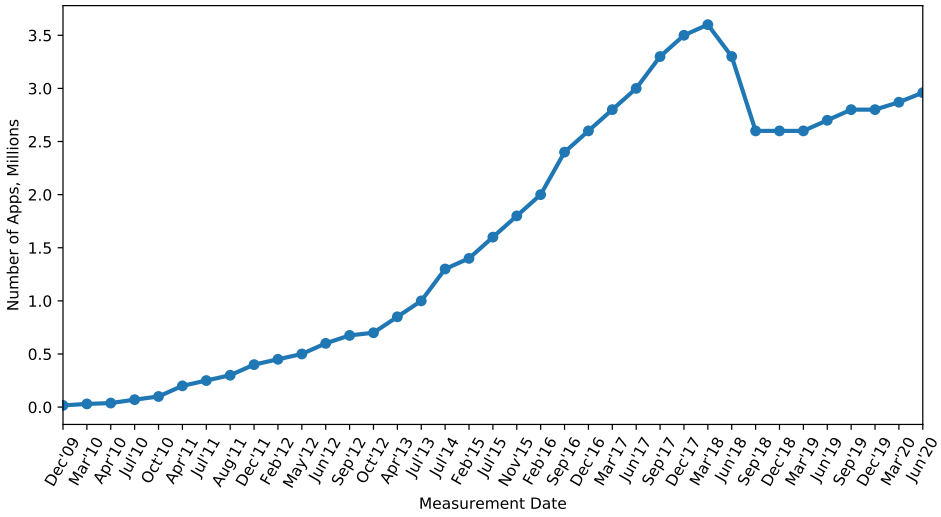


Figure 1.1: Number of apps on Google Play in 2009-2020 (Statista [Cle20b]).

quality and malicious apps. Figure 1.1 demonstrates the number of uploaded apps since the opening of Google Play [Cle20b]. We could see stable growth of Google Play followed by the unexpected drop in 2018. While the Google filtering mechanisms are not always transparent, Wang et al. report on the main reasons for app removal [WLL+18]. The deleted apps include apps demonstrating high privacy risks, fake apps, spamming apps, malware and ad-blocking apps.

Another significant achievement that Google recently fulfilled in terms of Android trustworthiness is the certification of Google Pixel 3/3XL devices as compliant to Common Criteria[1]. Certification allowed strict-policy enterprises and government institutions to use Google phones trustfully. Besides the devices, the validation report certified Android 9.0, and mentioned new libraries and rich APIs for app development.

However, 3rd-party Android apps are not the target of evaluation in this certification. To regulate potentially malicious apps, the enterprise version of Android introduced a Mobile Device Management (MDM) system capable of remote control of apps and phone settings [Goo20a]. Still, vulnerabilities and developed attacks continue to threaten the Android ecosystem [LVBEV17, TFA+17, FBR+16]. Despite all the security improvements of the Android system, Google and external researchers continue to investigate techniques for ensuring security and reliability of 3rd-party Android apps.

Indeed, the Android community continuously studies new techniques to analyze such apps. Novel frameworks for automated testing and analysis of 3rd-party apps are being developed by the community. Many of these frame-

[1]https://www.niap-ccevs.org/Product/Compliant.cfm?PID=10941

works are being evaluated based on the code coverage achieved in testing [ANKS16, ZPG+15]. Novel frameworks for automated testing of 3rd-party apps have been developed, where code coverage is one of the most common metrics for evaluating the effectiveness of these frameworks [ANKS16, ZPG+15] or use code coverage as part of a fitness function for guiding evolutionary, mutation and fuzzy testing techniques [KLG+18, MMM14, MHJ16, WDS+19, WJL+20].

1.2 Code Coverage Measurement in Automated Testing

Code coverage measurement is an essential element of software development and quality assurance cycles for all programming languages and ecosystems, including Android. It is routinely applied by developers, testers, and analysts to understand the degree to which the system under test has been evaluated [AO16], to generate test cases [YLW09], to compare test suites [GGZ+15], and to maximize fault detection by prioritizing test cases [YH12]. In the context of Android application analysis, code coverage has become a critical metric. Fellow researchers and practitioners evaluate the effectiveness of tools for automated testing [CGO15, MHJ16, KLG+18, WLY+18] and security analysis [MEK+12, HCLT15] using code coverage, among other metrics. It is also used as a fitness function to guide application exploration in testing [MHJ16, SMC+17, KSM+18]. However, there are no reliable tools for measuring fine-grained code coverage in 3rd-party Android app testing.

Unfortunately, the Android ecosystem introduces a particular challenge for security and reliability analysis: Android apps submitted to markets (e.g., Google Play) have been already compiled and packaged, and their source code is often unavailable for inspection. Measuring code coverage achieved in testing and analysis is not a trivial endeavor in this setting. This is why some third-party app testing systems, e.g., [SJM17, CNS13, MHH+19], use open-source apps for experimental validation, whereby the source code coverage could be measured by popular tools developed for Java, such as EMMA [Rub06] or JaCoCo [JaC18]. These, and other systems will benefit from a reliable tool for measuring code coverage in testing third-party Android apps. In this thesis, we aim to close this gap with measuring fine-grained code coverage in third-party app testing. To achieve this and to propose a tool that will be a useful addition to the Android app testing and analysis ecosystem, we will address the following research question.

Research question 1: *How to measure instruction coverage when testing Android apps?*

In the absence of source code, code coverage is usually measured by instrumenting the bytecode of applications [LMOD13]. 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 [THB+16, LMOD13]. 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 [HCLT15], there have been so far only few attempts to track code coverage for Android applications at the bytecode level [ZLZ+16], and they all still have limitations. The most significant one is the *coarse granularity* of the provided code coverage metric. For example, ELLA [ELL16], InsDal [LWD+17] and CovDroid [YH15] measure code coverage only at the method level. Another limitation of the existing tools is the *low percentage* of successfully instrumented apps. For instance, the tools by Huang et al. [HCLT15] and Zhauniarovich et al. [ZPG+15] 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., [LWD+17, CR17a, YH15]). Thus, it is important not only to unlock the new approach to measure code coverage as we stated in RQ1 but also it is necessary to evaluate the impact of the chosen approach on the app functioning and confirm its non-prohibitive nature for testing purposes. We will address this goal with the next research question.

Research question 2: *How reliable and versatile is the developed instrumentation approach?*

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., [MHJ16, SQH17, CR17a, MHH+19, LR19]. 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. Since the various automated testing frameworks use method coverage provided by Ella or their own implementation for code coverage, when testing apps without source code, the following

question appears.

Having a reliable and efficient approach to measure fine-grained code coverage will benefit many automated testing and analysis tools by proving them a way to benchmark their achieved coverage in a straightforward way. Developers of tools like Sapienz [MHJ16] and Stoat [SMC+17] will be able to use the same code coverage metrics on third-party apps and in this way establish the advantages of their innovations. It is still an open question, however, whether usage of some particular code coverage metrics within the fitness functions used in fuzzy or evolutionary testing will have a positive effect on the testing process quality. In this thesis we aim to make the first step in investigating this question by answering the following research question.

Research question 3: *What is the impact of the granularity of the code coverage metric on the quality of the testing process?*

The novel fine-grained code coverage tool is to be readily used with various dynamic analysis and automated testing tools, e.g., IntelliDroid [WL16], CopperDroid [TKFC15], Sapienz [MHJ16], Stoat [SMC+17], DynoDroid [MTN13], CuriousDroid [CML+17], PATDroid [SJM17], Paladin [MHH+19], to mention a few, to measure code coverage.

1.3 Dynamic Binary Shrinking

The security measures such as Google Play Protect, Mobile Device Management system (appeared in the enterprise version of Android), security improvements on Android APIs permissions that Google constantly introduce in Android drive the system to be more secure. However, attacks also evolved into more sophisticated and targeted [TFA+17, FBR+16].

Logic bombs are a great example of a popular targeted attack. A logic bomb-driven attack may target people with certain powers or secrets up to state-sponsored attacks [FBR+16]. Another example of a logic bomb is cryptojacking. Recently, in the wake of an increased interest in cryptocurrencies, Android markets have seen the rise of cryptojacking — apps mine cryptocurrencies secretly from the device owner. Remarkably, attackers more actively used devices at full capacity during the owner's inactive hours and when charging [DZG+20].

Researchers and practitioners developed many approaches for analyzing Android apps as we have mentioned earlier. However, the problem of reliably detecting security issues or crashes in unknown 3rd party apps is still an open one.

Currently, sandboxing [JZ16, BLL18] is a solution on the frontier to restrict not observed functionality. Still, this approach continues evolving, e.g.

Le et al. suggested to enhance original sensitive APIs restriction by limiting also parameter values in the API calls [LBL+18], which, however, misses malicious behaviors in at least 5% of apps in their experiments. Android apps need a simple, ready-to-go solution to verify existing behaviors, restrict not observed potentially malicious ones, and give a guarantee with a simple metric. A more radical path we propose in this work is the removal of not tested code from the app. This raises the next research question.

Research question 4: *Is it possible to eliminate not-executed code to achieve full code coverage?*

We present a novel approach, called Dynamic Binary Shrinking. It allows to monitor app behavior while exploring the app and eventually cut it towards the tested benign code. We incorporated our shrinking technique into the ACVCut tool, that is based on the ACVTool instruction coverage. Thus, we cut the app towards the executed code, and the shrunk app will, therefore, achieve 100% instruction coverage on the tested functionality. Comparing to the sandboxing, our approach limits the app functionality in a more radical way — by removing not executed (potentially malicious) code.

Yet, the shrunk app loses not explored features including not explored hidden functionality such as logic bombs. With respect to the observed and acceptable behaviors, the app stays fully functioning.

1.4 Contributions

We summarize the contributions of this dissertation as follows:

- **An instrumentation approach to measure coverage of 3rd-party Android apps.** We propose a novel approach to track execution of app instructions by inserting probes into Android app that does not have source code. The technique works on the `smali` representation of the Dalvik bytecode and allows to track execution of app instruction that helps us to achieve the finest granularity of a coverage metric for Android apps. Our approach is fully self-contained and transparent to the testing environment. The proposed approach answers the **Research Question 1** by proposing a fine-grained coverage measurement approach for Android apps.
- **An implementation of the instrumentation approach in ACVTool.** The tool can be integrated with any testing or dynamic analysis framework. Our tool presents the coverage measurements and information about encountered crashes as handy reports that can be either

visually inspected by an analyst, or processed by an automated testing environment. It consists of 4K lines of Python code.

- **An extensive empirical evaluation that shows the high reliability and versatility of our approach.**
  - While previous works [HCLT15, ZPG+15] have only reported the number of successfully instrumented apps[2], we also verified whether apps can be successfully executed after instrumentation. We report that **96.9%** have been successfully executed on the Android emulator, which is only 0.9% less than the initial set of successfully instrumented apps.
  - In the context of automated and manual application testing, ACVTool introduces only a **negligible instrumentation time overhead**. In our experiments ACVTool required on average 33.3 seconds to instrument an app.
  - The **runtime overhead introduced by ACVTool is very low** for real apps: in experiments with real Android apps the mean CPU overhead introduced by ACVTool is 0.53%.
  - We have evaluated whether ACVTool reliably measures the bytecode coverage by comparing its results with those reported by JaCoCo [JaC18] and ELLA [ELL16]. Our results show that the ACVTool results can be **trusted**, as code coverage statistics reported by ACVTool, JaCoCo and ELLA are highly correlated.

  This study answers the **Research Question 2** by concluding the compliance of our instruction coverage to the existing code coverage tool JaCoCo and method coverage tool ELLA.

- **A detailed study on the influence of code coverage metric granularity on automated testing.** By integrating ACVTool with Sapienz [MHJ16], an efficient automated testing framework for Android, we demonstrate that our tool can be **useful** as an integral part of an automated testing or security analysis environment. With ACVTool, we were able to compare how different coverage metrics fare in bug finding with Sapienz. We show that code coverage indeed makes the difference while searching for bugs and that different coverage granularities do not tend to find the same crashes, but none of them clearly outperforms the others. Also, we observed that the instruction-, method- and class- code coverage metrics are highly correlated to each other. These results demonstrate the interchangeability of the coverage metrics when an automated tool relies only on the absolute coverage value. Thus, this study answers **Research Question 3**.

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

- **Dynamic debloating approach based on ACVTool instruction coverage.** To answer **Research Question 4**, we propose the Dynamic Binary Shrinking System, a novel methodology created to shrink 3rd-party Android apps towards the observed benign functionality on executed code. We present a sophisticated technique for removal of not executed code from the app. We demonstrate that apps tend to keep significant amounts of not used code. More than 80% of apps code stayed not executed in our two samples. The shrinking technique did not lead to crashes or a change in their behavior.
- **An implementation of the dynamic debloating technique incorporated into the ACVCut tool.** The tool allows app producers and security analysts to cut potentially malicious functionality by removing not tested code from the app. Therefore, the testing may achieve 100% coverage on the shrunk app. It is based on top of ACVTool and contains 2K more lines of Python code.

1.5 List of Publications

The above mentioned contributions led to the following scientific output.

1. [PZG18] — Pilgun, A., Zhauniarovich, Y. and Gadyatskaya, O., 2018. Artifact. ACVTool: Fine-grained Code Coverage for a 3rd-party Android app. *Zenodo*, *Github*, URL: https://github.com/pilgun/acvtool.
2. [PGD+18] — Pilgun, A., Gadyatskaya, O., Dashevskyi, S., Zhauniarovich, Y. and Kushniarou, A., 2018, October. An effective Android code coverage tool. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (pp. 2189-2191).
3. [DGPZ18] — Dashevskyi, S., Gadyatskaya, O., Pilgun, A. and Zhauniarovich, Y., 2018. The influence of code coverage metrics on automated testing efficiency in Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (pp. 2216-2218).
4. [PGZ+20] — Pilgun, A., Gadyatskaya, O., Zhauniarovich, Y., Dashevskyi, S., Kushniarou, A. and Mauw, S., 2020. Fine-grained code coverage measurement in automated black-box Android testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4), pp.1-35.
5. [Pil20b] — Pilgun, A., 2020. Artifact. ACVCut: Dynamic Binary Shrinking for a 3rd-party Android app. *Zenodo*, *Github*. URL: https://github.com/pilgun/acvcut.

6. [Pil20a] — Pilgun, A., 2020. Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-party Android app. (accepted at *27th Asia-Pacific Software Engineering Conference (APSEC)*, in press).

Articles not included in this dissertation.

- [DZG+20] — Stanislav Dashevskyi, Yury Zhauniarovich, Olga Gadyatskaya, Aleksandr Pilgun, Hamza Ouhssain. 2020, March. Dissecting Android Cryptocurrency Miners. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy* (pp. 191-202).

1.6 Thesis outline

This dissertation presents the results achieved by unlocking of fine-grained app instrumentation that allowed us to measure code coverage Android apps at the instruction level. We further studied the impact of our instrumentation approach on the app performance and behavior, how such coverage is compliant to the other code coverage tools and how different coverage metrics influence automated testing. Finally, we made use of the high coverage granularity in our dynamic shrinking approach. We demonstrated its viability on two apps by showing that the shrunk apps produce 100% code coverage under selected tests.

This dissertation is based on the scientific output we listed above. Chapters 3, 4 and 5 are parts of our ACVTool CCS demo paper [PGD+18] and the TOSEM article [PGZ+20]. Moreover, Chapter 5 (and our TOSEM article [PGZ+20]) rejects the initial hypothesis expressed in our CCS poster [DGPZ18] regarding the influence of different coverage granularity on the efficiency of automated testing tools. Chapter 6 elaborates on the novel work which was recently accepted at APSEC'20.

With respect to this description, the dissertation is structured as follows.

Chapter 2 discusses the technical background necessary to understand this work. First, we introduce the reader to the internals of Android apps. Then we describe how the instrumentation works and is used in Android apps. Further, we give background on the automated testing techniques used for 3rd-party Android apps. Finally, we give an overview of the notion of debloating in Android.

In Chapter 3, we describe our technique on instrumenting Android apps to measure code coverage. We start from the code instrumentation details and further continue on ACVTool design. In the end of this chapter we discuss the related work on Android code coverage measurement.

Chapter 4 evaluate the effectiveness and efficiency of ACVTool and assesses how the coverage data reported by ACVTool is compliant to the data

measured by the JaCoCo system on the source code and Ella without source code. We present an evaluative comparison with related work in the end of this chapter.

Chapter 5 presents our results on integrating ACVTool with the Sapienz automated testing framework, evaluates the impact of ACVTool instrumentation on app runtime behavior, and discusses the contribution of code coverage data to bug finding in Android apps.

In Chapter 6, we propose the Dynamic Binary Shrinking System, a novel methodology created to shrink Android apps towards only executed code. This Chapter also presents the implementation of our approach in the ACV-Cut tool. We further demonstrate the viability of our cutting approach on two examples.

Finally, in Chapter 7, we conclude this dissertation and discuss the novel research directions and future works emerging from the results of this thesis.

Chapter 2

Background

This Chapter discusses the technical background necessary to understand this work. First, we introduce the reader to the internals of Android apps. Then we describe how instrumentation works and is used in Android apps. Further, we give background on the automated testing techniques used for 3rd-party Android apps. Finally, we give an overview of debloating in Android.

Contents

2.1 Android

Android is a modern mobile operating system, that was released with the first Android mobile device in 2008 by Google. Android now is an open source project run by Google, which consistently updates the operating system and releases it into Android Open Source Project (AOSP). Google has released 30 Android versions (by API level) with the latest release of Android 11 (API 30) on September 8, 2020. Although the original image of Android OS is run on a few devices, 3rd-party vendors make changes and maintain their own images of Android for their devices with specific (often proprietary) hardware drivers. However, such devices need to pass the Compatibility Test Suite, so the developers can rely on this compatibility when writing apps [MSBK19].

Figure 2.1 highlights five layers in modern Android. First of all, Android is based on the Linux Kernel. At this lowest level Android runs system-level services and drivers. The next layer is Hardware Abstraction Layer that serves as a middle layer between the device hardware capabilities and the higher-level Java API framework. The Libraries and Android Runtime (or Dalvik virtual machine prior to API level 21) layer hosts Android system libraries. Android Runtime (ART), in turn, is in charge of running Android apps in their own processes. The next layer is for the Java API Framework that includes reusable components for building Android apps. Finally, the System Apps layer provides a set of core apps included with the platforms that have a special status (e.g. an SMS messaging app and a calendar app). Developers can create their own apps for Android devices using its APIs, but they can also use other third-party components.

2.2 Android App Internals

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

Android apps are usually developed in Java and, more recently, in Kotlin – a JVM-compatible language [Cle17]. Upon compilation, code files are first transformed into Java bytecode files (`*.class`), and then converted into a Dalvik executable file (`classes.dex`) that can be executed by the Dalvik/ART Android virtual machine (DVM). Usually, there is only one `dex` file, but Android also supports multiple `dex` files. Such apps are called multidex applications.

