On Supporting Android Software Developers and Testers

Carlos Eduardo Bernal Cárdenas

Bogotá, Colombia

Master of Science, Universidad Nacional de Colombia, 2015
Bachelor of Engineering, Universidad Nacional de Colombia, 2012

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
August, 2021

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Carlos Eduardo Bernal Cárdenas

Approved by the Committee, August 2021

DP

Committee Chair
Denys Poshyvanyk, Professor, Computer Science
College of William & Mary

Evgenia Smirni, Professor, Computer Science
College of William & Mary

Adwait Nadkarni, Assistant Professor, Computer Science
College of William & Mary

Oscar Chaparro, Assistant Professor, Computer Science
College of William & Mary

Andrian Marcus, Professor, Computer Science
The University of Texas at Dallas

ABSTRACT

Users entrust mobile applications (apps) to help them with different tasks in their daily lives. However, for each app that helps to finish a given task, there are a plethora of other apps in popular marketplaces that offer similar or nearly identical functionality. This makes for a competitive market where users will tend to favor the highest quality apps in most cases. Given that users can easily get frustrated by apps which repeatedly exhibit bugs, failures, and crashes, it is imperative that developers promptly fix problems both before and after the release. However, implementing and maintaining high quality apps is difficult due to unique problems and constraints associated with the mobile development process such as fragmentation, quick feature changes, and agile software development.

This dissertation presents an empirical study, as well as several approaches for developers, testers and designers to overcome some of these challenges during the software development life cycle. More specifically, first we perform an in-depth analysis of developers' needs on automated testing techniques. This included surveying 102 contributors of open source Android projects about practices for testing their apps. The major findings from this survey illustrate that developers: (i) rely on usage models for designing test app cases, (ii) prefer expressive automated generated test cases organized around use cases, (iii) prefer manual testing over automation due to reproducibility issues, and (iv) do not perceive that code coverage is an important measure of test case quality.

Based on the findings from the survey, this dissertation presents several approaches to support developers and testers of Android apps in their daily tasks. In particular, we present the first taxonomy of faults in Android apps. This taxonomy is derived from a manual analysis of 2,023 software artifacts extracted from six different sources (*e.g.,* stackoverflow and bug reports). The taxonomy is divided into 14 categories containing 262 specific types. Then, we derived 38 Android-specific mutation operators from the taxonomy. Additionally, we implemented the infrastructure called MDroid+ that automatically introduces mutations in Android apps.

Third, we present a practical automation for crowdsourced videos of mobile apps called V2S. This solution automatically translates video recordings of mobile executions into replayable user scenarios. V2S uses computer vision and adopts deep learning techniques to identify user interactions from video recordings that illustrate bugs or faulty behaviors in mobile apps.

Last but not least, we present an approach that aims at supporting the maintenance process by facilitating the way users report bugs for Android apps. It comprises the interaction between an Android and a web app that assist the reporter by automatically collecting relevant information.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to thank numerous people who helped me in preparing this dissertation. Without their hard work, patience, and guidance it would not have been possible.

First, I would like to thank my advisor and mentor Denys Poshyvanyk for his support and patience during my time at W&M.

Next, I would like to thank my current and past research group members that made my time during Ph.D. enjoyable including Mario Linares-Vásquez, Kevin Moran, Michele Tufano, Christopher Vendome, and David Nader.

Lastly, I would like to thank our Computer Science Department Chair, Professor Robert Michael Lewis, and the wonderful Computer Science administration team, Vanessa Godwin, Jacqulyn Johnson, and Dale Hayes.

I would like to dedicate this dissertation to my wife and son, Karen and Thomas who provided endless support and love throughout my time at William & Mary.

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

The ubiquity of mobile devices has brought developers' attention to build and publish applications (apps) in marketplaces such as Google Play [17] and Apple store [9]. These mobile apps must compete with many others to gain terrain and users. Thus, the importance of having high quality apps, yet with the pressure for frequent releases [97, 104] in a constantly evolving platform [123, 52]. In this dissertation we focus our work on four topics: (i) current in-field testing, (ii) Android mutation testing, (iii) bug reproduction & crowdsourced testing, and (iv) bug reporting.

1.2 Motivation

The first *goal* of this dissertation is to understand developers' and testers' needs on automated testing techniques. The second *goal* is to devise a set of techniques that are able to help developers and testers to improve quality of the apps and speed up the testing process. This work is motivated primarily by the fact that literature's landscape focuses on creating new automated techniques without focusing on real developer needs on testing practices. Moreover, there is a lack of tools that can help to improve, automate, and speed up some software engineering tasks. This leaves developers and testers ending up relying

on manual work or heuristics prone to human errors and inefficiencies.

1.2.1 Current In-field Testing

Currently, to write or record automated tests, developers must integrate a testing framework into a target testing device and the app under development. This can add undue complexity to the development process and the effort involved may cause developers to abandon such testing practices altogether [114] and perform them manually [106]. Moreover, code coverage and fault detection are common metrics used to validate the effectiveness of automated testing techniques [64]. However, there is some discussion [101, 109] on whether these can be actionable metrics in an industry environment.

We aim to help researchers and practitioners to understand developers' and testers' practices in terms of automated testing techniques used in the wild. Also, we used all the insights collected in the survey to structure this dissertation and motivate our work.

1.2.2 Android Mutation Testing

The purpose of mutation testing is to validate the robustness or quality of a test suite. This involves the creation of mutated versions (*i.e., mutants*) of the same app with some modifications applied based on rules called *mutation operators*. The purpose of the *mutation operators* is to mimic apps faulty behaviors which are defined by experts derived from the observation of real bugs. A state of the art mutation tool such as Pit [24] implements 29 *mutation operators* in its latest version. For instance, operators related to return statements (*e.g.,* empty returns, false returns, null returns) modify a random method in the code by changing the return value. Other mutation operators related to conditionals (*e.g.,* negate conditionals, and remove conditionals) change code statements that involve mostly `if` statements. Currently there is no approach that supports Android-specific operators that are representative of real Android faults. Therefore, we see the need to increase the quality of test suites by developers. Thus, improving the quality of test suites directly benefits the applications under test (AUT), users, and developers.

We propose a framework that automatically seeds Android-specific mutation operators based on a taxonomy with 14 categories derived from 2,023 software artifacts. The mutation operators are a representative result from the taxonomy resulted from a very diligent amount of manual work.

1.2.3 Bug Reproduction & Crowdsourced Testing

Mobile developers make decisions based on software artifacts such as user reviews, crash reports, bug reports, and emails, among others. Moreover, recently there is a trend for including graphical software artifacts such as screenshots and screen recordings. This is due to the number of bug reporting frameworks that have built-in screen recording features to help developers collect mobile application usage data and faults [36, 11, 10, 35]. However, all these software artifacts can overwhelm developers with so much information that will take huge amounts of time to go through all of them. We believe that the main effort of developers should be focused on core software development tasks and therefore automation for processing graphical software artifacts is required to help them with time consuming and tedious tasks.

Additionally, relying upon the crowd to perform functional and usability testing of mobile apps is becoming increasingly common as evidenced by