In contrast to most JVM implementations that are stack-based, DVM

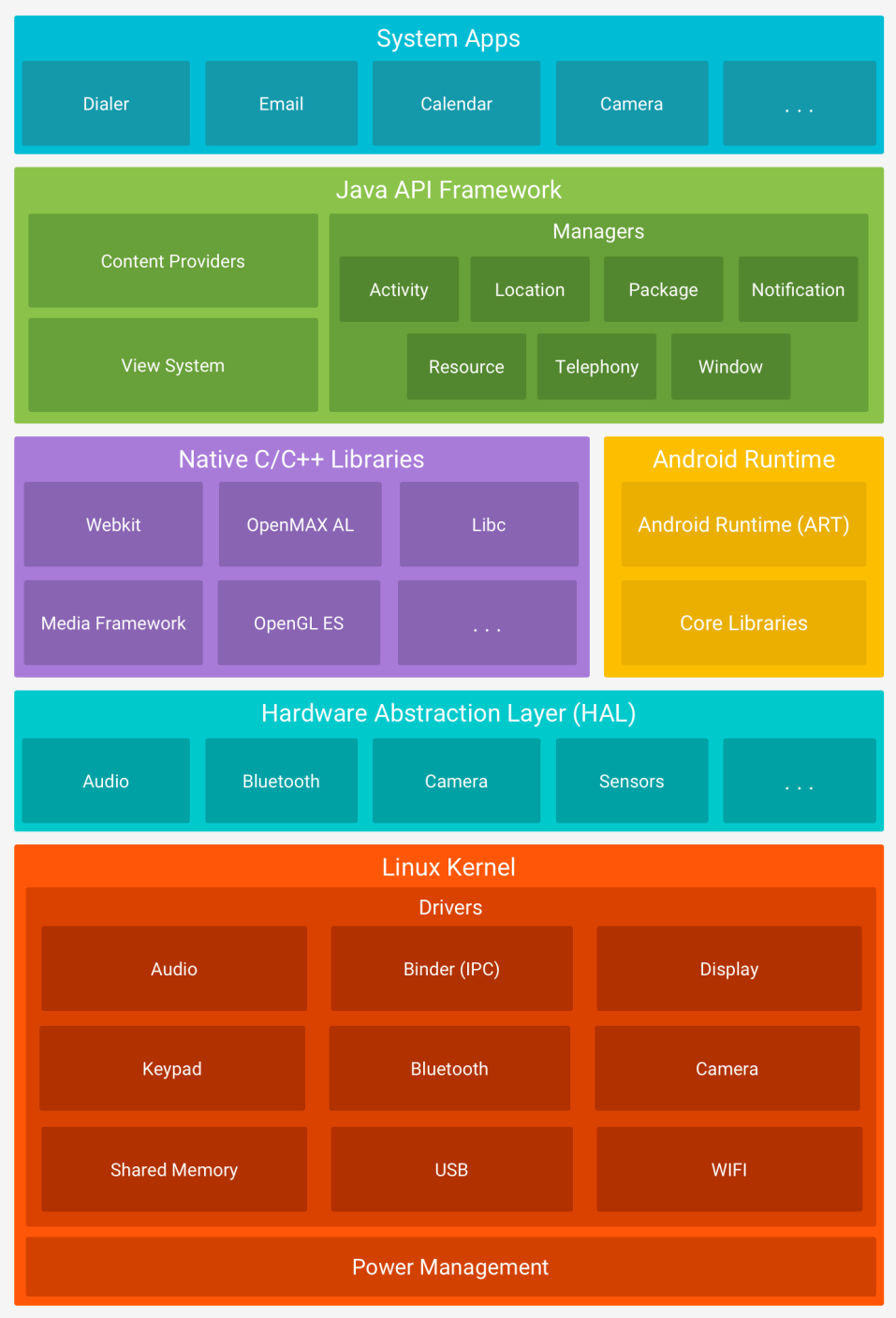


Figure 2.1: Android platform architecture.
Android Developers, Platform Architecture: https://developer.android.com/guide/platform (image source).

is a register-based virtual machine. It assigns local variables to registers, and the DVM instructions (opcodes) directly manipulate the values stored in the registers. Each application method has a set of registers defined in its beginning, and all computations inside the method can be done only through this register set. The method parameters are also a part of this set. The parameter values sent into the method are always stored in the registers at the end of the method's register set. For more details, we refer the interested reader to the official Android documentation about the Dalvik bytecode internals [Goo17a] and the presentation by Bornstein [Bor08].

Since raw Dalvik binaries are hard to understand for humans, several intermediate representations have been proposed that are more analyst-friendly: `smali` [Ben18, Goo18b] and `Jimple` [VRH98]. In this dissertation, we work with `smali`, a low-level programming language for the Android platform. `Smali` is supported by Google [Goo18b], and it can be viewed and manipulated using, e.g., the *smalidea* plugin for the IntelliJ IDEA/Android Studio [Ben18].

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 work: it allows one to specify the instrumentation class that can monitor at runtime all interactions between the Android system and the app. We rely upon this functionality to enable the code coverage measurement, and to intercept the crashes of an app and log their details.

Before an app can be installed onto a device, it must be cryptographically signed with a developer's certificate (the signature is located under the `META-INF` folder inside an `.apk` file) [ZGC+14]. The purpose of this signature is to establish the trust relationship between the apps of the same signature holder: for example, it ensures that the application updates are delivered from the same developer. Still, these signatures cannot be used to verify the authenticity of the developer of an application being installed, as other parties can modify the contents of the original application and re-sign it with their own certificates. Our approach relies on this possibility of code re-signing to instrument the apps.

2.3 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 [AO16]. 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 [THB+16], automatic test generation [MHJ16], and test prioritization [THB+16].

In the Android realm, not only application developers are interested in measuring code coverage. For example, Google tests all submitted (already packaged) apps to ensure that they meet the security standards[1]. For independent testers and analysts it is important to understand how well a third-party app has been exercised [KLG+18], and various third-party app testing and analysis tools are routinely evaluated with respect to the achieved code coverage [HCLT15, KLG+18, CGO15, WLY+18].

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 [AO16]. 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 [JaC18] and EMMA [Rub06]. On the other hand, the Android community often uses the *activity* coverage metric, that counts the proportion of executed activities [MHJ16, AN13, ZLZ+16, CML+17] (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 [Bor08]. 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 [THB+16, LMOD13].

It is not always possible to map the source code statements to the corresponding bytecode instructions without having the debug information. Therefore, it is practical to expect that the source code statement coverage cannot be reliably measured within the third-party app testing scenario, and in this work we resort to measuring the bytecode instruction coverage. We call the third-party app testing scenario *black-box testing*, to emphasize the absence of source code and implementation details. This terminology is standard in the Android community [KLG+18].

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

2.4 Protection from Malicious Apps

Taking into account the tremendous number and diversity of Android apps published by third-party developers on Google Play and beyond, the protection of Android users from malicious apps becomes a harsh task. Google, other industry security specialists and researchers yet do significant effort studying and patching the Android system.

Thus, according to the National Information Assurance Partnership (NIST), Google certified two models of their Google Pixel 3/3XL phones[2] as Protection Profile compliant to Common Criteria for Information Technology Security Evaluation (version 3.1, revision 5, April 2017). The document [Par20], particularly, mentions new libraries for secure app development, added in Android 9.0, rich APIs and the Mobile Device Management (MDM) system capable of remotely controlling phone settings. Indeed, certification allows government institutions, enterprises and individual users to trustfully use Pixel devices. Such devices seem the first Android candidates to be used in strictly secure environments.

However, the Common Criteria standard is expensive to implement, can not provide security guarantees, but rather specifies a set of security requirements [Jac07, Kar12]. For example, Beckert, Bruns and Grebing in their discussion paper, claim vague interpretation of Common Criteria standard (high level of policies abstraction in from specification can be interpreted in different ways in the implementation) and offer to strengthen the quality of software by using formal verification instead of functional testing since the effort their implementation is comparable [BBG10]. Moreover, the validation report on Google Pixel 3/3XL clearly states that *Target of Evaluation (TOE) does not include the user apps* running on top of the operating system [oSTA19]. Finally, the Enterprise version of Android is very new and it has not gained yet an interest from users and researchers (no alternative security analyses available yet).

According to Google, Android security is based on a multi-party consent model and includes the following principles [MSBK19].

- Actors control access to the data they create.
- Consent is informed and meaningful.
- Safe by design/default.
- Defense in depth.

The security principles implementation imposes security by design, declares requirements on the Android components and restricts the user from access to sensitive components. Furthermore, Android ensures user's consent

[2]Selected models are Google Pixel 3 (blueline) and Google Pixel 3 XL (crosshatch)

regarding the system actions or an app taking on the device. However, in practice, app developers find plenty of ways to circumvent Android security mechanisms and leak the end-user's data [RFW+19].

An app communicates with the Android OS through the application framework to perform various actions using Android APIs. Sensitive Android APIs cover such functions as obtaining user location or sending an SMS and stay behind a user-controlled permission mechanism [BBD+16]. At any time, the user can grant or take away the permission from the app.

On Android, sensitive APIs restricted by permissions are one of the main targets to both defenders and attackers [ZG16, FBR+16]. Indeed, early works focused on enforcing correct permissions usage in apps [BKLTM12]. Although permissions may be granted correctly, an attacker may use benign functionality declared in the app for malicious purposes. One of the mechanisms is to inject a logic bomb — malicious code that triggers only under peculiar conditions. Such code is, therefore, invisible under normal usage [FBR+16].

One approach to enforce user safety from malicious apps comes from the idea *to turn the incompleteness of dynamic analysis into a guarantee* [JvSRZ16, BLL18]. It offers a sandbox that restricts the app towards only explored behaviors. Another approach aims to remove the dead code (and decrease the attack surface) using static analysis [JZWL16, JWL16, JBW+18, HLPN18] or dynamically measuring traces of executed code [AJWK+19, ALN19, QHA+19].

2.5 Program Analysis

Researchers and practitioners continue improving static and dynamic techniques to identify malware. Both techniques however have their challenges, e.g. static analysis suffers from over-approximation, incompleteness of code, code obfuscation, while dynamic analysis — from *incompleteness* of testing [JvSRZ16]. Many researchers believe that hybrid analysis combining both static and dynamic benefits is more efficient [OAM+18].

Static analysis of 3rd-party Android apps often relies on the Soot [VRCG+10] framework, which converts app binaries into Jimple intermediate representation and provides vast capabilities for code analysis. Several use Soot and Jimple to cut excessive code from the apps [JZWL16, JWL16, JBW+18]

`Smali` is another intermediate representation for Android apps. It is supported by Google [Goo18b] and precisely reflects Android bytecode instructions into `smali` instructions making app bytecode human readable. Hence, `smali` representation gained significant popularity when reverse engineering, repackaging and hacking Android apps.

The goal of dynamic analysis is to report on the activities the app per-

formed when running. A number of automated testing tools such as DroidMate [JZ16], Droidmate-2 [BJHZ18], Droidbot [LYGC17] and simple Monkey [Goo18d] emerged to automate testing routines for different purposes. For example, Monkey is a simple input generator that allows us to quickly generate and inject UI events into the app, while Sapienz aims at the same time to maximize code coverage and the number of faults found when testing.

Unfortunately the current state of automated tools does not allow to fully test the app, while some tools managed to achieve pretty good results [BJHZ18]. Systematic literature review by Kong et al. provides an in-depth overview on automated testing tools [KLG+18].

The idea of restricting app behaviors by combining program analysis, sandboxing, and test generation tools for Android apps first appeared in the keynote by Zeller [Zel15]. He presented the definition of *test complement exclusion* term — disallowing behavior not seen during testing [Zel15]. The talk turned into the Mining Sandboxes paper by Jamrozik et al. [JvSRZ16]. The work presented BOXMATE — a tool for automated extraction of a sandbox from an Android app. The full report is available in the doctoral dissertation written by Jamrozik [Jam18].

Another outcome of Mining sandboxes are DroidMate [JZ16] and DroidMate-2 [BJHZ18] automated testing tools that consequently grew up from the BOXMATE tool. These automated testing tools help to observe behaviors in 3rd-party apps and the authors claim DroidMate-2 outperforms other popular automated tools — Droidbot [LYGC17] and Monkey [Goo18d].

Chapter 3

ACVTool: Instruction Coverage

Instruction coverage is a fine-grained code coverage metric, that we unlocked for Android apps in this work. We further elaborate on basic code coverage metrics relevant for a 3rd-party Android app and their implementation.

Contents

3.1 Code Instrumentation

In this section, we describe the bytecode instrumentation approach used in ACVTool. In the literature, Huang et al. [HCLT15] propose two approaches for measuring bytecode coverage: (1) *direct instrumentation* by placing probes right after the instruction that has to be monitored for coverage (this requires using additional registers); (2) *indirect instrumentation* by wrapping probes into separate functions. The latter instrumentation approach introduces significant overhead in terms of added methods that could potentially lead to reaching the upper limit of method references per `.dex` file (65536 methods, see [Goo18a]). 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 3.1: *Java* code example.

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

Listing 3.2: *Smali* representation of the original Java code example.

3.1.1 Bytecode Representation

To instrument Android apps, ACVTool relies on the `apkil` library [Yan18] that creates a tree-based structure of `smali` code. The tree generated by `apkil` contains classes, fields, methods, and instructions as nodes. It also maintains relations between instructions, labels, `try-catch` and `switch` blocks. We use this tool for two purposes: (1) `apkil` builds a structure representing the code that facilitates bytecode manipulations; (2) it maintains links to the inserted probes, allowing us to generate the code coverage report.

```
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 3.3: Instrumented *smali* code example. The highlighted lines mark the added instructions.

The original `apkil` library has not been maintained since 2013. Therefore, we adapted it to enable support for more recent versions of Android. In particular, we added annotation support for classes and methods, which has appeared in the Android API 19, and has been further extended in the API 22. Our modifications specify the `.annotation` word and its structure for classes and methods for the `apkil smali` parser. Other our additions

to `apkil` contain 4 new instructions: `filled-new-array`, `invoke-custom`, `filled-new-array/range` and `invoke-custom/range`. We added them to the list of `35c` and `3rc` Dalvik instruction formats[1]. For parsing, `apkil` finds such instructions by name as they have a different format compared to other instructions [Goo18a]. Thus, the `apkil` library evolves according to ACVTool's needs, and it is maintained within the ACVTool project.

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

3.1.2 Register Management

To exemplify how our instrumentation works, Listing 3.1 gives an example of a Java code fragment, Listing 3.2 shows its `smali` representation, and Listing 3.3 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 3.3). 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 [Goo18a]. Therefore, the addition of new registers could make these instructions malformed. Similarly, we cannot easily use new registers at the end of the stack because method parameter registers must always be the last ones.

To overcome this issue, we use the following approach. We allocate three new registers, however, in the beginning of a method we copy the values of the argument registers to their corresponding places in the original method. For instance, in Listing 3.3 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 3.2). Depending on the value type,

[1] See https://source.android.com/devices/tech/dalvik/instruction-formats for more details about instruction formats.

we use different `move` instructions for copying: `move-object/16` for objects, `move-wide/16` for paired registers (Android uses register pairs for `long` and `double` types), `move/16` for others. Then we update all occurrences of parameter registers through the method body from `p` names to their `v` aliases (compare Line 7 in Listing 3.2 with Line 17 in Listing 3.3). Afterwards, the last 3 registers in the stack are safe to use for the probe arguments (for instance, see Lines 4-6 in Listing 3.3).

3.1.3 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 3.3), 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 [Goo18a] (see Lines 17-18 in Listing 3.3). Therefore, we cannot insert a probe between them. Similarly, in case of an exception, the result must be immediately handled. Thus, a probe cannot be inserted between the `catch` label and the `move-exception` instruction.

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

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

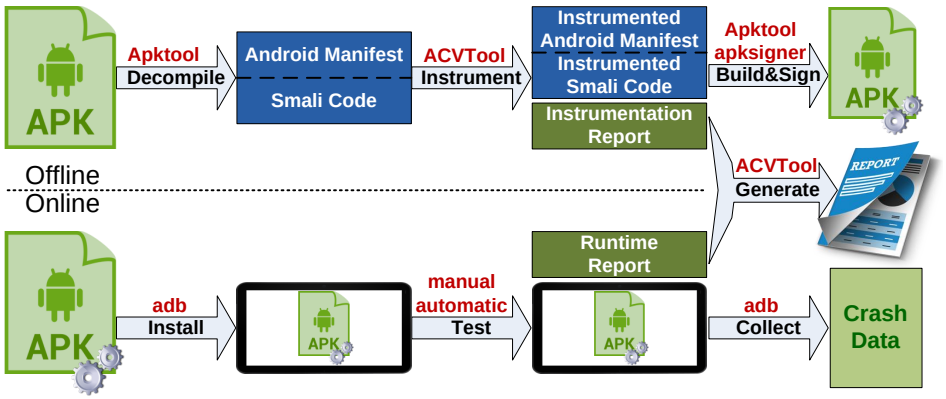


Figure 3.1: ACVTool workflow

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.

3.2 ACVTool Design

ACVTool allows one to *measure* and *analyze* the degree to which the code of a *closed-source* Android app is executed during testing, and to *collect crash reports* occurred during this process. We have designed the tool to be self-contained by embedding all dependencies required to collect the runtime information into the application under test (AUT). Therefore, our tool does not require to install additional software components, allowing it to be effortlessly integrated into any existing testing or security analysis pipeline. For instance, we have tested ACVTool with the random input event generator Monkey [Goo18d], and we have integrated it with the Sapienz tool [MHJ16] to experiment with fine-grained coverage metrics (see details in Section 5.1). 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 3.1 illustrates the workflow of ACVTool that consists of three phases: *offline*, *online* and *report generation*. At the time of the offline phase, the app is instrumented and prepared for running on a device or an emulator. During the online phase, ACVTool installs the instrumented app, runs it and collects its runtime information (coverage measurements and crashes). At the report generation phase, the runtime information of the app is extracted from the device and used to generate a coverage report. Below we describe these phases in detail.

3.2.1 Offline Phase

The offline phase of ACVTool is focused on app instrumentation. In a nutshell, this process consists of several steps depicted in the upper part of Figure 3.1. The original Android app is first decompiled using `apktool` [WT17]. Under the hood, `apktool` uses the `smali/backsmali` disassembler [Ben18] to disassemble `.dex` files and transform them into `smali` representation. To track the execution of the original `smali` instructions, ACVTool inserts special *probe* instructions after each of them. These probes are invoked right after the corresponding original instructions, allowing us to precisely track their execution at runtime. After the instrumentation, ACVTool compiles the instrumented version of the app using `apktool` and signs it with `apksigner`. Thus, by relying on native Android tools and some well-supported tools provided by the community, ACVTool is able to instrument almost every app. We present the details of our instrumentation process in Section 3.1.

In order to collect the runtime information, we used the approach proposed in [ZPG+15] and developed a dedicated `Instrumentation` class. ACVTool embeds this class into the app code, allowing the tool to collect the runtime information. After the app has been tested, this class serializes the runtime information (represented as a set of boolean arrays) into a binary representation, and saves it to the external storage of an Android device. The `Instrumentation` class also collects and saves the data about crashes within the AUT, and registers a broadcast receiver. The receiver waits for a special event notifying that the process collecting the runtime information should be stopped. Therefore, various testing tools can use the standard Android broadcasting mechanism to control ACVTool externally.

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

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

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

Along with the instrumented apk file, the *offline* phase produces an *instrumentation report*. It is a serialized code representation saved into a binary file with the `pickle` extension that is used to map probe indices in a

binary array to the corresponding original bytecode instructions. This data along with the runtime report (described in Section 3.2.2) is used during the *report generation* phase. By default, ACVTool instruments an application to collect instruction-, method- and class-level coverage information. It is also possible to instrument an app to collect only method- and class-level coverage data, in case only a coarser-grained coverage information is required.

3.2.2 Online Phase

During the online phase, ACVTool installs the instrumented app onto a device or an emulator using the `adb` utility, and initiates the process of collecting the runtime information by starting the `Instrumentation` class. This class is activated through a command issued to `adb`. Developers can then test the app manually, run a test suite, or interact with the app in any other way, e.g., by running tools, such as Monkey [Goo18d], IntelliDroid [WL16], or Sapienz [MHJ16]. 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.

3.2.3 Report Generation Phase

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

Currently, ACVTool generates reports in the `html` and `xml` formats. These reports have a structure similar to the reports produced by the JaCoCo tool [JaC18]. While `html` reports are convenient for visual inspection,

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 3.2: ACVTool *html* report

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

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

Figure 3.3: Covered `smali` instructions highlighted by ACVTool

`xml` reports are more suitable for automated processing. Figure 3.2 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 3.3).

3.2.4 Interface

ACVTool is a standalone Python package that can be introduced in other software such as automated software testing tools. ACVTool contains a number of functions to instrument, sign, build, install, measure code coverage and generate the code coverage report. We made source code available at the GitHub, which allows users to modify it and interfere in the code coverage measurement procedure at any step.

Furthermore, ACVTool provides its command line interface for manual use. We list the full set of capabilities in the Table 3.1 and Table 3.2. From the terminal ACVTool may be run as follows.

$ acv <**command**> <argument> <options>

Each command of the command line interface reflects on the ACVTool workflow presented in Figure 3.1. `instrument` command covers the decompilation of the APK, injecting the probes and modifying the Android Manifest file. After execution of this command ACVTool generates signed APK and the instrumentation report. Thus, this command covers the offline phase of ACVTool workflow. We additionally distinguished the `sign` command to give a user more flexibility, when using the tool. Further, commands `install`/`uninstall` duplicate the install/uninstall functionality of

Table 3.1: ACVTool command line interface.

Command	Argument	Description	Options
instrument	path_to_apk	Instruments an APK	--wd, -r, -i, --dbgstart, --dbgend
install	path_to_apk	Installs an APK	
uninstall	path_to_apk	Uninstalls an APK	
start	package_name	Starts runtime coverage data collection	
stop	package_name	Stops runtime coverage data collection	
report	package_name	Produces a report	-p, -o, -ec
sign	apk_path	Signs and alignes an APK	

the `adb` tool (with an additional flag). Commands `start` and `stop` launch and finalize the code coverage measurement process. The `report` command generates html and xml code coverage reports.

Debug capabilities. Along to the required options (Table 3.2), ACVTool offers the `dbgstart` and `dbgend` options that allow to instrument only selected methods or a part of the app, e.g. a half of all methods. This functionality mainly created for debug purposes to help in finding methods where ACVTool instrumentation breaks the app. The argument takes the number of a method in the app that we would like to instrument.

When the instrumented app did not work at the device anymore, we used the debug functionality to find the exact broken method and improve our approach. In this case, we would notice a crash instead of the demonstrated original behavior (e.g. the app could not launch anymore). Therefore, we would instrument only a part of the app, install it again and check if it still crashes. Thus, we could find the exact broken method leading to the crash. We automated the procedure of instrumenting, installing/uninstalling, launching of an app and catching a crash from the `logcat`, and we used the binary search algorithm to simplify the search of the broken method. Further, we would carefully investigate the insertion of probes in the particular method to reveal the reason of the crash.

3.3 Related Work

Literature on Software Testing [AO16, Mat13, KFN99] usually refers to code coverage in the context of a metric for *white-box* testing since it works on the source code. Developers traditionally use code coverage to measure quality of the code or completeness of unit and integration tests. On the contrary to

Table 3.2: ACVTool options for the commands listed in the Table 3.1

Option	Argument	Description
-h, --help	-	Shows help message and exits
--version	-	Shows program's version number and exits
--wd	result_directory	Path to the directory where the working data is stored (default: ./smiler/acvtool_working_dir)
--dbgstart	methods_number	For troubleshooting purposes. The number of the first method to be instrumented. Only methods from `DBGSTART` to `DBGEND` will be instrumented
-r, --r	-	Working directory (-wd) will be overwritten without asking
-i, --i	-	Installs the application immediately after instrumenting
-p	pickle_file	Path to the Pickle file, that was generated during the instrumentation process (required)
-o	output_dir	Output directory
-ec	ec_dir	The directory with the code coverage binary files pre-loaded from the emulator

white-box, *grey-box* and *black-box* testing do not have access to source code. Although the *grey-box* testing utilizes knowledge of app internals, *black-box* testing works without even this information. Therefore, code coverage term normally is not associated with *grey-box* nor with *black-box* testing. However, in the absence of source code we still can measure the proportion of (binary) code that was executed. In this case, coverage metric is frequently used to evaluate effectiveness and efficiency of automated testing tools that may operate in *black-box* or *grey-box* manner [HYWH15, ZPG+15, YH15, PGZ+20, PGD+18, DGPZ18]. On that occasion, many researchers and practitioners still refer to this measurement as code coverage, sometimes calling it *black-box* code coverage to indicate the lack of access to the app source code. Other terms may include binary instrumentation coverage, instruction coverage, binary coverage.

When an automated tool actually relies on code coverage measurements in its operation (utilizing it as a fitness function), we can talk about *grey-box* testing. The app may not have source code but the tool leverages information retrieved from the app internals. The tool would disassemble, read and analyze and even understand the app structure. Since the Android research community usually does not make a distinction between the

black-box and the *grey-box* settings, considering them both to be "black-box" [ZPG+15, PGZ+20, HYWH15, YH15], we will adopt this terminology. Further, we discuss the most popular examples of tools for *white-box* coverage measurement on Android, and then review the state of the art for the *black-box* testing approaches.

3.3.1 White-box Coverage Measurement

Code coverage measurement tools for white-box testing are included into the Android SDK maintained by Google [Goo18c]. Supported coverage libraries are JaCoCo [JaC18], EMMA [Rub06], and the IntelliJ IDEA coverage tracker [s.r17]. 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.

3.3.2 Coverage Measurement in Black-box

Several frameworks for measuring black-box code coverage of Android apps already exist, however, they have many drawbacks and are inadequate for large-scale automated testing, as we discuss in this section. Notably, these frameworks often measure code coverage at coarser granularity. For example, ELLA [ELL16], InsDal [LWD+17], CovDroid [YH15], and the tool by Horvath et al. [HBG+14] measure code coverage only at the method level.

ELLA [ELL16] is arguably one of the most popular tools to measure Android code coverage in the black-box setting, however, it is no longer supported. ELLA relies on the same approach to app instrumentation as ACVTool (at the method level): it inserts probes at the beginning of methods, manipulates registers and tracks probe execution [ELL16]. The difference in coverage measurement approaches appears in the reporting procedure. While executed, an app instrumented by ELLA sends identifiers of executed methods via a socket to the ELLA server.

Huang et al. [HCLT15] 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 reported a forbidding success rate – only 36% of apps could be instrumented. Such a low instrumentation rate makes it impossible to use this tool at large scale. Unfortunately, the authors stopped there and did not report on the reasons and possible fixes of appeared issues. The opposite, with ACVTool we performed an extensive reverse engineering work on failing apps to fix and generalize our ACVTool approach.

From the tool description Huang et al. implemented logging of the inserted probes by printing specific identifiers into `logcat` on the fly. To com-

Table 3.3: Coverage frameworks for black-box analysis

Tool	**References**	**Coverage granularity**	**Target representation**	**Code available**
ELLA	[ELL16, WLY+18]	method	`smali`	Y
Huang et al.	[HCLT15]	class, method, basic block, instruction	`smali`	N
BBoxTester	[ZPG+15]	class, method, basic block	Java	Y
InsDal	[LWD+17, YWYZ16, LWY+16]	class, method	`smali`	N
CovDroid	[YH15]	method	`smali`	N
Asc	[SQH17]	basic block	Jimple	Y
ABCA	[HYWH15]	class, method, instruction	Jimple	N
Horvath et al.	[HBG+14]	method	Java bytecode	N
Sapienz	[MHJ16]	activity	`smali`	Y
DroidFax	[CR17a, CMRY18, CR17b]	instruction	Jimple	Y
AndroCov	[BGZ18, Li16]	method, instruction	Jimple	Y
ACVTool	this work	class, method, instruction	`smali`	**Y**

pare, ACVTool registers probe execution in the binary array (Section 6.3) and saves a coverage file when testing is over. Therefore, Huang's approach may be heavy for an app and the Android system but also requires parsing Android log for coverage calculation. Unfortunately, Hunag et al. tool is not available for experimenting.

BBoxTester [ZPG+15] 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` [Pan17]. The `jar` files are instrumented using EMMA [Rub06], and assembled back into an apk.

The InsDal tool [LWD+17] 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 [YWYZ16], and for logging the number of method invocations in measuring the energy consumption of apps [LWY+16].

CovDroid [YH15], another black-box code coverage measurement system for Android apps, transforms apk code into `smali`-representation using the `smali` disassembler [Ben18] 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.

Alternative approaches to Dalvik instrumentation focus on performing detours via other languages, e.g., Java or Jimple. For example, Bartel et al. [BKM+12] worked on instrumenting Android apps for improving their privacy and security via translation to Java bytecode. Zhauniarovich et al. [ZPG+15] translated Dalvik into Java bytecode in order to use EMMA's code coverage measurement functionality, while Horvath et al. [HBG+14] used translation into Java bytecode to use their own `JInstrumenter` library for `jar` files instrumentation. The limitation of such approaches, as reported in [ZPG+15], 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 [SQH17], DroidFax [CR17a], ABCA [HYWH15], and AndroCov [Li16, BGZ18]. Jimple is a suitable representation for subsequent analysis with Soot [ARB17], yet, unlike `smali`, it does not belong to the "core" Android technologies maintained by Google. Moreover, Arnatovich et al. [ATD+14] 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 [MHJ16, SQH17, CR17a, LYGC17, CML+17, MHH+19, LR19] 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. ACV-Tool addresses this need of the community. As the coverage measurement is not the core contribution of these works, the authors have not provided enough details on the implementation of their coverage measurement techniques.

Chapter 4

ACVTool Evaluation

In this chapter, we evaluate effectiveness and efficiency of our instrumentation approach and compare it to other tools.

Contents

4.1 Introduction

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

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

- *Instrumentation success rate.* We report how many apps from our datasets have been successfully instrumented with ACVTool.
- *App health after instrumentation.* We measure the percentage of instrumented apps that can run on an emulator. We call these apps *healthy*[1]. To report this statistic, we installed the instrumented apps on the Android emulator and launched their main activity. If an app is able to run for 3 seconds without crashing, we count it as healthy.

Efficiency. We assess the following characteristics:

- *Instrumentation-time overhead.* Traditionally, the preparation of apps for testing is considered to be an *offline* activity that is not time-sensitive. Given that the testing process may be time-demanding (e.g., Sapienz [MHJ16] tests each application for hours), our goal is to ensure that the instrumentation time is insignificant in comparison to the testing time. Therefore, we have measured the time ACVTool requires to instrument apps in our datasets.
- *Runtime overhead.* Tracking instructions added into an app introduce their own runtime overhead, what may be a critical issue in testing. Therefore, we evaluate the impact of the ACVTool instrumentation on app performance and codebase size. We quantify the runtime overhead measured as the CPU utilization overhead on a subset of applications and on the benchmark PassMark application [Sof18] by comparing executions of original and instrumented app versions. We also measure the increase in the `.dex` file size.

Compliance with other tools. We compare the coverage data reported by ACVTool with the coverage data measured by JaCoCo [JaC18] that relies on the white-box approach and requires source code, and by Ella [ELL16], which does not require source code, but measures coverage

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

only at the method level. This comparison allows us to draw conclusions about the reliability of the coverage information collected by ACVTool.

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

4.2 Benchmark

We downloaded 1000 apps from the Google Play sample of the AndroZoo dataset [ABKT16]. 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 crashed upon startup. These crashes could appear due to some misconfigurations in the apps, but also due to the fact that we used an emulator. Android emulators lack many features present in real devices. We have used the emulator, because we subsequently test ACVTool together with Sapienz [MHJ16] (these experiments are reported in the next section). We excluded these unhealthy apps from our sample. In total, our **Google Play benchmark** contains **832** healthy apps. The apk sizes in this set range from 20KB to 51MB, with an average apk size of 9.2MB.

As one of our goals is to evaluate the reliability of the coverage data collected by ACVTool comparing to JaCoCo as a reference, we need to have some apps with the available source code. To collect such apps, we use the F-Droid[2] dataset of open source Android apps (1330 application projects as of November 2017). We could `git clone` 1102 of those, and found that 868 apps used Gradle as a build system. We have successfully compiled 627 apps using 6 Gradle versions[3].

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

[2]https://f-droid.org/

[3]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 4.1: 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

4.3 Effectiveness

4.3.1 Instrumentation Success Rate

Table 4.1 summarizes the main statistics related to the instrumentation success rate of ACVTool.

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

Besides the 24 apps that could not be repackaged in both app sets, ACVTool has instrumented all remaining apps from the Google Play benchmark. Yet, it failed to instrument 3 apps from the F-Droid set. The found issues were the following: in 2 cases `apktool` raised an exception `ExceptionWithContext` declaring an invalid instruction offset, in 1 case `apktool` threw `ExceptionWithContext` stating that a register was invalid and must be between `v0` and `v255`.

4.3.2 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 4.1). Five cases were due to the absence of the Retrofit annotation (four `IllegalStateException` and one `IllegalArgumentException`), 3 cases – `ExceptionInInitializerError`, 1 case – `NullPointerException`, 1 case – `RuntimeException` in a background service. In the F-Droid dataset, 2 apps became unhealthy due to the absence of Retrofit annotation, raising `IllegalArgumentException`.

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

Conclusion: we can conclude that ACVTool is able to process the vast majority of apps in our dataset, i.e., it is effective for measuring code coverage of third-party Android apps. For our total combined dataset of 1278 originally healthy apps, ACVTool has instrumented 1251, what constitutes 97.8%. From the instrumented apps, 1239 are still healthy after instrumentation. This gives us the instrumentation survival rate of 99%, and the total instrumentation success rate of 96.9% (of the originally healthy population). The instrumentation success rate of ACVTool is much better than the instrumentation rates of the closest competitors BBoxTester [ZPG+15] (65%) and the tool by Huang et al. [HCLT15] (36%).

4.4 Efficiency

4.4.1 Instrumentation-time Overhead

Table 4.1 presents the average instrumentation time required for apps from our datasets. It shows that ACVTool generally requires less time for instrumenting the F-Droid apps (on average, 27.4 seconds per app) than the Google Play apps (on average, 36.6 seconds). This difference is due to the smaller size of apps, and, in particular, the size of their `.dex` files. For our total combined dataset the average instrumentation time is 33.3 seconds per app. This time is negligible compared to the testing time usual in the black-box setting that could easily reach several hours.

4.4.2 Runtime Overhead

CPU utilization overhead To assess the runtime overhead induced by our instrumentation in a real world setting, we ran the original and instrumented versions of 10 apps (size range 1–32MB, 10MB mean APK size) randomly chosen from our dataset with Monkey [Goo18d], a random input event generator from Google (same seed for reproducibility, 1 second throttle, 500 events), and measured CPU utilization with the Qualcomm Snapdragon Profiler [Qua19].

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

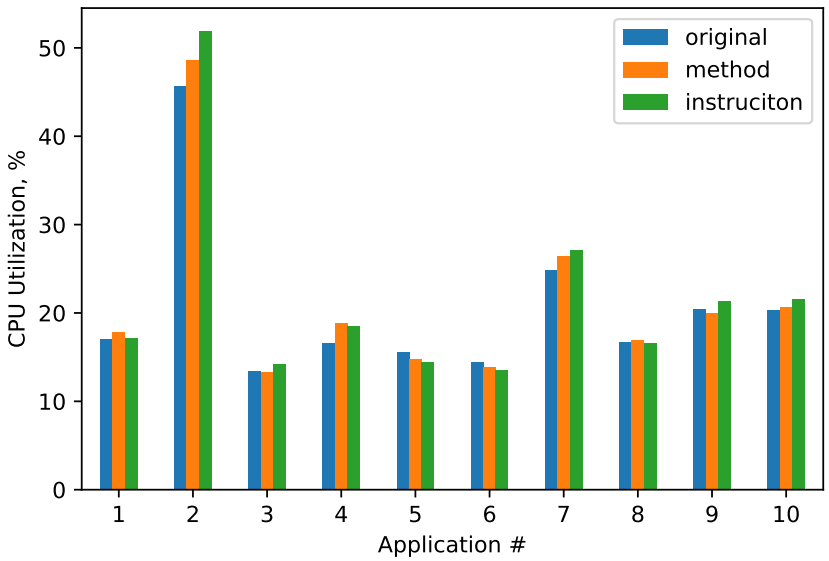


Figure 4.1: Average CPU utilization measured in 10 applications.

instrumented at the method level and instrumented at the instruction level. Finally, we calculated the average CPU utilization for every app version and logic processor (since the CPU on our device has 4 logic processors).

Figure 4.1 shows that CPU utilization of instrumented apps slightly differs from the CPU utilization by their original versions. However, as seen from Figure 4.2, the difference is insignificant: the mean difference of CPU utilization is 0.25% and 0.53% for the method- and instruction-instrumented versions, respectively. This experiment shows that the runtime overhead introduced by ACVTool is not prohibitive in a user-like application testing scenario.

PassMark overhead To further estimate the runtime overhead we used a benchmark application called PassMark [Sof18]. Benchmark applications are designed to assess performance of mobile devices. The PassMark app is freely available on Google Play, and it contains a number of test benchmarks related to assessing CPU and memory access performance, speed of writing to and reading from internal and external drives, graphic subsystem performance, etc. These tests do not require user interaction. The research community has previously used this app to benchmark their Android related-tools (e.g., [BBS+17]).

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 man-

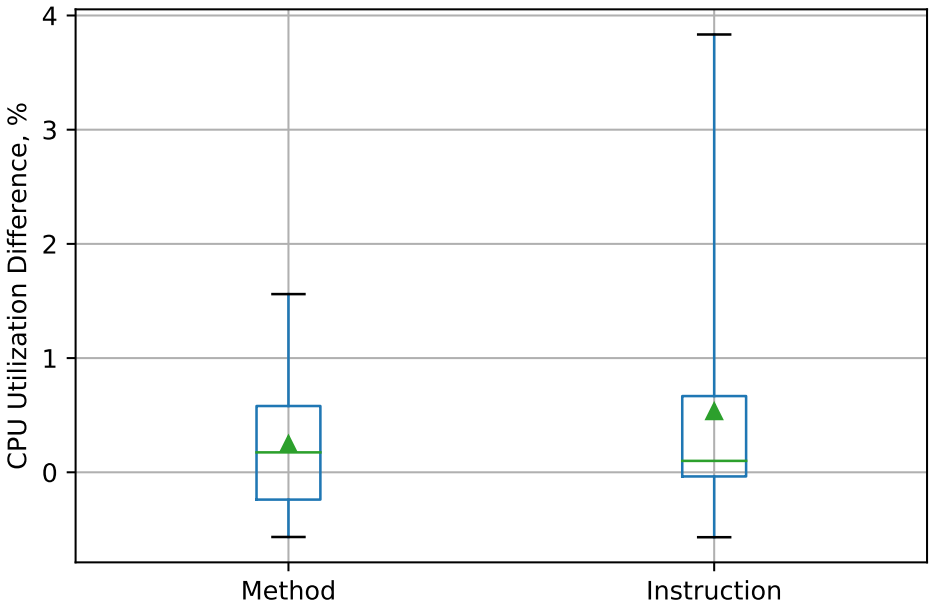


Figure 4.2: Boxplot of the CPU utilization difference for instrumented versions of applications (instrumented versions at the method-only and at the instruction level compared to the original app versions).

Table 4.2: PassMark overhead evaluation

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

aged runtime (Dalvik and ART) rather than on a bare metal using native libraries. We have prepared two versions of the PassMark app instrumented with ACVTool: one instrumented at the method level, and another instrumented at the instruction level.

Table 4.2 summarizes the performance degradation of the instrumented PassMark version in comparison to the original app. When instrumented, the size of the Passmark `.dex` file increased from 159KB (the original version) to 178KB (method granularity instrumentation), and to 556KB (instruction granularity instrumentation). We have run the Passmark application 10 times for each level of instrumentation granularity against the original version of the app. In the CPU tests that utilize high-intensity computations, Passmark slows down, 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.

Table 4.3: 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%)

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

Dex size inflation As another metric for overhead, we analysed how much ACVTool enlarges Android apps. We measured the size of `.dex` files in both instrumented and original apps for the Google Play benchmark apps. As shown in Table 4.3, 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 [LWD+17] has introduced `.dex` size increase of 18.2% (on a dataset of 10 apks; average `.dex` size 3.6MB), when instrumenting apps for method-level coverage. Thus, ACVTool shows smaller code size inflation in comparison to the InsDal tool.

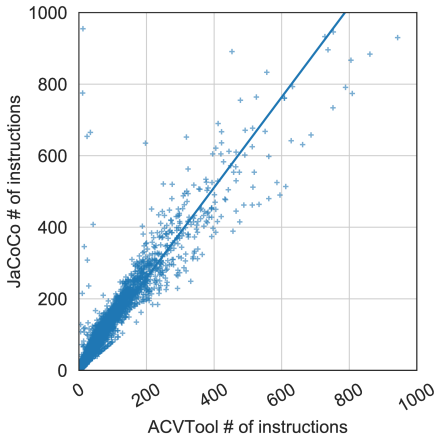
Conclusion: ACVTool introduces an off-line instrumentation overhead that is acceptable for common testing scenarios. The run-time overhead (measured as CPU utilization) in live testing with Monkey is negligible. When stress-testing with the benchmark PassMark app, ACVTool introduces 27% overhead in CPU.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

4.5 Compliance with Other Coverage Tools

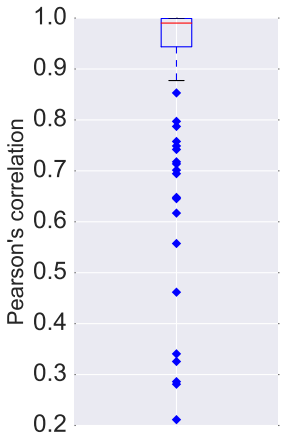
4.5.1 Instruction Coverage Measurement

When the source code is available, developers can log code coverage of Android apps using the JaCoCo library [JaC18] that could be integrated into the development pipeline via the Gradle plugin. We used the coverage data reported by this library to evaluate the correctness of code coverage metrics reported by ACVTool.

For this experiment, we used only the F-Droid benchmark because it



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



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

Figure 4.3: Compliance of coverage data reported by ACVTool and JaCoCo.

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

First, we analyze this app population in terms of instructions. Indeed, `smali` code and Java bytecode are organized differently. Figure 4.3a shows a scatterplot of the number of method instructions in `smali` code (measured by ACVTool, including the "untrackable" instructions) and in Java code (measured by JaCoCo). Each point in this Figure corresponds to an individual method of one of the apks in our benchmark. The line in the Figure is the linear regression line. The data shape demonstrates that the number of instructions in the `smali` code is usually slightly smaller than the number of instructions in the Java bytecode.

Figure 4.3a also shows that there are some outliers, i.e., methods that have low instruction numbers in `smali`, but many instructions in Java bytecode. We have manually inspected all these methods and found that outliers were constructor methods that contain declarations of arrays. `Smali` (and Dalvik VM) allocates such arrays with only one pseudo-instruction (`.array-data`), while Java bytecode is much longer [Bor08]. Given these

differences in the code organization, we can expect that, generally, there will be discrepancies in the coverage measured by ACVTool and JaCoCo.

Discrepancies in code coverage measurements can appear also due to the fact that some instructions are not tracked by ACVTool, as mentioned in Section 3.1. It is our choice to not count those instructions towards *covered*. In our F-Droid dataset, about half of the methods consist of 7 *smali* instructions or less. For such small methods, if they contain untraceable instructions, code coverage measurements by ACVTool and JaCoCo can differ substantially.

To compare the measured coverage data, we ran two copies of each app (instrumented with ACVTool and with JaCoCo) on the Android emulator using the same Monkey scripts for both versions. Figure 4.3b shows a boxplot of the correlation of code coverage measured by ACVTool and JaCoCo. Each data point corresponds to one application, and its value is the Pearson correlation coefficient between percentage of executed code, for all methods included in the app. The minimal correlation is 0.21, the first quartile is 0.94, median is 0.99, and maximal is 1.00. This means that for more than 75% of apps in the tested applications, their code coverage measurements have correlation equal to 0.94 or higher, i.e., they are strongly correlated. The boxplot in Figure 4.3b contains a number of outliers that appear due to the reasons explained above. Still, overall, the boxplot demonstrates that code coverage logged by ACVTool is strongly correlated with code coverage logged by JaCoCo.

4.5.2 Method-level Coverage Measurements

When the application source code is not available, testers cannot use JaCoCo to measure code coverage. In this situation researchers and practitioners frequently use the ELLA library [ELL16] to measure the method coverage [MHJ16, WLY+18]. As ELLA is no longer maintained, ACVTool can be now used by testers to measure code coverage at the method level, if such need arises.

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

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

Figure 4.4 shows scatterplots of method coverage measurements for pairwise comparison of data from the three coverage tools; each data point corresponds to an application. This Figure demonstrates that the vast majority

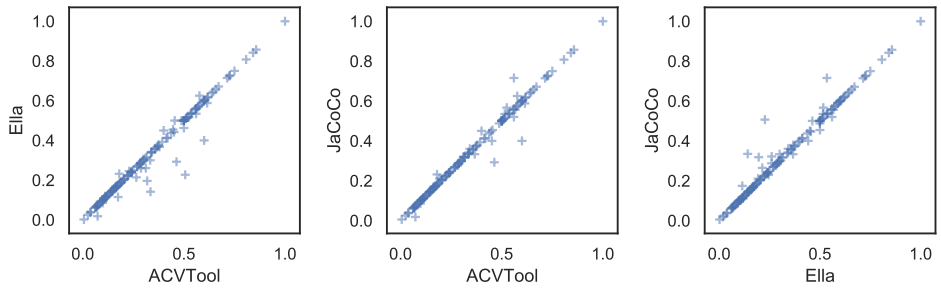


Figure 4.4: Code coverage measurements at the method level: pair-wise coverage comparisons for ACVTool, ELLA and JaCoCo.

of data points lie on the symmetry line from (0,0) to (1,1), i.e., the tools report practically identical method coverage results for most of the apps in this set. The deviations from the main line are results of possible differences in app behavior (further elaborated in 5.1.3).

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

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

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

4.6 Comparison to Other Coverage Frameworks

Unfortunately, other coverage frameworks lack of detailed evaluation such as we performed for ACVTool in this chapter. In this section we summarized available information with regards to the described in the previous Chapter related work on coverage measurement for black-box analysis.

An empirical study by Wang et al. [WLY+18] has evaluated performance of Monkey [Goo18d], Sapienz [MHJ16], Stoat [SMC+17], and

Table 4.4: Evaluation of coverage frameworks for black-box analysis

Tool	**References**	**Sample size**	**Instrumentation success rate (%)**		**Overhead**		**Compliance evaluated**
			Instrumented	**Executed**	**Instr. time (sec/app)**	**Run time (%)**	
ELLA	[ELL16, WLY+18]	68	60%	60%	N/A	N/A	N
ELLA	this work	1278	95.9%	91.1%	15.7	N/A	Y
Huang et al.	[HCLT15]	90	36%	N/A	N/A	N/A	Y
BBoxTester	[ZPG+15]	91	65%	N/A	15.5	N/A	N
InsDal	[LWD+17, YWYZ16, LWY+16]	10	N/A	N/A	1.5	N/A	N
CovDroid	[YH15]	1	N/A	N/A	N/A	N/A	N
Asc	[SQH17]	35	N/A	N/A	N/A	N/A	N
ABCA	[HYWH15]	6	N/A	N/A	N/A	9-86% of system time	N
Horvath et al.	[HBG+14]	10	N/A	N/A	N/A	N/A	N
Sapienz	[MHJ16]	1112	N/A	N/A	N/A	N/A	N
DroidFax	[CR17a, CMRY18, CR17b]	195	N/A	N/A	N/A	N/A	N
AndroCov	[BGZ18, Li16]	17	N/A	N/A	N/A	N/A	N
ACVTool	**this work**	**1278**	**97.8%**	**96.9%**	**33.3**	**up to 27%**	**Y**

WCTester [ZLZ+16] automated testing tools on large and popular industry-scale apps, such as Facebook, Instagram and Google. They have used ELLA to measure method code coverage, and they reported the total success rate of ELLA at 60% (41 apps) on their sample of 68 apps.

In our own experiment with ELLA reported in Section 4.5.2, ELLA has nearly the same instrumentation success rates as ACVTool: the same 98.7% apps in the F-Droid dataset, and 94.4% in the Google Play dataset (against 97.8% for ACVTool). After the ELLA instrumentation, in total, 91.1% out of 1278 apps are healthy (against 96.9% for ACVTool). While success rates are similar between the tools (ACVTool performs slightly better), ACVTool does more sophisticated instrumentation at the instruction level and, therefore, takes twice as much time as compared to ELLA.

Huang et al. [HCLT15] reported a low instrumentation success rate of 36%, and only 90 apps have been used for evaluation. Unfortunately, the tool is not publicly available, and we were unable to obtain it or the dataset by contacting the authors. Because of this, we cannot compare its performance with ACVTool, although we report a much higher instrumentation rate, evaluated against a much larger dataset.

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.

Liu et al. [LWD+17] evaluated their InsDal tool on a limited dataset of 10 apps, but did not provide instrumentation success rate. The authors have reported an average instrumentation time overhead of 1.5 sec per app, and an average instrumentation code overhead of 18.2% of `dex` file size. ACVTool introduces a smaller code size overhead of 11%, on average, but requires more time to instrument an app. On our dataset, the average instrumentation time is 24.1 seconds per app, when instrumenting at the method level only. It is worth noting that half of this time is spent on repackaging with `apktool`.

While the instrumentation approach of CovDroid [YH15] is similar in nature to our ACVTool, the former tool has been evaluated on a single application only.

Among the Android application instrumentation approaches, the most relevant for us are the techniques discussed by Huang et al. [HCLT15], InsDal [LWD+17] and CovDroid [YH15]. 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 [DWZ12] and Swift-

Hand [CNS13] frameworks, which do not report their instrumentation success rates.

Table 3.3 aggregates the related work discussed in Section 3.3 and 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., [HCLT15] and [ZPG+15]. 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 [ELL16] and BBoxTester[ZPG+15]) or publicly unavailable tools ([HCLT15, YH15]).

Chapter 5

The Influence of Code Coverage Metrics on Automated Testing

This Chapter presents our results on integrating ACVTool with the Sapienz automated testing framework, evaluates the impact of ACVTool instrumentation on app runtime behavior, and discusses the contribution of code coverage data to bug finding in Android apps.

Contents

5.1 Usefulness of ACVTool in Testing with Sapienz

To assess the usefulness of ACVTool in practical black-box testing and analysis scenarios, we integrated ACVTool with Sapienz [MHJ16] – a state-of-art automated Android search-based testing tool. Its fitness function looks for Pareto-optimal solutions using three criteria: code coverage, number of found crashes and the length of a test suite. This experiment had two main goals: (1) demonstrate that ACVTool fits into a real automated testing/-analysis pipeline; (2) evaluate whether the fine-grained code coverage measure provided by ACVTool can be useful to automatically uncover diverse types of crashes with black-box testing strategy.

Sapienz integrates three approaches to measure code coverage achieved by a test suite: EMMA [Rub06] (reports source code statement coverage); ELLA [ELL16] (reports method coverage); and its native plugin to measure coverage in terms of launched Android activities. EMMA does not work without the source code of apps, and thus in the black-box setting only ELLA and own Sapienz plugin could be used. The original Sapienz paper [MHJ16] did not evaluate the impact of the code coverage metric used on the discovered crashes population, because it was not possible to compare results for the same group of applications.

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

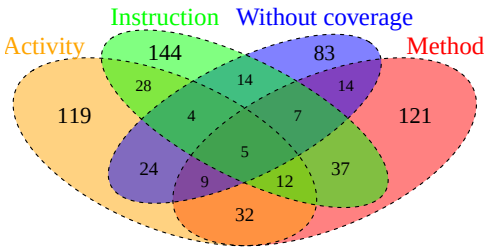
As our dataset, we use the healthy instrumented apks from the Google Play dataset described in the previous section. We have run Sapienz against each of these 799 apps, using its default parameters. Each app has been tested using the activity coverage provided by Sapienz, and the method and instruction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

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

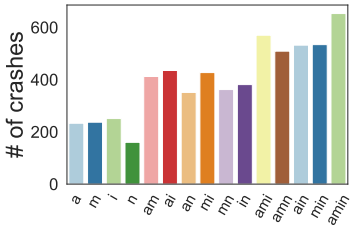
Like many other automated testing tools for Android, Sapienz is non-

Table 5.1: Crashes found by Sapienz in 799 apps

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



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



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

Figure 5.1: Crashes found by Sapienz.

deterministic. Thus, we apply statistical tests to analyze significance of all reported findings. Throughout this section, we will evaluate how effectively Sapienz finds bugs with each coverage metric in a particular app. For each app population, we obtain records of found crashes in each application, and we compare performance of each coverage metric *per each app record*. This gives us paired measurements for all coverage metrics, which are not necessarily normally distributed. Thus, to evaluate statistical significance of the results, we use the non-parametric Wilcoxon signed-rank test [WRH+12] that is appropriate in this setting. The *null-hypothesis* for the Wilcoxon test is that there is no difference which metric to use in Sapienz. *Alternative hypothesis* is that Sapienz with one coverage condition will consistently find more crashes than Sapienz with another coverage condition.

To measure the effect size we use the Vargha-Delaney A12 statistics [VD00] that was applied in the original Sapienz paper [MHJ16].

5.1.1 Descriptive Statistics of Crashes

Table 5.1 shows the numbers of crashes grouped by coverage metrics that Sapienz has found in the 799 apps. We consider a *unique crash* as a distinctive combination of an application, its component where a crash occurred, the line of code that triggered an exception, and a specific Java exception type.

In total, Sapienz has found 353 apps out of 799 to be faulty (at least one crash detected), and it has logged 653 unique crashes with the four coverage conditions. Figure 5.1a summarizes the crash distribution for the coverage metrics. The intersection of the results for all code coverage conditions contains only 5 unique crashes. Individual coverage metrics have found 38% (instruction coverage), 36% (method coverage), 36% (activity coverage), and 24% (without coverage) of the total number of found crashes. These results suggest that coverage metrics at different granularities find distinct crashes.

In these experiments, instruction coverage has shown slightly better performance in bug finding as it found more crashes on the dataset. However, when comparing chances to find a bug in a particular app, its edge over the activity and method coverage is not statistically significant according to the Wilcoxon signed-rank test [WRH+12]. On the other hand, all valid coverage metrics outperform testing without coverage data in a statistically significant way (p-values $\leq 10^{-4}$). Still, Vargha-Delaney effect sizes [VD00] are very small: 0.52 for method and instruction coverage (compared to no coverage), and 0.53 for activity coverage. Thus, we can conclude that it is likely that Sapienz with coverage performs better than without coverage data. However, the practical importance of coverage data used in Sapienz may be limited.

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

5.1.2 Evaluating Bug Finding Efficiency on Multiple Runs

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

Performance in 5 Runs

We have run Sapienz for 5 times with each coverage metric and without coverage data, for 3 hours per each of 150 apps. This gives us two crash

Table 5.2: Crashes found in 150 apps with 1 and 5 runs.

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

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 5.2 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 5.2, the activity coverage has found a larger proportion of total crash population (38% from 32%). Sapienz without coverage data also shows better performance over multiple runs (36% from 31%), while the instruction coverage has increased performance from 36% to 40%. The method coverage has achieved the best improvement (45% form 34%). For all coverage metrics, the increases in the found crashes populations due to repeated testing are statistically significant according to the Wilcoxon signed-rank test (p-values $\leq 10^{-5}$), but the Vargha-Delaney effect sizes [VD00] are small: all in the range (0.61, 0.64). Thus, repeating Sapienz test executions improves chances to find a crash in an app, but not a lot. The edge of method coverage over other metrics in repeated experiments is not statistically significant.

These findings suggest that even with 5 repetitions a single coverage metric is not able to find all crashes that were detected by other metrics. Our results in this experiment are consistent with a previously reported smaller experiment that involved only 100 apps (see [DGPZ18] for more details).

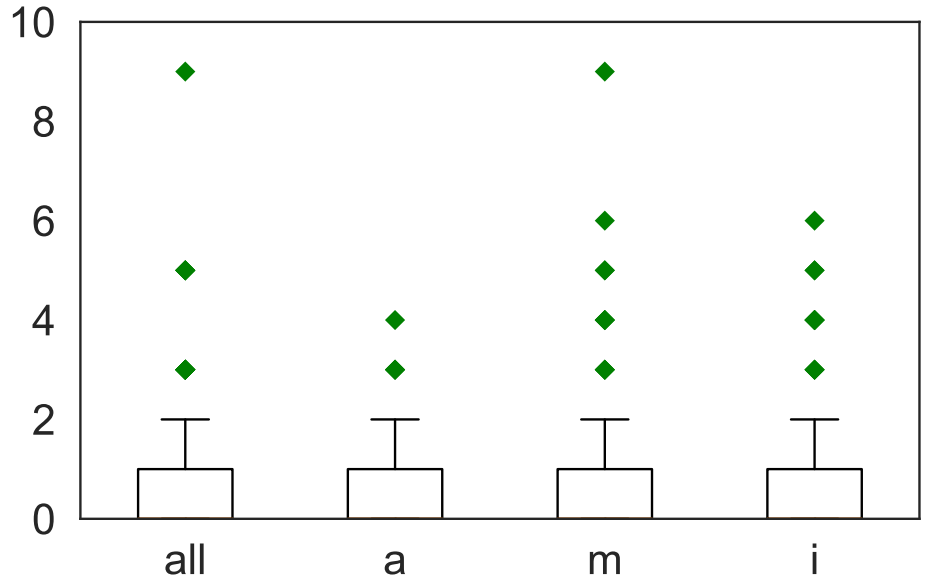


Figure 5.2: Boxplots of crashes detected per app (*a* stands for activity, *m* for method, and *i* for instruction, respectively).

Evaluating a Combination of Metrics

The previous experiment indicates that even repeating the runs multiple times does not allow any of the code coverage metrics to find the same number of bugs as all metrics together. We now fix the time that Sapienz spends on each apk[1], and we want to establish whether the number of crashes that Sapienz can find in an apk with 3 metrics is greater than the number of crashes found with just one metric but with 3 attempts. This would suggest that the combination of 3 metrics is more effective 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 5.3 summarizes the basic statistics for the apk crash numbers data, and the data shapes are shown as boxplots in Figure 5.2. The summary statistics show that Sapienz equipped with 3 coverage metrics has found, on average, slightly more crashes per apk than Sapienz using only one metric but executed 3 times. To verify this, we apply the Wilcoxon signed-rank test [WRH+12].

The results of the Wilcoxon test did not reject the null-hypotheses for

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

Table 5.3: 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	0.65	0.48	0.64	0.58
Median	0	0	0	0
3rd. Quartile	1	1	1	1
Max	9	4	9	6

all coverage metrics (p-values 0.43 and 0.58, and 0.51 for activity, method and instruction coverage, respectively). This can be interpreted as a high probability that the crashes data have been drown from similarly distributed populations.

To confirm this negative result, we apply the Vargha-Delaney A12 statistic to measure the effect size. The A12 effect sizes for differences in the crash population found by 3 metrics jointly and the population found by the activity, method and instruction coverage are, respectively, 0.515, 0.516 and 0.513, which all correspond to a negligible effect.

Our findings from this experiment are not fully consistent with the previously reported experiment on a smaller set of 100 apps [DGPZ18]. The difference could be explained by the following factors. First, we have used only healthy instrumented apps in this experiment (the ones that did not crash upon installation). The experiment reported in [DGPZ18] did not involve the check for healthiness, and the crashing apps could have affected the picture. In the unhealthy app case, Sapienz always reports one single crash for it, irrespectively of which coverage metrics is used. Note that in our Google Play sample approximately 17% are unhealthy, i.e., they cannot be executed on an emulator, as required by Sapienz. Second, the new apps tested in this experiment could have behaved slightly differently than the previously tested cohort. And, finally, in these experiments we used more recent releases of the testing environment components, including the Android SDK, that are more stable and have less compatibility issues.

5.1.3 Impact of ACVTool on Sapienz

Instrumentation and repackaging of the app's codebase may introduce differences in runtime behavior and additional faults. Such deviations can make parts of the app unreachable, which may impact further testing. Despite our positive evaluation demonstrated in Section 4, automated testing tools, such as Sapienz, look deeper into the app and can be more significantly impacted by issues raised by instrumentation. Here we analyze how much does ACVTool interfere with the Sapienz testing process. We consider two

main aspects.

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

Preserving behavior

To evaluate the ACVTool impact on app behavior we designed the following experiment. For every app from the 150 apps subset mentioned in Section 5.1.2 we took the most evolutionary developed Sapienz test suite and ran it on two versions of the app: original and instrumented at instruction level. After every triggered event we saved a screenshot of the UI state and its XML layout (with the help of UIAutomator [Goo19]). Then we removed the status bar (around 90px at the image top) from every image and performed an automated comparison of images and XML files for the original and instrumented versions in Beyond Compare [Sco19].

The publicly available version of Sapienz produces sequences containing mostly atomic Monkey [Goo18d] events, with one exception. An event named `GUIGen` in the sequence produces up to 12 random events on Android. Thus, in this experiment we excluded the `GUIGen` line from all the sequences to achieve sequence reproducibility. Moreover, we kept 1 second pause between the events to make sure that content loading and app animation have lower impact on the produced screenshots.

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

- Pop-ups: One of the apps in the pair in some cases fires a pop-up related to the Android OS state or the app itself. This happens to both instrumented and original apps. In this case we re-run the test and it solves the problem.
- Advertisement: Apps frequently load ads, which may look differently each time, as shown in Figure 5.3).
- Dynamic content: Some apps may display their UI each time differently or download completely new content, as exemplified in Figure 5.3.
- PNG artefacts: Apktool sometimes breaks PNG files or changes the color and transparency properties during decoding. Therefore, parts of the app may look different.

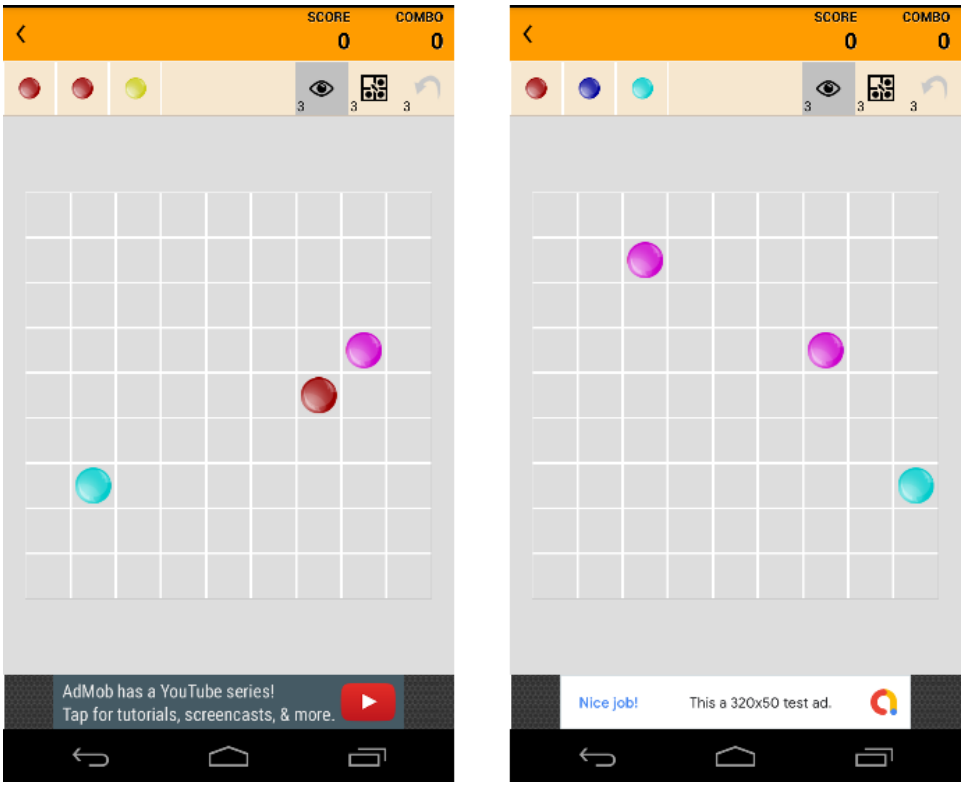


Figure 5.3: Example of a justified difference in behavior between the original (left) and the instrumented (right) app versions: the screenshots are different due to the dynamic nature of the app itself and the loaded ad content.

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

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

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

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

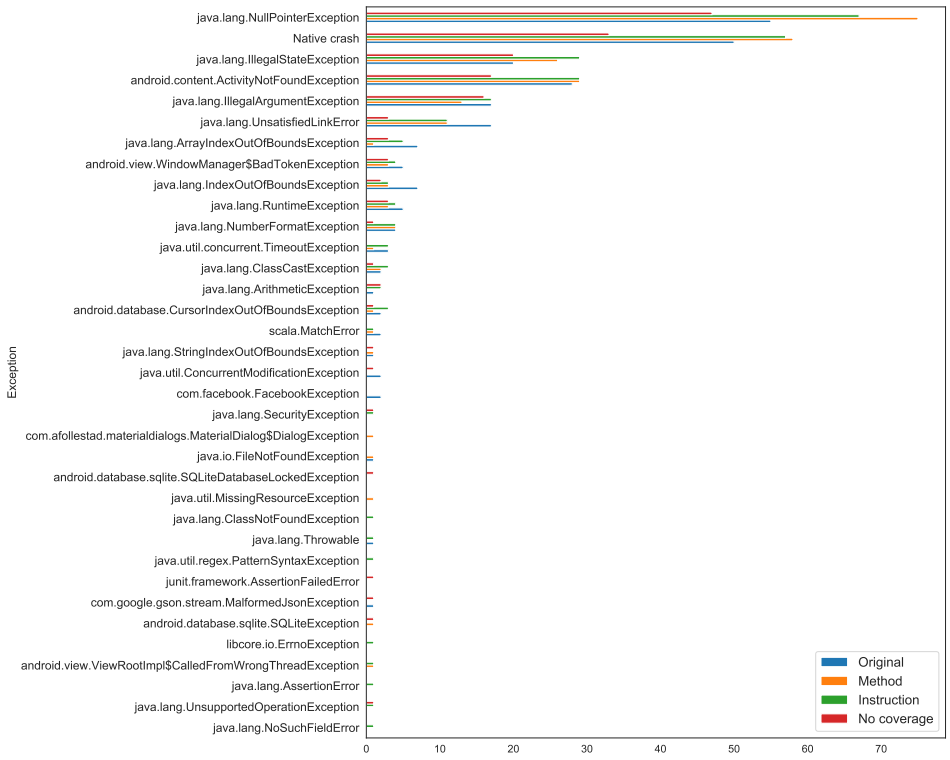


Figure 5.4: Distribution of the crash types (exceptions) found by Sapienz with different code coverage granularities. Exceptions are sorted in the descending order with respect to the total number of unique found crashes with this exception type.

Fault analysis

Figure 5.4 demonstrates the distribution of crashes found by Sapienz with respect to the code coverage metric applied. The 653 unique crashes found on the set of 799 apps were caused by 35 exception types. We note that our crash type distribution resembles the results reported in the original Sapienz paper for the main crash types on the Google Play subjects [MHJ16]. The most prevalent Java exception types in Figure 5.4 also generally agree with the statistics of Java exceptions in open source Android projects reported by [CAG+17]. As Android bug finding is not the core goal of this work, we limit ourselves to reporting the general crash distribution as provided by Sapienz, and we do not focus here on attributing the root causes of crashes as in, e.g., [LVBT+17].

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

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

As we described in Section 3.2.1, during the offline phase ACVTool embeds the custom `Instrumentation` class and probes into apps. We carefully analyzed the stack traces of all exceptions obtained in our experiment with Sapienz and did not find any evidence of ACVTool methods there. However, the probes we embed do not use method calls, but rather directly change binary array values corresponding to original bytecode instructions at runtime. Thus, if any fault happens due to these probes we will not see any probe-specific symptoms in the stack trace. Therefore, the first approach cannot completely confirm the absence of crashes introduced by ACVTool.

The applicability of the second approach is hurdled by flaky tests. Flaky tests are those that do not reliably fail even in identical circumstances. The authors of the follow-up paper about Sapienz [AGH+18] admit that the tool is subject to flaky tests, but they do not provide estimates about how many tests are flaky. It is mentioned only that "it is safer to assume that we live in a world where all tests are flaky", what may indicate that the flaky tests proportion is high.

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

The main reason for this low reproducibility of faults is the asynchronous nature of Android [AGH+18]. Depending on the wait time, an asynchronous call to a service may produce different results. When the service returns a value in time, this value is used, but if a value is not returned, the default value is used. This may lead to completely different execution paths.

We should also mention that the default throttle setting that Sapienz uses for Monkey is quite aggressive. It intensively bombards an app with events irrespective of the app's state and its animation. Since Monkey's throttle parameter significantly affects crash detection [PSRN18], satisfactory crash reproducibility on Sapienz may be achieved with a proper throttle value (e.g., as we set in Section 5.1.3). However, the consequence and a huge disadvantage would be a dramatic slowing down of Sapienz in finding new faults. Still, even this approach cannot guarantee full reproducibility of the crashes. Thus, with this approach we cannot confirm that ACVTool does not introduce new faults, because these faults could be due to flaky tests.

Thus, we can confidently confirm only the crashes described in Section 4.3.2 as caused by the ACVTool instrumentation phase. However, out of the 5 exception types found when filtering healthy apps in Section 4.3.2, 3 types – `IllegalStateException`, `IllegalArgumentException`,

`NullPointerException` – appear prevalently in the found crashes distribution. We expect that ACVTool could have contributed to at least some of these exceptions.

5.1.4 Analysis of Results

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

We can also conclude that better investigation and integration of different coverage granularities is warranted in the automated Android testing domain, just like in software testing in general [CPTH17]. Our crash data analysis and the experiment with repeating executions 5 times show that no coverage metric is able to find the vast majority of the total found crash population. Sapienz without coverage finds fewer bugs than with coverage data (160 crashes on the total app population versus, e.g., 233 crashes found with the activity coverage), yet it is still able to uncover a significant crash population. Further investigation of these aspects could be a promising line of research. Our open-source ACVTool can be helpful in these studies.

5.2 Comparison of Coverage Metrics Measurement

As mentioned, many automated testing tools for Android apps use various code coverage metrics to achieve better results in bug finding [CGO15, KLG+18, WLY+18]. With ACVTool such tools are now able to utilize the fine-grained instruction coverage while working directly on Android apps. However, it is currently not evident from the Android literature whether the instruction-level coverage will be a game-changer in the software testing community, or whether more coarse-grained coverage metrics, e.g., activity coverage [MHJ16], might be sufficient for the testing needs.

Researches and app developers can use line coverage provided by JaCoCo [JaC18] or EMMA [Rub06] when the app source code is available, while method and activity coverage were mainly used in black/grey-box testing before c ACVTool. For example, authors of EvodDroid [MP15], SIG-Droid [MBMM15], Sapienz [MHJ16] and Stoat [SMC+17] measured code coverage using EMMA on the app source code. Both Sapienz and Stoat used Ella to measure method or implemented activity coverage measurement on their own when working with 3rd-party apps (see more details

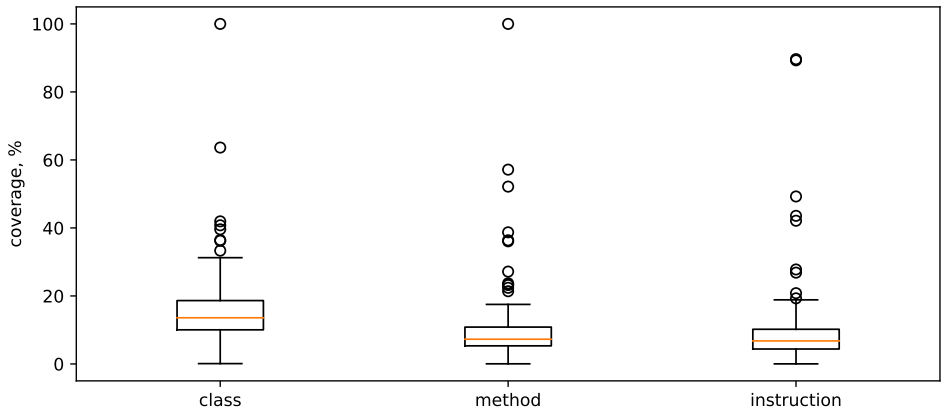


Figure 5.5: Code coverage measured at class, method, and instruction levels for 141 apps in our F-Droid dataset.

examples in the previous chapter). Yang et al. [YHH19] explain these different treatments by the lack of finer-grained code coverage metric in Android closed-source app testing.

We would like to empirically establish how do measurements of different coverage granularity relate to each other in the context of automated Android app testing. In this section we report on a small study with apps from our dataset that evaluates the code coverage differences as measured at the instruction, method and class levels. Our study empirically validates the results presented by Yang et al. [YHH19] and concludes a small advantage of the instruction-level code coverage over the method- and class-level coverage.

To compare coverage metrics we run Monkey just once over our set of 141 F-Droid apps instrumented by ACVTool. The generated code coverage reports provide all necessary information about instruction, method and class coverage at the same time. Figure 5.5 shows how coverage is measured in terms of class, method and instruction, with median values at 13.6%, 7.3%, and 6.8% respectively. Correlation between the coverage metrics over 1785068 methods in 213652 classes in our set goes as follows: 94% instruction-method, 88% method-class, 75% instruction-class. These results show high interchangeability of measured metrics and a possibility of using less accurate metric in more demanding circumstances, specifically on performance-sensitive apps.

On the contrary, method coverage hides valuable knowledge such as whether all instructions in the method were executed and how many of them have run. To answer this question we can calculate the amount of fully covered methods and instruction coverage over the executed methods. Our data demonstrates that, while the median method coverage was 7.3%, only

0.7% of all methods were fully covered (i.e., only 9.8% of executed methods were fully covered, figure 5.6a). More broadly, on average, instruction coverage of *executed* methods was 64% (figure 5.6b). This experiment quantifies the difference in precision when analysing coverage data based on diverse metrics. Thus, if precision is important, more precise instruction coverage can be recommended. Moreover, the fine-grained instrumenting technique itself opens opportunity to develop other, more sophisticated types of code coverage such as path coverage.

Conclusion: Despite the high correlation of instruction and method coverage metrics (94%), method coverage, so popular in closed-source testing, does not give the full picture and underestimates covered instructions (at 36% per executed method, on average), whereas instruction coverage is precise. In meanwhile, amount of fully covered methods is dramatically low. In line with the findings by Yang et al. [YHH19], this small study shows that the instruction coverage provides better precision than coarser-grained coverage metrics. This can be especially important when identifying features [XBFO19] or targeting specific parts of code such as API calls. However, as the absolute values of these metrics are highly correlated, it is acceptable to use different coverage granularities interchangeably in the automated testing.

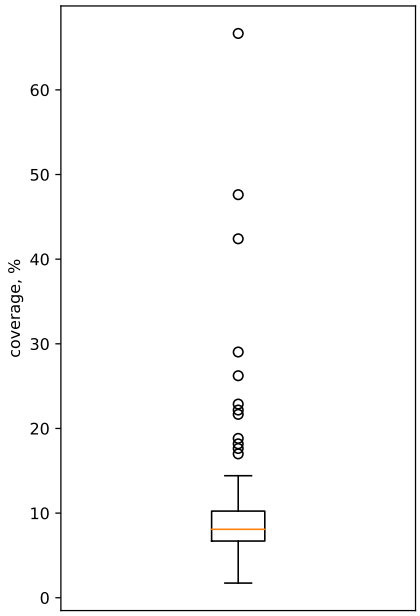
5.3 Discussion

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.

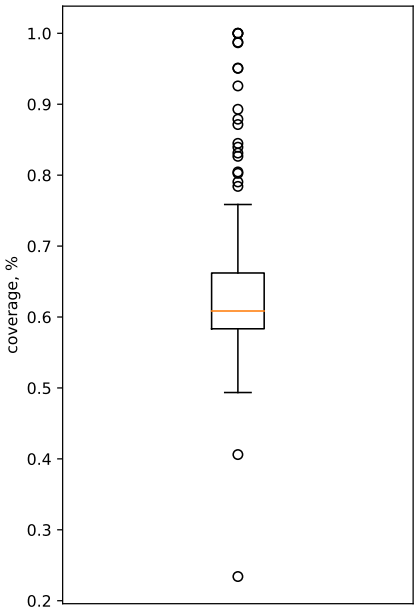
In this section, we discuss limitations of the tool design and current implementation, and summarize the directions in which the tool can be further enhanced. We also review threats to validity regarding the conclusions we make from the Sapienz experiments.

5.3.1 Limitations of ACVTool

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



(a) Fully covered methods over executed methods.



(b) Instruction coverage over executed methods.

Figure 5.6: Instruction coverage of methods (for each executed method in our F-Droid dataset).

Limitations of the ACVTool design An inherent limitation of our approach is that apps must be first instrumented before their code coverage can be measured. Indeed, in our experiments, there was a fraction of apps that could not be repackaged and instrumented. Furthermore, apps can employ various means to prevent repackaging, e.g., they can check their signature at the start, and stop executing in case of a failed signature check. Moreover, as shown by the experiment reported in Section 5.1.3, the repackaging step may inhibit the usage of Google APIs. Still, this limitation is common to all tools that instrument applications (e.g., [ZPG+15, ELL16, HCLT15, YH15, LWD+17]). 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., [HCLT15, ZPG+15]). Therefore, ACVTool is practical and reliable. We examine the related work and compare ACVTool to the available tools in Section 4.6.

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

is also well-maintained, and new improved versions are released regularly.

We assessed that the code coverage data from ACVTool is compliant to the measurements from the well-known JaCoCo [JaC18] and ELLA [ELL16] tools. We have found that, even though there could be slight discrepancies in the number of instructions measured by JaCoCo and ACVTool, the coverage data obtained by both tools is highly correlated and commensurable. Therefore, the fact that ACVTool does not require the source code makes it, in contrast to JaCoCo, a very promising tool for simplifying the work of Android developers, testers, and security specialists.

Limitations of our instrumentation approach One of the reasons for the slight difference in the JaCoCo and ACVTool measurements of the number of instructions is the fact that we do not track several instructions, as specified in Section 3.1. Technically, nothing precludes us from adding probes right before the "untraceable" instructions. However, we consider this solution to be inconsistent from the methodological perspective, because we deem the right place for a probe to be immediately after the executed instruction. In the future we plan to extend our approach to compute also basic block coverage, and then the "untraceable" instruction aspect will be fully and consistently eliminated. Alternatively, ACVTool can be enhanced by introducing a lightweight static analysis at the `smali` code level for a control flow graph-aware instrumentation [HWHH18].

Another limitation of our current approach is the constraint of 256 registers per method. Our instrumentation approach introduces 3 new registers. This register manipulation technique is safe as long as the total number of registers in the original `smali` method is less than or equal to 256. The only problematic instruction in this respect is `aput-boolean`, which can access up to 256 registers. While this limitation could potentially affect the success rate of ACVTool, we have encountered only one app, in which this limit was exceeded after the instrumentation. This limitation can be addressed either by switching to another instrumentation approach, whereby inserting probes as specific method calls, or by splitting big methods. Both of the approaches may require to reassemble an app that has more than 64K methods into a multidex apk [Goo17b]. We plan this extension as future work.

Taken to extremes, insertion of probes may potentially lead to issues. It is not clear what is the limit to the amount of instructions in a single app method and whether this limit can be reached by increasing the total number of instructions by a factor of 4. We have not encountered such cases, but this aspect may be worthy of further investigation in case of testing very complex applications [ZLZ+16].

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 exam-

ple, the very resource-intensive computations performed by the PassMark benchmark app degrade the CPU utilization by 27% in the instruction-level instrumented version. This is a critical scenario, and the overhead for an average app will be much smaller, which is confirmed by our experiments on real apps.

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

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

5.3.2 Threats to Validity

Our experiments with Sapienz reported in Section 5.1 allow us to conclude that black-box code coverage measurement provided by ACVTool is useful for state-of-art automated testing frameworks. Furthermore, these experiments suggest that it is necessary to better study the impact of coverage data for achieving time-efficient and effective bug finding.

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

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

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

that on a larger dataset we will obtain different results in terms of number of unique crashes and their types. A significantly larger experiment involving thousands of apps could lead to more robust results.

Second, Sapienz relies on the random input event generator Monkey [Goo18d] as the underlying test engine, and thus it is nondeterministic. It is possible that this randomness may have influence on our current results, and the results obtained in another experiment will show a clear edge for some coverage granularity.

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

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

As reported in Section 5.1.3, we have tried to address this threat by comparing application behaviors on original and instrumented app versions, and by investigating the crashes. We have shown that ACVTool does not change the app behavior, as visible in the GUI. However, we are yet not able to automatically confirm that ACVTool does not introduce crashes at runtime. Unfortunately, the publicly available Sapienz version does not support crash reproducibility. In the future, we consider to systematically evaluate reproducibility of found crashes across the original and instrumented app versions using tools like RecDroid [ZYS+19], CrashScope [MLVBC+17] or Paladin [MHH+19].

Finally, our findings may be affected by the experimental set-up. We run Sapienz with Android emulators, which are not fully representative of real devices and may introduce some stability issues that can result in app crashes [AGH+18].

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 [SMC+17], AimDroid [GCL+17] or QBE [KSM+18] 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 Google Play for our experiment. While preparing a delivery of an app to this market, developers usually apply different post-processing tools, e.g., obfuscators and packers, to prevent potential reverse-engineering. Some crashes in our experiment may be introduced by these tools. In addition, obfuscators may introduce some additional dead code and alter the control flow of apps. These features may also impact the code coverage measurement, especially in case of more fine-grained metrics. Therefore, in our future work we plan to also investigate this issue.

Chapter 6

Dynamic Binary Shrinking

In this Chapter, we present a novel approach, that allows a user to review app functionality and leave only tested code. The shrunk app produces 100% instruction coverage on observed behaviors and in such a way guarantees the absence of unexplored, and therefore, potentially malicious code.

Contents

6.1 Running Examples

In this section, we study the behaviors of two apps. One of them is a toy example where we purposely injected a hidden time bomb. Another one is the real-life Twitter Lite app. We explored both apps to find out how much of the code is executing in the app. We further give present the interpretation of our measurements.

6.1.1 Time Bomb Sample

The goal of the Time Bomb sample is twofold. First, it gives us an understanding of the exact amount of code run under a complete test suite. Second, further, in Section 4, we demonstrate in detail how our approach eliminates hidden, potentially malicious functionality out of the app.

We based our sample on the available at Github Java example of a time bomb [Rab16]. To build the app, we used Android Studio 3.5.2, build tools 29.0.2 with targeted minimum Andriod SDK 25, and enabled androidx[1]. We also enabled all available optimizations to minify extra code as much as possible. The optimizations included Proguard [Gua20] resource shrinking, code optimization, and minification and R8 aggressive optimizations in the full mode [Goo20b]. Optimizations helped to drop the APK size from 1328KB to 784KB (41% smaller), while the compressed binary code (DEX) file located inside the APK decreased from 1879KB to 262KB (86% smaller).

Indeed, our app is straightforward. It contains only one activity and a timer configured for one day from the app's build time. When launched, the app displays a text indicating one day left before the time expires (*day 0* functionality). However, the next day, when the timer is over, the app triggers an alert (*day 1* functionality). We further use the terms *day 0* and *day 1* to specify the app states before and after the time bomb explosion respectively. We also declare *day 1* functionality as hidden, since it stays unexpected after exploration on the *day 0*. Thus, we have an optimized time bomb app that explodes on the *day 1*.

Exploration

Since our app has no buttons or other elements to interact with, and we are fully aware of its functionality, we developed a short but complete test routine intending to achieve peak coverage. Our test suite combines the following set of actions: launching the app, hiding the app and bringing it to front, switching between the apps, changing a device orientation, stopping the app with a close button, stopping it with a swipe and with the *Clear all* button, launching the app again, taking a screenshot, changing volume, tapping, swiping. Though we did not declare most of the listed actions in the

[1] https://developer.android.com/jetpack/androidx

app, Android Application Framework libraries may have default listeners to some of them. In this scenario, an attacker targets to demonstrate benign behavior and pass our testing routine on *day 0*. We now report instruction coverage measured in testing on *day 0* and *day 1*.

When testing the app, we measured instruction coverage with the help of ACVTool. It provides us with information on exact executed instructions (this becomes essential in Section 6.3). ACVTool marks executed instruction through the whole codebase in the app's DEX file (including libraries). It first instruments the app in `smali` representation and then tracks executed instructions at the run-time. After the end of exploration, ACVTool generates an instruction coverage report. In this work, we used ACVTool capability to measure instruction coverage at any time without stopping the exploratory procedure.

In our experiments we managed to reach instruction coverage peak at 15.22% and 14.94% on *day 0* and *day 1* respectively. Remarkably, the app demonstrated lower code coverage on *day 1*, though the main package (the code was written by us, excludes libraries) showed the opposite picture — instruction coverage is higher on *day 0* with 52.63% against 72.37% on *day 1*. Furthermore, we observed more UI elements on *day 1* because the time bomb functionality runs an alert on top of the existing page.

However, a closer look at coverage differences revealed that on *day 1* some methods from Android Application Framework libraries did not run compared to *day 1*. These methods are `dispatchKeyEvent`, `dispatchTouchEvent`, `onWindowFocusChanged`, `dispatchKeyEvent`, `onKeyDown`, `setBackgroundDrawable`, `onBackPressed`. This happened due to the time bomb alert message popped out over the main page and blocked default event listeners on the page. We could not identify some other framework methods due to their obfuscation. The difference in the main package only revealed two different branches that executed with regard to the current time.

Although we revealed the hidden feature on *day 1*, the observer could not predict it from *day 0*. Moreover, more than 80% of the app code stayed not tested and we (as app developers) could not guarantee the absence of another logic bomb in this code.

6.1.2 Twitter Lite App

For the second, more complex example, we took a real-life case — a popular social network app Twitter Lite (version 2.1.2). This is the secondary official Twitter client that targets lower-cost devices limited on storage and performance. Compared to the main official Twitter app, the Lite version has a much smaller size: 1.3Mb against 31.8Mb of the main app. Twitter Lite is a popular app with over 10 million installs and 4 stars rating on

Google Play. This app is convenient for us due to its relatively generalized and straightforward architecture. In particular, its architecture simplifies our exploratory procedure in an attempt to reach peak instruction coverage. We further describe Twitter Lite features and our results on observing its behaviors.

App internals

Twitter Lite is a typical example of a lightweight client app for network communication. Along with Twitter, other major companies, targeting emerging markets, provide Lite versions of apps to their users. To mention a few popular, Facebook Lite, Messenger Lite, Instagram Lite (temporarily rolled back to date), LINE Lite, Skype Lite, Pinterest Lite.

Lite apps make use of a single trick to achieve lightness. An app embeds a web browser that loads and displays web pages from the dedicated website. The embedded browser has no regular address field nor navigation buttons. Thus, the app is often indistinguishable from an ordinary native Android app.

Twitter, as well as most of the other Lite apps, runs the standard Android component WebView [Goo20c] to vest the app with browser capabilities. This component allows Twitter to display web page content through the embedded browser, but also it provides JavaScript APIs to interact between the web page, app code, and Android system. WebView often simplifies developers' work because they can adapt original website pages to fit in a mobile app without significant changes in their website logic. This may reduce development costs compared to native apps. Second, it is possible to instantly roll out updates for the app on the website without undergoing the whole update procedure in the app market. However, the main advantage when using WebView is the small size of the app.

In our analysis, we found that the app was compiled using a standard Android `dx` compiler, but the code was obfuscated and minified (most likely, Proguard [Gua20] was used). The app makes a check if a debugger is connected and collects device information. Developers often use such techniques for anti-debugging purposes [Fou20]. In addition to the standard Android Framework libraries, Twitter Lite also uses such popular libraries as retrofit2, okhttp3, crashlytics, firebase, and gson. AndroidManifest file declares permissions from several categories. In the *Photos/Media/Files* and Storage permission categories: read, modify, and delete the contents of USB storage. From the "Other" category the app declares permissions to receive data from the Internet, view network connections, obtain full network access, run at startup, and prevent the device from sleeping[2].

[2]Google Play also gives permissions information at the app's page https://play.google.com/store/apps/details?id=com.twitter.android.lite

Functionality

The Twitter Lite app's major features include posting a tweet, finding and following another account, bookmarks, home timeline, notifications, profile settings, and direct messages. Twitter hosts a detailed description of Twitter Lite functionality on its website [Twi20].

This functionality is available through the embedded WebView that takes full display space. Therefore, a user mostly interacts with web page content. In turn, the web page calls appropriate methods in the app and Android APIs.

Exploration of behaviors

To explore Twitter behaviors, we used three automated testing tools, and we explored the app manually. Monkey [Goo18d], our first tool, is a popular, efficient and integrated into Android automated testing tool that randomly pushes UI events into an app. Second, Droidbot [LYGC17] is a more advanced tool based on Monkey but reinforced with the use of UIAutomator [Goo19]. Droidbot recognizes particular UI elements and combines its actions based on this knowledge. In particular, Droidbot allows users to specify the login page id of the app and credentials to pass app authentication. The third tool is DroidMate-2 [BJHZ18]. This state of art tool targets specifically 3rd-party apps. Like Droidbot, DroidMate-2 uses UIAutomator, but it provides additional test strategies and even allows anyone to develop a new. Last, we tested the app manually, which allowed us to run as much functionality as possible consciously.

Instruction coverage on observed behaviors

Figure 6.1 demonstrates the results of instruction coverage measurement when performing automated and manual exploratory procedures.

Our exploratory procedure worked as follows. Instrumented by ACVTool Twitter Lite app was installed on Android emulator (API 25), each automated testing tool worked for 1 hour as well as manual exploration performed by one of the authors. During this procedure, ACVTool generated instruction coverage every 5 seconds (in total, 720 values per hour).

We achieved the best results by manual exploration: 19.65% instruction coverage. Automated tools performed differently and demonstrated the following results: 14.08% Droidbot, 13.82% Monkey and 13.78% DroidMate-2. We ran the experiment twice and received very similar results. Among automated testing tools, Droidbot performed better, though it reached the peak slower. In the case of DroidMate-2, the tool managed to finish exploration in less than 3 minutes since it could not find more states (short red line appears in Figure 6.1).

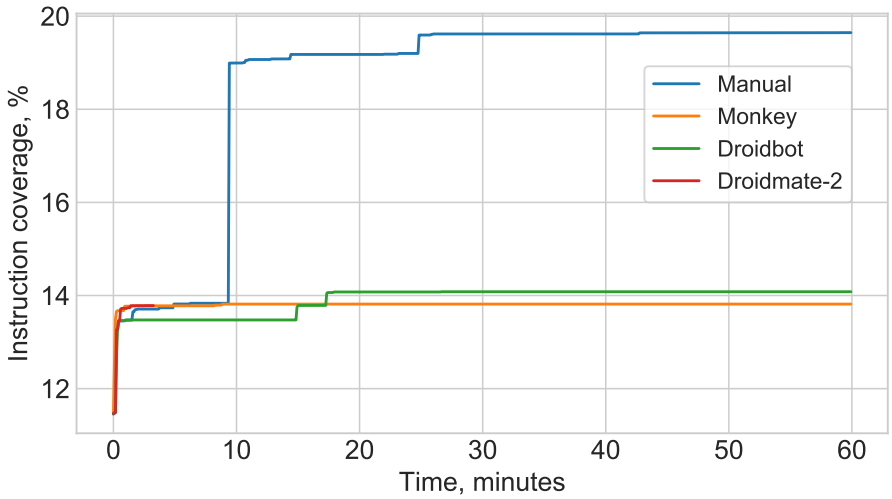


Figure 6.1: Twitter Lite full instruction coverage under automated (Monkey, Droidbot, DroidMate-2) and manual explorations.

Such a big difference in instruction coverage between the automated exploration and manual has its reasons. The main obstacle for automated tools was to sign up and log in. Though Droidbot and DroidMate-2 can to pass the login page in native apps (credentials need credentials need pre-configuration), the issue remains unresolved in the case of WebView apps, such as Twitter Lite.

Remarkably, both Droidbot and DroidMate-2 use UIAutomator under the hood. It helps tools to locate UI elements and extract their attributes. Therefore, Droidbot and DroidMate-2 are capable of recognizing login and password fields with specific identifiers. However, this is not the case in the WebView app. Here we confirm that the current implementation of UIAutomator is not able to pull field attributes out of a web page. Thus, both tools are not capable of passing either login or sign up pages in the WebView. This issue prevents automated tools from efficiently exploring such apps since a significant amount of functionality lies on the authorized side.

Indeed, during manual exploration, when passed the authorization, instruction coverage is significantly higher compared to automated tools. Coverage increased on more than 5% when tweeting a picture that we instantly shot with the camera, while sign up and login functionality took less than 0.3% of the codebase. Worth noting that after half an hour of exploration, we could not find more actions to improve overall coverage. Therefore, we continued to interact with the app more randomly. However, *the peak coverage we managed to achieve is less than one-fifth of the number of instructions*

in the app.

The Twitter Lite app experiment, in particular, demonstrated that neither absolute value nor relative increase of code coverage gave additional information on what or how to test more intensively. We could not also estimate the total amount of runnable code in the app. Our experiment further raises questions. What does the rest of the app (more than 80% not executed)? Can we guarantee goodness of not executed code?

6.1.3 Interpretation

Many officials and even states leaders nowadays use Twitter. Potentially they may become a target to an attack through the app on their devices. An attacker may hide malicious functionality in the app that can trigger under specific conditions (logic bomb).

Indeed, Frantantonio et al. in their work [FBR+16] described logic bombs as a commonly employed by targeted malware mechanism that may be used in state-sponsored attacks. From the dynamic analysis perspective, we could not find the logic bomb. Moreover, on our samples, the percentage of instruction coverage did not give more understanding of time bomb behavior since on *day 1* it did not differ much from *day 0*. When measuring only the main package (code written by us), not covered code is present on both *days*.

Although we put in our Time Bomb app as little functionality as we could, while the Twitter app is quite complicated app, instruction coverage on both apps was under 20%. Furthermore, for the Time Bomb app, we used Proguard and aggressive R8 shrinking — modern tools that use static analysis to eliminate dead code and compress apps.

The experiments brought us to the conclusion that app instruction coverage depends on app functionality, including libraries and optimizations techniques. Hence, *the instruction coverage metric is not comparable between the apps*. In this case, testing techniques that rely on absolute code coverage values may not scale on different apps.

Moreover, the *maximum coverage value* for 3rd-party apps is always *unknown* due to the absence of requirements or a *complete test suite*. Code coverage, therefore, does not help to clarify when to stop testing too. This is a known issue in automated testing approaches, when authors suggest to use certain thresholds such as time limit, certain level of code coverage or code coverage increase, number of total test cases [MMM14, MMP+12, NH19].

From the examples mentioned above, we can see that not only imperfection of automated tools contribute to the inability to reach 100% code coverage, nor dead code, but also the *extra code from libraries or written by app developers stays never executed*. This extra code constitutes the app's *bloat*, and thus, unnecessarily increases the attack surface [AJWK+19].

6.1.4 Implications on the App Distribution Model

With regards to the obtained results, in this work we propose a novel technique on shrinking Android apps towards their declared capabilities. Although we target this technique onto security analysts, so they could test and monitor the app and remove not executed code as potentially malicious.

To make it possible, before installation on the targeted device an analyst has to explore the functionality of a new app. When exploring, the analyst should aim to run the *declared by the app functionality* that the end-user supposed to use. During this procedure *monitoring of sensitive resources* is essential to ensure no malicious actions happening. Further, the app undergoes a shrinking procedure towards executed code.

It is clear, such manual exploration is expensive. However, the effort is justifiable against targeted attacks on specific authority (e.g. when the end-user is a head of state).

Yet, there is another option to protect users, which requires action from app producers. Those app producers willing to run their apps in a trustful authority environment would shrink the app at the delivery stage and provide a set of tests covering the declared functionality. This way the app would become verifiable and could demonstrate 100% instruction coverage on the provided test. Thus, an analyst could make sure that there is no potentially malicious code remained in the app. This approach, however, may impose a few limitations such as to restrict the use of dynamic loaded code and reflection mechanisms since dynamically loaded code may be potentially malicious. In future work, we may study vulnerabilities that shrinking may impose on the app.

6.2 Methodology

Figure 6.2 demonstrates a high-level overview of the Dynamic Binary Shrinking System for Android. Using this approach, security specialists and test engineers may monitor an app for malicious activities while exploring its behaviors. When the exploratory procedure is finished, our tool, ACVCut, shrinks the app towards covered code. Thus, only executed code may remain in the shrunk version of the app.

The core of such a system is fine-grained code coverage. In this work, we use ACVTool [PGD+18, PGZ+20] — the only available fine-grained code coverage tool for Android. ACVTool provides code coverage information at the level of instructions, methods and classes. Full and precise information about executed instructions is essential for us since substantial modifications of a 3rd-party app may easily break it.

The Dynamic Shrinking System consists of two phases. The first phase aims to observe and analyze all relevant behaviors, while in the second phase,

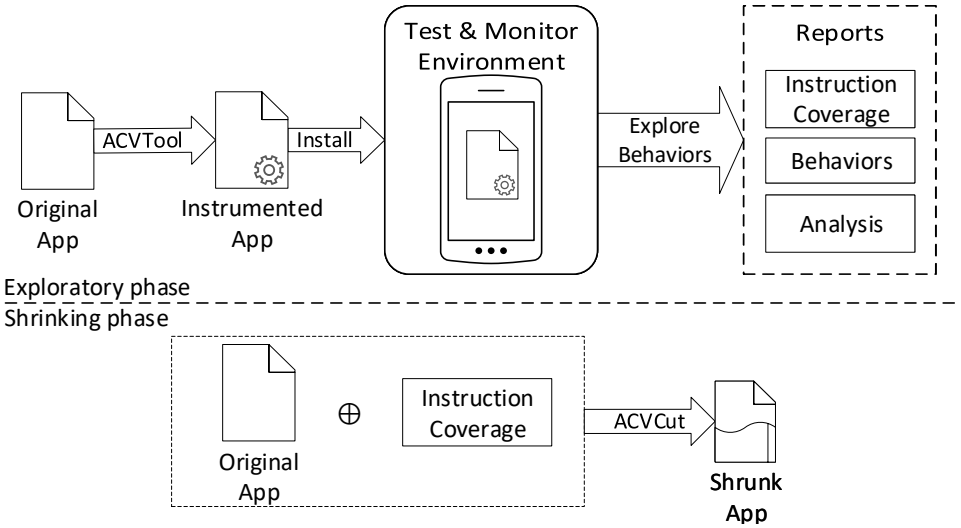


Figure 6.2: Dynamic Binary Shrinking System for Android.

ACVCut shrinks the app. We further describe them in depth.

6.2.1 Exploratory Phase

During the exploration procedure, an analyst (or an automated tool) observes 3rd-party app functionality. The analyst may want to leave only specific behaviors or to test the app exhaustively. Thus, depending on the analyst requirements, the app may lose not essential features when shrunk. Therefore, in the best scenario, the app provider willing to pass checks with all behaviors may present a complete test suite for their app.

Software development companies with full development cycle maintain requirements documentation, unit-, integration and regression tests, mock objects, and manual test cases with a simple goal — to check all required features. We consider these testing artifacts as a complete test suite.

This approach is beneficial to both sides since the app provider is interested in delivering a qualitatively tested app. At the same time, the analyst needs to make sure that the app does not perform malicious actions. Therefore, an app provider could shrink the app himself and deliver it together with the complete test suite, so that anyone could verify the app on 100% coverage.

The exploratory phase from a technical perspective goes as follows. The original APK gets instrumented by ACVTool to allow code coverage measurements. Then we install the instrumented app onto a device or emulator. An analyst may enable dynamic analysis tools to monitor which sensitive actions the app performs. Thus, by the end of testing, we may obtain three outputs: instruction coverage, explored behaviors and app activity analysis

reports.

From the obtained reports, instruction coverage is generated by ACVTool and goes as an input to the Shrinking phase. The behaviors report represents a complete test suite to the next generated shrunk version of the app. The analysis report aims to confirm the absence of malicious activities that happened under the exploratory phase. The analysis report relies upon *Monitor Environment* used to track suspicious activities.

This work aims to present a shrinking approach and its applications, which brings Dynamic Binary Shrinking System to life. Therefore, the monitor environment is not the goal of this work. However, monitoring is an essential part of the system to confirm that the shrunk app is benign. We suggest using state-of-the-art approaches to monitor sensitive APIs and HTTP requests. Further, the shrunk app loses a significant part of the code. Therefore, existing static analysis approaches may benefit from significantly smaller codebase available for analysis.

6.2.2 Shrinking Phase

The instruction coverage report specifies the exact code executed during the testing phase. Therefore, we can remove all not tested functionality. In a nutshell, we match the original smali code on our code coverage report and, through peculiar code manipulations, remove all unnecessary functionality. Such an operation is not trivial and requires attention to specific Android bytecode architecture since we can easily break the app. We give details on `smali` code modification in Section 6.3.

6.2.3 ACVCut Implementation

ACVCut heavily relies on ACVTool under the hood, which performs a full cycle of app repackaging. ACVTool instruments the smali code, initiates the instrumentation process before the beginning of the exploratory procedure, saves runtime report at the end of exploration and finally generates code coverage report [PGD+18, PGZ+20]. Here we give more details on the design of ACVCut.

ACVCut starts form the preparation of the working directory with the help of ACVTool. The directory should include the instrumented by ACVTool app, instrumentation report and the decompiled app directory. The instrumented app takes its part in the exploratory procedure and produces the runtime report. Runtime reports contain instruction coverage information in the form of binary arrays and, when applied to instrumentation reports, produce tree-based smali code structure, where executed instructions are marked as covered. ACVCut walks through the tree in several passes to determine and remove not-executed basic blocks and remove or make *stubs* from not called methods. *Stub* methods removal may break the app due

to its structural role, e.g. they may take part in the inheritance hierarchy. However, *stubs* have no executable code and never called. Therefore, they should be excluded from all types of analysis. When code processing is finished, ACVCut builds the shrunk app with the help of `apktool`. Finally, the output of ACVCut contains the shrunk app and a list of neutralized *stub* methods.

6.3 Cutting the Code

In this section, we describe our approach to removing the not-executed `smali` code from an app. The approach performs an extensive `smali` code processing and even a lightweight static analysis for solving challenges related to maintaining consistency, coherence, and cohesion of complex compiler-generated code. We first get rid only of instructions in executed methods and further remove not executed methods.

6.3.1 Instructions

The smallest executable unit of an app is an instruction. In this work, we rely on `smali` representation of binary executable since it gives us the possibility to modify the app binary at the finest granularity.

Basic block

Besides the fact that ACVTool currently supports instruction- and method-level code coverage, we found it is more efficient to operate on `smali` basic blocks. We made this technical decision because ACVTool does not mark specific instructions when it is impractical or impossible to insert probes [PGZ+20].

Indeed, in Section 3.1 we mentioned the following cases. It is impossible to insert a probe after the `invoke-*` instruction followed by `move-result` and after a `catch` followed by `move-exception`. It is impractical to insert a probe after instructions `return`, `goto` and `throw`. ACVTool relies on execution of probes linked to specific instructions and by default it does not give coverage information about the above-listed instructions.

We focused on basic blocks since, normally, instructions in a basic block either executed all or none. This strategy works well with the only special case when one instruction throws an exception. Then, the instructions next to the failing instruction stay not executed. This case is easy to find by checking if the last instruction of the basic block worked. Further we describe our study on basic block definition for `smali` representation.

Basic block contains a sequential branch-free set of instructions that starts with a labeled operation and ends with a brunch, jump or predicated

operation [CT11]. In `smali` representation, we developed the following rules. A basic block may start with the first instruction in the method, a label or `if-*` instruction. The basic block ends on the last instruction before the beginning of the next basic block, `goto` instruction or the end of the method.

ACVCut gets rid of all not executed basic blocks. Removal of a basic block means sequential removal of instructions from it and removal of corresponding labels. In our implementation, we extended the definition of a basic block to a *labeled block* since all instructions between two labels are always covered or none of them. However, basic block removal is still a challenging task since executed branch instructions (`if-*` or `goto`) may stay in place and reference to the deleted basic block label.

For instance, statement `"if-eqz v7, :cond_0"` compares the `v7` register value to zero. When the condition was satisfied, a program pointer jumps to the label `:cond_0`. Otherwise, the program pointer passes the statement to the next instruction. ACVCut may remove the basic block corresponding to the specified label (`:cond_0`) when they were not executed. In this case, the `"if-eqz v7, :cond_0"` statement becomes invalid since the target label declaration is absent now. To solve this issue, we analyzed possible cases and suggest code manipulations that eventually modify the original program control flow.

Conditional jump logic

The issue of an absent label reference, mentioned above, may occur to all instructions able to reference a label: `if`, `goto` and `.catch`. Since `goto` always finalizes the basic block and gets removed along to the not executed basic block, it does lead to the issue in our implementation. However, `if` and `.catch` statements need a more sophisticated approach. We describe a solution to the `if` statement that references an absent label and further explain the `.catch` case in the next *Try-Catch* paragraphs.

`If` instruction in `smali` representation has the same meaning as in Java code. It has four combinations depending on the coverage of if instruction itself and the coverage of the referenced by the specified label basic block. Concerning the case, we do the following.

We check each targeted label specified for the executed `if` statement. If the label does not appear in the code anymore, ACVCut removes the `if` statement since it did not perform conditional jump during the test.

The opposite case is when ACVTool did not mark the `if` statement. It does not always mean that the statement did not execute. Indeed, the statement could work and make a jump to the specified label. Therefore, the probe placed next to the statement did not run, and ACVTool could not mark the appropriate statement. In this case, we check if the previous instruction run. This means to us that the program pointer reached `if`

statement and performed a conditional jump to the specified label. Since only one branch worked, we replace the statement and the rest of the basic block with the unconditional `goto`.

Try-Catch

The exception mechanism in `smali` representation relates to Java try-catch structure. The monitored block starts and ends with `:try_start` and `:try_end` labels correspondingly and followed by the `.catch` and `.catch-all` meta-instructions. They first declare the type of an exception try-catch block catches, second — where the monitored code starts and ends (labels `:try_start` and `:try_end`) and third — the target label where to handle the exception.

When the try-catch is covered, we check exception handlers mentioned in the `.catch` meta-instruction. If the code did not throw the specified exception, then the corresponding handling basic block did not work, too, and was removed in previous steps. Therefore, we remove the `.catch` as well. If no `.catch` meta-instructions left in try-catch, we remove `:try_start` and `:try_end` labels too. Thus, if the code did not throw an exception, it would continue working smoothly.

The opposite, when the code throws an exception, the try-catch and handling code stay in place. Moreover, an instruction that threw an exception is not marked as covered (since the probe next to it was not executed), but it has to stay in code and perform its function to throw the exception.

Synchronized code

Complex apps may deal with multiple threads that often need access to a shared resource. In this case, developers use Java keyword `synchronized` to allow one thread at a time to the selected code. Android implements this mechanism using a monitor object and a pair of instructions `monitor-enter` and `monitor-exit` to specify the beginning and the end of synchronized code.

Synchronization is sensitive to exceptions and can affect the work of a device. Hence, an app required to unconditionally catch possible exceptions in synchronized code and call emergency `monitor-exit` (contains two instructions). Android compiler takes responsibility for developers to generate the required exception handler with additional `monitor-exit` code, so that the app code could pass Android Runtime Verifier checks concerning the synchronized code.

In the same way, we need to follow the rules when modifying the `smali` code. When synchronized code worked without any exception, the generated try-catch can be removed. However, we leave the `monitor-exit` code in place since it helps the app to pass the Runtime Verifier. As a result,

`monitor-exit` code stays as an exceptional addition that may never be called.

Switches

The switch statement in Java represents a set of cases that may be chosen depending on the passed value. Android Compiler translates Java switch into either `packed-switch` or `sparse-switch` instructions depending on the value sparseness in the cases.

`Switch` instruction consists of two parts in `smali` . The first part declares the switch statement in the code, while the second part stays at the end of the method and contains a tuple of cases. Each case is a pseudo-instruction that references a label attached to the basic block to handle a specific case. The whole second part cannot be tracked with ACVTool. Therefore, we again take a close look at what labels stay in the code.

Since we removed not executed basic blocks, we also need to get rid of the corresponding pseudo-instructions that reference absent labels. However, the removal of a pseudo-instruction violates original switch table value allocation. Instead of removal of a case, we replace it with another working case in the original order. Though a switch statement may have a series of identical pseudo-instructions, its logic remains correct for the left cases. If no cases worked during the test, we would remove both parts of the switch instructions.

Arrays

Similarly to the switch statements, arrays statements also have two parts where the first part declares an array (`fill-array-data` instruction), and the second part keeps enumerated array values. When `fill-array-data` instructions disappeared from the code, the corresponding `.array` meta-instruction containing array values gets removed.

Merging gotos

As a result of our above-mentioned code manipulations, we noticed many cases when a `goto` instruction jumps right to the next line. This operation is equal to standard program pointer increment and happens due to the removal of basic blocks between the `goto` instruction and the label it jumps to. We remove such a `goto` instruction and the label if it has no other references.

6.3.2 Methods

Above, we described a sophisticated analysis of `smali` structures on instruction removal. We perform this approach only on called methods since other methods stayed untouched under exploration and, intuitively, should be deleted. However, the removal of not used methods is not an obvious task too.

Methods removal

Android picks up object-oriented features of Java, such as *encapsulation*, *polymorphism* and *inheritance* for its bytecode. Therefore, Android also has notions of abstract methods and classes, interfaces and virtual tables for inheritance. When calling the methods, Android uses the corresponding `invoke-*` instructions: invoke-virtual, invoke-super, invoke-direct and invoke-static

Even when not called, a method may constitute inheritance hierarchy or polymorphism taking place in the corresponding virtual table at run time. Such a method is usually called through `invoke-super` or `invoke-virtual` instructions that take into account object type hierarchy at run time. Predicting the correct run time type based on the code coverage and class hierarchy and altering method calls accordingly is not a part of this study. We see it as an exciting field of research from the perspective of bytecode optimization. However, this issue does not relate to *static* and *direct* methods, and we can remove them for sure.

Stub methods

The rest of the methods may stay in the app since some of them maintain an object-oriented skeleton of the app, as we have mentioned earlier. However, here we do the following trick. We first remove all the instructions inside the not called method and the second — put default return value according to the declared type. One particular case here is the constructor of an inherited class. We add a statement to call default `Ljava/lang/Object;-><init>V()` constructor, since the inherited constructor always calls the constructor the superclass.

Thus, the app has many empty methods that we call stub methods. ACVCut saves them into a list so that other tools could exclude them from their analyses.

6.3.3 Offensive Mode

When removed all extra instructions, some not explored functionality could disappear. Though we claim not tested code as potentially malicious, the

absence of legitimate features can be unexpected to the end-user. For instance, the user may press a not tested button. Then, the button will not react since now its method-listener is empty.

A friendly approach to address this issue is to bring into the app exception code to alert the end-user that the functionality was purposefully cut.

6.4 Results

In this section, we applied our shrinking technique on the running examples.

6.4.1 Shrunk Time Bomb

We shrunk the app on *day 0* to verify what happened with the time bomb behavior on *day 1*. To make the results more transparent, we also enabled instruction coverage measurement using ACVTool on the shrunk app and run the complete test suite we described in Section 6.1.1.

Table 6.1 demonstrates results on shrinking Time Bomb on *day 0*. When testing the app, we observed the same behavior, as we described in Section 6.1. Indeed, instruction coverage of the shrunk version shows 99.9% since we removed all the extra code. Additional benefits are smaller app size and less code. Thus, the decompressed binary DEX file decreased twice, while the amount of instructions, methods and classes greatly decreased in 9, 7 and 4 times correspondingly. App size changed slightly because the DEX file is saved in a compressed form, while app resources took the most of space.

Table 6.1: The original and the shrunk Time Bomb app metrics on Day 0

App Version	Coverage	Instructions	Methods	Classes	APK Size	DEX Size
Original	15.2%	58045	3002	388	784 KB	591 KB
Shrunk	99.9%	6342	431	102	701 KB	327 KB
Profit	-	x9	x7	x4	12%	x2

The app behavior remained unchanged on *day 1*. The time bomb functionality disappeared. However, there are cases when apps have similar benign functionality. If we wanted to keep both behaviors, we have two options. First is to test both behaviors separately and merge their instruction coverage before shrinking the app. Second, specific functions can be shortlisted to exclude their shrinking.

Not covered instructions. Worth to note why instruction coverage is very close to 100% but not precisely. ACVTool marked only 7 instructions as not covered. Four of them are necessary to respect runtime verifier rules concerning the synchronized code. Three others relate to the handled exception. We describe the reasons in Section 6.3.1.

6.4.2 Shrunk Twitter Lite App

On the Twitter Lite app, we performed the following experiment. We took instruction coverage generated by Monkey as an example of automatically explored behaviors keeping in mind that it did not pass the login page and did not see a significant part of the app. Since this is a WebView app, we expect that shrinking preserved the major part of functionality except for not observed functions that closely work with Android APIs. The goal of this experiment is to determine what behaviors continue to stay in the app and how the user interface (UI) responds to shrinking.

Table 6.2 presents changes that happened to the app after shrinking. Though the size of the app was already small compared to the full Twitter version, the app size decreased to 1036KB from 1327KB after shrinking. The number of instructions, methods and classes fell dramatically: 14, 7 and 3.5 times less than in the original app.

Table 6.2: The original and the shrunk Twitter Lite app metrics

App Version	Coverage	Instructions	Methods	Classes	APK Size	DEX Size
Original	13.82%	166031	11140	1680	1327 KB	1896 KB
Shrunk	97.9%	11198	1562	488	1036 KB	1757 KB
Profit	-	x14	x7	x3.5	28%	≈

Remarkably, the app preserved all web functionality that Monkey could not observe, such as logging in, tweeting, interacting with other profiles (follow/unfollow, like, comment, retweet), sending and receiving messages, profile editing, changing theme color, changing app language and other app settings. However, other not observed features, that rely on interacting with the Android code, disappeared. Indeed, the app lost notifications, sharing a tweet to other apps, sharing pictures and text from other apps to Twitter, posting pictures, changing user avatar and the background picture. Moreover, when tried to tweet a picture, we tapped on the *attach picture* button, the button animation worked. However, the standard dialog did not pop out to choose between the gallery and camera pictures. The same happened when we tried to change the user avatar and background picture.

Thus, when exploring behaviors before shrinking the app, we can decide

on what functionality we want to leave. This particular ability allows us to impose privacy requirements on the app. For instance, when a solid enterprise company tries to regulate the use of apps on employees' devices, it may allow them to access social networks but restrict posting sensitive pictures. Moreover, the app will never ask the user for a suspicious permission if the corresponding functionality was deleted. Thus, the user of the shrunk app has no chance to give a permission by mistake. For instance, our shrunk Twitter Lite app preserved most of the major features while restricted access to sensitive Android APIs.

6.5 Discussion

In this section, we shortly review existing threats and limitations that we may address in future work.

6.5.1 Threats and Limitations

Internal validity. Threats to internal validity relate to the implementation of our tool and discussed methodology on Dynamic Binary Shrinking System.

Although our tool processed a significant amount of obfuscated code represented in our samples, the tool may contain bugs and may break some apps. We will continue the development of ACVCut and evaluate it on more apps.

The stub methods continue staying in the code. However, we save them into a list of stubs to exclude from other analyses.

Our approach may shrink legitimate, however, not tested behaviors. Only app reviewer can decide which functionality to leave in the app. However, interested app producers can share their full tests addressing all the app requirements.

Android applications framework contains backward compatibility code that we shrink too. The app may be tested and shrunk for the device model, where it is expected to work.

External validity. We acknowledge the following threat to the generalization of our findings. A simple time bomb example does not scale on the possible diversity of logic bombs and attacks. Although we decreased the attack surface, an attacker may maliciously use existing app vulnerabilities to exploit benign app capabilities.

ACVCut limitations. Since our tool depends on the ACVTool, it inherits all its limitations. It is impossible to shrink an app when ACVTool was not able to generate instruction coverage. We refer to the original Section 3 for the limitations. These cases include apps that use hardening techniques to prevent repackaging with `apktool`, apps with multiple DEX files (ACV-

Tool currently works only with main DEX file), dynamically loaded code, native C/C++ libraries.

Another limitation of our tool is a steppy learning curve when troubleshooting errors that ACVCut may inject into apps due to yet unfamiliar compiler generated code semantics appearing in new apps. We commented on ACVCut development in Section 6.5.2. However, we are always happy to verify the tool in real life challenges and improve it towards everyone's advantage.

6.5.2 On the ACVCut Development

The development process of ACVCut is far from trivial. Unfortunately, `smali` language suffers from a lack of documentation and the absence of code practices commonly available for ordinary programming languages. The language itself is pretty low-level; it is an assembler. No wonder its learning curve is pretty steep. These limitations badly contribute to ACVCut development because it requires a rare skill-set that would allow debugging potential errors injected by ACVCut into a 3rd-party app.

Doing this work, we transformed many observed `smali` code cases and rules (see Section 6.3.1) into an automated tool. Taking into account the flaws mentioned earlier, the development of the tool became more evolutionary. Although ACVCut worked well on the running examples (Section 4), it may break some other apps. We will continue to generalize our code processing approach along with shrinking more apps.

We also improved the correctness of instruction coverage generated by ACVTool. Notably, we fixed coverage calculating for `.catch` meta-instructions (incorrect coverage may inject errors into ACVCut). We also fully reworked the Instrumentation class to allow on-demand coverage reporting at any time. Moreover, ACVTool can now track instructions even on the app termination since we keep the instrumentation process running all the time. These updates are essential for our tool because apps are sensitive to careless code manipulations and break easily. When coverage information, for instance, is absent on app termination, the shrunk app would be crashing on each app exit. We will pull request these updates into the original ACVTool repository after the acceptance of this work.

Impact on mutation testing

To the best of our knowledge the only work, called MutAPK [EVLVB+20], recently suggested a limited set of Android-specific mutation operators for closed-source mutation on `smali` representation [EVLVB+20]. The authors of MutAPK assume that `smali` mutation speeds up the testing process comparing to traditional source code mutation. Our detailed analysis on `smali` modification will help to extend existing for `smali` basic mutation opera-

tors, e.g. our technique allows to remove statements and basic blocks and branches, and thus, to modify the app's control flow. Moreover, the shrunk version of the app is itself can be considered as a mutant that is going to survive and demonstrate 100% instruction coverage.

6.6 Related Work

Program debloating is in the closest relation to our approach since it describes program thinning and the removal of specific features. It found applications in such areas as debloating of libraries, source code, containers, hardware, and in delta debugging. Debloating techniques emerged in several recent works and available now for various platforms to reduce programs with and without source code.

Brown et al. remove features from software based on source code to feature mapping [BP19]. Azad et al. remove complete features in web applications by using a PHP profiler to extract code coverage information [ALN19]. Heo et al. developed the Chisel system to effectively customize C programs with the help of reinforcement learning [HLPN18].

Agadakos et al., in their recent work, developed Nibbler, a tool for bloat removal from binary shared libraries. Nibbler works on x86 binaries, does not recompile the program and relies on static analysis to eliminate not reachable code [AJWK+19]. Qian et al. developed a RAZOR framework that use control-flow heuristics to predict and leave more user-expected functionalities in Linux binaries [QHA+19]. Landsborough et al. make software thinner with two approaches: first relies on dynamic tracing as a guide while the second uses a genetic algorithm to mutate a program [LHF15]. The authors also argue on the pros and cons of thinned programs from security, size, validity and optimality perspectives.

In Android, researchers rely on static analysis to remove dead code. Jiang et al. proposed JRed and RedDroid approaches to remove bloat from Java applications, Java Runtime and Android apps [JZWL16, JWL16, JBW+18]. Proguard, a Gradle plugin for shrinking Android apps, performs static analysis to cut dead code [Gua20].

While static analysis and debloating are well researched, to the best of our knowledge, Android did not have a solution for thinning apps based on the app execution traces or code coverage.

Chapter 7

Conclusions

We conclude this dissertation and discuss potential future research directions in this Chapter.

Contents

7.1 Conclusions

In this work we presented a novel approach to measure fine-grained code coverage of Android apps without access to their source code and a methodology to shrink an Android app towards executed code while observing the app's functionality.

More specifically, we organized this dissertation in three parts: 1) An instrumentation-based fine-grained code coverage measurement approach and its evaluation; 2) A study on the influence of code coverage granularity on automated testing; 4) A dynamic binary shrinking methodology.

Regarding the first part, our objective was to deploy a new reliable and versatile approach for code coverage measurement when testing apps without access to their source code. To support our approach, we demonstrated a high execution success rate (compared to the previous work) for the apps instrumented by ACVTool at the instruction level. Furthermore, we have established negligible runtime and instrumentation time overheads introduced by ACVTool, we reported on justified behavior changes and we confirmed the compliance of code coverage measurement to JaCoCo and ELLA tools.

Further, we performed a detailed study on the influence of code coverage metric granularity on automated testing. The study confirms the interchangeability of different code coverage granularity when used to guide automated testing tools for fault detection. Moreover, instruction-, method- and class-level coverage turned out to be highly correlated. This study demonstrated that the use of a coarser-grained coverage metric is not prohibitive in automated testing and suggested to choose the metric with respect to testing requirements such as possible limitations on the runtime overhead.

In the final part, we presented the Dynamic Binary Shrinking System - a novel methodology created for shrinking 3rd-party Android apps towards observed benign behaviors. On our running examples we demonstrated several findings. First, apps contain large amounts of not used code. Second, the measured coverage does not provide information on whether all legitimate app behaviors have been seen. The Dynamic Binary Shrinking System can guarantee the absence of potentially malicious code with 100% instruction coverage. We shrunk two apps and confirmed the viability of our approach. However, shrinking may lead to the disappearing of legitimate features if they stay not tested.

This dissertation, moreover, delivered two artifacts. First, the ACVTool — the implementation of our instrumentation-based fine-grained coverage measurement approach. The tool is our significant engineering effort that is available now to the Software Testing Android community. it has already gained interest from other research teams, which poured out into new articles [PDGCS20, YHH19]. Second, the ACVCut tool — an implementation of the Dynamic Binary Shrinking approach based on the instruction coverage gen-

erated by ACVTool. We confirmed the viability of our shrinking approach on two apps that remained functioning after shrinking, furthermore, they produced 100% instruction coverage on the explored behaviors.

7.2 Future Work

We can identify several novel research directions and future work extensions stemming from this thesis. One area of future research concerns possible improvements and extensions of the coverage measurement approach and the dynamic binary shrinking technique.

We plan to extend our code coverage measurement approach to instrument and report on the level of a basic block, which should make the system to be more efficient in terms of performance, and may also simplify the instrumentation approach, allowing to eliminate the "untraceable" instruction aspect. We futher consider to implement other more complex code coverage metrics such as path coverage. This will allow the analysts to have a better view on the observed functionalities.

Furthermore, our experiments with Sapienz have produced interesting conclusions that no coverage granularity is able to find all crashes, even in repeated experiments. We have also found negative results on the importance of coverage granularity, when used as a component of the fitness function in the 3rd-party app testing. ACVTool that works with most of the apps has uniquely enabled us to perform this coverage comparison study. Another line of future work for us is to expand our experiments with more testing tools, thus establishing better guidelines on which coverage metric(s) is more effective and efficient in bug finding.

One of the outputs of our experiment with Sapienz are faults. We consider in the future to systematically evaluate reproducibility of found crashes across the original and instrumented app versions.

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

To enable better support for the automated testing community, we are working to add support for multidex apps, extend the set of available coverage metrics to branch coverage, and to alleviate the limitation caused by the fixed amount of registers in a method. We will also investigate an option to store counters for each executed instruction, which will allow identifying

most and least executed code locations. As another promising line of future work, we will investigate on-the-fly dex file instrumentation that will make ACVTool even more useful in the context of analyzing highly complex applications and malware.

From the Dynamic Binary Shrinking part of this thesis we plan to continue improve our approach and validate it on more apps and use cases. The current ACVCut prototype demonstrated its viability on two examples. However, the use of this technique raises a few questions and opens many interesting directions for further studying. Although our technique worked well on our examples, the loss of legitimate behaviors may severely affect reliability and even introduce bugs and vulnerabilities. Therefore, we plan to investigate how shrinking changes the reliability of the app and if it introduces severe bugs and vulnerabilities.

In future work, we also plan to finalize building of the Dynamic Binary Shrinking System by integrating the monitoring and analysis tools (e.g. to monitor sensitive APIs and HTTP requests with possible use of analysis sandboxes [SFE+13, WYZ+15]). This will allow to validate such system on real logic bombs and other malware and propose the complete Dynamic Binary Shrinking solution to security analysts.

Furthermore, our shrinking techniques applies changes to the branching code structures such as `if`, `try-catch` and `switch`. We found absence of such operators that could significantly change the control flow of an app (e.g. removal of a branch as we do in shrinking). Therefore, in the future work, we may carefully study possible operators on `smali` representation and introduce a new set of operators for 3rd-party mutation testing.

Another prominent line of research is possible in collaboration with a real app distributor. App producers that maintain the full app development cycle aiming to consistently and in full cover functional requirements with automated integration and unit tests can benefit from our shrinking approach. With the test suites provided by the app producer, the shrunk app could demonstrate its full instruction coverage, which unlocks the app's verifiability and the guarantees on the absence of a logic bomb or a backdoor (with a few clauses) to the end-user, but also the test suites guarantee the remaining of planned functionality. Indeed, such a test suit may not be complete in terms of a combination of all possible paths, however, it would be complete in terms of designed features avoiding the loss of legitimate features. Therefore, in the future, we would like to study the possibility of maintaining such a development process and develop the guidelines that would allow production of verifiable apps in the shrunk shape.

Bibliography

[ABKT16] 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. doi:10.1109/MSR.2016.056.

[AGH+18] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with sapienz at facebook. In *International Symposium on Search Based Software Engineering*, pages 3–45. Springer, 2018.

[AJWK+19] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 70–83, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359789.3359823.

[ALN19] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: quantifying the security benefits of debloating web applications. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1697–1714, 2019.

[AN13] 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. URL: http://doi.acm.org/10.1145/2509136.2509549, doi:10.1145/2509136.2509549.

[ANKS16] Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Tan Hee Beng Kuan, and Charlie Soh. Achieving high code coverage in android ui testing via automated widget exercising.

In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 193–200. IEEE, 2016.

[AO16] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2 edition, 2016. doi:10.1017/9781316771273.

[ARB17] 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. doi:10.1109/MOBILESoft.2017.2.

[ATD+14] Yauhen Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaiping Liu, and Lwin Khin Shar. Empirical comparison of intermediate representations for android applications. In *SEKE*, pages 205–210, 2014.

[BBD+16] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th {USENIX} security symposium ({USENIX} Security 16)*, pages 1101–1118, 2016.

[BBG10] Bernhard Beckert, Daniel Bruns, and Sarah Grebing. Mind the gap: Formal verification and the common criteria (discussion paper). In *VERIFY@ IJCAR*, pages 4–12, 2010.

[BBS+17] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495. IEEE, 2017.

[Ben18] Ben Gruver. Smali/baksmali tool. 2018. URL: https://github.com/JesusFreke/smali.

[BGZ18] 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. URL: http://doi.acm.org/10.1145/3197231.3197243, doi:10.1145/3197231.3197243.

[BJHZ18] Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 916–919, 2018.

[BKLTM12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. IEEE, 2012.

[BKM+12] 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. `arXiv:1208.4536`.

[BLL18] Lingfeng Bao, Tien-Duy B Le, and David Lo. Mining sandboxes: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE, 2018.

[Bor08] D. Bornstein. Google I/O 2008 - Dalvik Virtual Machine Internals, 2008. URL: https://sites.google.com/site/io/dalvik-vm-internals.

[BP19] Michael D Brown and Santosh Pande. Carve: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 1–7, 2019.

[CAG+17] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. Exception handling bug hazards in android. *Empirical Software Engineering*, 22(3):1264–1304, 2017.

[CGO15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[Cle17] Mike Cleron. Android Announces Support for Kotlin, May 2017. URL: https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html.

[Cle20a] Jessica Clement. Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 2nd quarter 2020, May 2020. https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/. URL: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/.

[Cle20b] Jessica Clement. Statistica, August 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. URL: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[CML+17] 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.

[CMRY18] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2018.

[CNS13] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices*, 48(10):623–640, 2013.

[CPTH17] 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. doi:10.1109/ICSE.2017.61.

[CR17a] 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. doi:10.1109/ICSME.2017.35.

[CR17b] 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. doi:10.1109/ICSME.2017.31.

[CT11] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[DGPZ18] 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. URL: http://doi.acm.org/10.1145/3243734.3278524, doi:10.1145/3243734.3278524.

[DWZ12] 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.

[DZG+20] Stanislav Dashevskyi, Yury Zhauniarovich, Olga Gadyatskaya, Aleksandr Pilgun, and Hamza Ouhssain. Dissecting Android cryptocurrency miners. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 191–202, 2020.

[ELL16] ELLA. A tool for binary instrumentation of Android apps, 2016. URL: https://github.com/saswatanand/ella.

[EVLVB+20] Camilo Escobar-Velásquez, Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Patrick Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutant generation for open- and closed-source android apps. *IEEE Transactions on Software Engineering*, 2020.

[FBR+16] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE, 2016.

[Fou20] The OWASP Foundation. Mobile security testing guide. android anti-reversing defenses, 2020. URL: https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering.

[GCL+17] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü. 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. doi:10.1109/ICSME.2017.72.

[GGZ+15] 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. URL: http://doi.acm.org/10.1145/2660767, doi:10.1145/2660767.

[Goo17a] Google. Dalvik Executable format, 2017. https://source.android.com/devices/tech/dalvik/dex-format. URL: https://source.android.com/devices/tech/dalvik/dex-format.

[Goo17b] Google. Enable Multidex for Apps with Over 64K Methods, 2017. URL: https://developer.android.com/studio/build/multidex.html.

[Goo18a] Google. Dalvik bytecode, 2018. URL: https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[Goo18b] Google. smali, 2018. URL: https://android.googlesource.com/platform/external/smali/.

[Goo18c] Google. Test your app, 2018. URL: https://developer.android.com/studio/test/index.html.

[Goo18d] Google. UI/Application Exerciser Monkey, 2018. URL: https://developer.android.com/studio/test/monkey.

[Goo19] Inc. Google. UI Automator, 2019. URL: https://developer.android.com/training/testing/ui-automator.

[Goo20a] Google. Mobile device management system for android enterprise, 2020. URL: https://www.android.com/enterprise/management/.

[Goo20b] Google. Shrink, obfuscate, and optimize your app, 2020. URL: https://developer.android.com/studio/build/shrink-code.

[Goo20c] Inc. Google. WebView: A View that displays web pages, 2020. URL: https://developer.android.com/reference/android/webkit/WebView.

[Gua20] GuardSquare. Proguard manual, 2020. URL: https://www.guardsquare.com/en/proguard/manual/introduction.

[HBG+14] F. Horváth, S. Bognar, T. Gergely, R. Racz, A. Beszedes, and V. Marinkovic. Code coverage measurement framework for Android devices. *Acta Cybernetica*, 21(3):439–458, 2014.

[HCLT15] 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. doi:10.1109/MobServ.2015.38.

[HLPN18] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 380–394, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243838.

[HWHH18] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[HYWH15] Shang-Yi Huang, Chia-Hao Yeh, Farn Wang, and Chung-Hao Huang. Abca: Android black-box coverage analyzer of mobile app without source code. In *Progress in Informatics and Computing (PIC), 2015 IEEE International Conference on*, pages 399–403. IEEE, 2015.

[IDC20] IDC. Smartphone market share, 2020. URL: https://www.idc.com/promo/smartphone-market-share/os.

[Jac07] William Jackson. Under attack-common criteria has loads of critics, but is it getting a bum rap. *Government Computer News*, 8(13):07, 2007.

[JaC18] JaCoCo. Java code coverage library, 2018. URL: http://www.jacoco.org/.

[Jam18] Konrad Jamrozik. *Mining sandboxes*. PhD thesis, Saarländische Universitäts-und Landesbibliothek, 2018.

[JBW+18] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th In-*

ternational Symposium on Software Reliability Engineering (ISSRE), pages 189–199. IEEE, 2018.

[JvSRZ16] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 37–48, 2016.

[JWL16] Y. Jiang, D. Wu, and P. Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 12–21, 2016.

[JZ16] Konrad Jamrozik and Andreas Zeller. Droidmate: a robust and extensible test generator for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 293–294, 2016.

[JZWL16] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 122–131. IEEE, 2016.

[Kar12] Mehmet Kara. Review on common criteria as a secure software development model. *International Journal of Computer Science & Information Technology*, 4(2):83, 2012.

[KFN99] Cem Kaner, Jack Falk, and Hung Q Nguyen. *Testing computer software*. John Wiley & Sons, 1999.

[KLG+18] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2018.

[KSM+18] 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. doi:10.1109/ICST.2018.00020.

[LBL+18] Tien-Duy B Le, Lingfeng Bao, David Lo, Debin Gao, and Li Li. Towards mining comprehensive android sandboxes. In *2018 23rd International conference on engineering of complex computer systems (ICECCS)*, pages 51–60. IEEE, 2018.

[LHF15] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the kitchen sink from software. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 833–838, 2015.

[Li16] Y. Li. AndroCov. measure test coverage without source code, 2016. URL: https://github.com/ylimit/androcov.

[LL19] Li Li and Trisha TC Lin. Smartphones at work: a qualitative exploration of psychological antecedents and impacts of work-related smartphone dependency. *International Journal of Qualitative Methods*, 18:1609406918822240, 2019.

[LMOD13] 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. doi:10.1109/ISSRE.2013.6698891.

[LR19] Duling Lai and Julia Rubin. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 115–127. IEEE, 2019.

[LVBEV17] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 2–13. IEEE, 2017.

[LVBT+17] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 233–244. ACM, 2017.

[LWD+17] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. Insdal: A safe and extensible instrumentation tool on dalvik bytecode for android applications. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–506, Feb 2017. doi:10.1109/SANER.2017.7884662.

[LWY+16] 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. doi:10.1109/TASE.2016.27.

[LYGC17] 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. doi:10.1109/ICSE-C.2017.8.

[Mat13] Aditya P Mathur. *Foundations of software testing, 2/e*. Pearson Education India, 2013.

[MBMM15] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. Sig-droid: Automated system input generation for android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 461–471. IEEE, 2015.

[MEK+12] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 22–28. IEEE press, 2012.

[MHH+19] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. Paladin: Automated generation of reproducible test cases for android apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 99–104, 2019.

[MHJ16] 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. URL: http://doi.acm.org/10.1145/2931037.2931054, doi:10.1145/2931037.2931054.

[MLVBC+17] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18. IEEE, 2017.

[MMM14] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[MMP+12] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[MP15] Inês Coimbra Morgado and Ana CR Paiva. The impact tool: Testing ui patterns on mobile applications. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 876–881. IEEE, 2015.

[MSBK19] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *arXiv preprint arXiv:1904.05572*, 2019.

[MTN13] 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. URL: http://doi.acm.org/10.1145/2491411.2491450, `doi:10.1145/2491411.2491450`.

[NH19] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[OAM+18] Lucky Onwuzurike, Mario Almeida, Enrico Mariconti, Jeremy Blackburn, Gianluca Stringhini, and Emiliano De Cristofaro. A family of droids-android malware detection via behavioral modeling: Static vs dynamic analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2018.

[oSTA19] National Institute of Standards, Technology, and National Security Agency. Validation report. google llc. google pixel 3 and pixel 3 xl, 2019. URL: https://www.niap-ccevs.org/MMO/Product/st_vid10941-vr.pdf.

[Pan17] Bob Pan. dex2jar, 2017. URL: https://github.com/pxb1988/dex2jar.

[Par20] National Information Assurance Partnership. Compliant product - pixel 3/3xl, 2020. URL: https://www.niap-ccevs.org/Product/Compliant.cfm?PID=10941.

[PDGCS20] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P Carloni, and Simha Sethumadhavan. Crylogger: Detecting crypto misuses dynamically. *arXiv preprint arXiv:2007.01061*, 2020.

[PGD+18] 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. URL: http://doi.acm.org/10.1145/3243734.3278484, doi:10.1145/3243734.3278484.

[PGZ+20] 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.

[Pil20a] Aleksandr Pilgun. Don't trust me, test me: 100app. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 0–0. IEEE, 2020.

[Pil20b] Aleksandr Pilgun. pilgun/acvcut: Acvcut v1.0, October 2020. doi:10.5281/zenodo.4060422.

[PSRN18] 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*, pages 34–37. ACM, 2018.

[PZG18] Aleksandr Pilgun, Yury Zhauniarovich, and Olga Gadyatskaya. pilgun/acvtool: Acvtool v0.2, October 2018. doi:10.5281/zenodo.4060443.

[QHA+19] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. {RAZOR}: A framework for post-deployment software debloating. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1733–1750, 2019.

[Qua19] Qualcomm Technologies. Snapdragon Profiler, 2019. URL: https://developer.qualcomm.com/software/snapdragon-profiler.

[Rab16] Jan Rabe. Timebomb: Stops app usage after a period of time has passed starting from app build date, 2016. URL: https://github.com/kibotu/TimeBomb.

[RFW+19] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 603–620, 2019.

[Rub06] V. Rubtsov. Emma: Java code coverage tool, 2006. URL: http://emma.sourceforge.net/.

[Sco19] Scooter Software. Beyond Compare, 2019. URL: https://www.scootersoftware.com.

[SFE+13] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815, 2013.

[SJM17] 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. URL: http://doi.acm.org/10.1145/3106237.3106250, doi:10.1145/3106237.3106250.

[SMC+17] 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. URL: http://doi.acm.org/10.1145/3106237.3106298, doi:10.1145/3106237.3106298.

[Sof18] PassMark Software. Passmark. interpreting your results from performancetest, 2018. URL: https://www.passmark.com/support/performancetest/interpreting_test_results.htm.

[SQH17] 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. URL: http://dl.acm.org/citation.cfm?id=3155562.3155570.

[s.r17] JetBrains s.r.o. Code coverage, 2017. URL: https://www.jetbrains.com/help/idea/2017.1/code-coverage.html.

[Str14] Tim Strazzere. The new NotCompatible., 2014. https://blog.lookout.com/blog/2014/11/19/notcompatible/. URL: https://blog.lookout.com/blog/2014/11/19/notcompatible/.

[TFA+17] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017.

[THB+16] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. 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. IEEE, 2016.

[TKFC15] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[Twi20] Twitter. Help center. how to use twitter lite on android, 2020. URL: https://help.twitter.com/en/using-twitter/twitter-lite.

[VD00] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, page 214–224, USA, 2010. IBM Corp. doi:10.1145/1925805.1925818.

[VRH98] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[WDS+19] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 1–15, 2019.

[WJL+20] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. NDSS, 2020.

[WL16] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[WLL+18] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 231–242. IEEE, 2018.

[WLY+18] 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. URL: http://doi.acm.org/10.1145/3238147.3240465, `doi:10.1145/3238147.3240465`.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[WT17] R. Wiśniewski and C. Tumbleson. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps, 2017. URL: https://ibotpeaches.github.io/Apktool/.

[WYZ+15] Xiaolei Wang, Yuexiang Yang, Yingzhi Zeng, Chuan Tang, Jiangyong Shi, and Kele Xu. A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, pages 15–22, 2015.

[XBFO19] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. Identifying features of android apps from execution traces. In *2019 IEEE/ACM 6th International Conference*

on Mobile Software Engineering and Systems (MOBILESoft), pages 35–39. IEEE, 2019.

[Yan18] K. Yang. APK Instrumentation library, 2018. URL: https://github.com/kelwin/apkil.

[YH12] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012. URL: http://dx.doi.org/10.1002/stv.430, doi:10.1002/stv.430.

[YH15] 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. doi:10.1109/COMPSAC.2015.125.

[YHH19] Sen Yang, Song Huang, and Zhanwei Hui. Theoretical analysis and empirical evaluation of coverage indictors for closed source app testing. *IEEE Access*, 7:162323–162332, 2019.

[YLW09] 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. doi:10.1093/comjnl/bxm021.

[YWYZ16] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. Target directed event sequence generation for android applications, 2016. arXiv:1607.03258.

[Zel15] Andreas Zeller. Test complement exclusion: Guarantees from dynamic analysis. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 1–2. IEEE, 2015.

[ZG16] Yury Zhauniarovich and Olga Gadyatskaya. Small changes, big changes: an updated view on the android permission system. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 346–367. Springer, 2016.

[ZGC+14] 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 XXVIII*, volume 8566, pages 130–145. Springer, 2014.

[ZLZ+16] 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, pages 987–992, 2016.

[ZPG+15] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards Black Box Testing of Android Apps. In *The Tenth International Conference on Availability, Reliability and Security*, pages 501–510. IEEE, 2015.

[ZYS+19] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 128–139. IEEE, 2019.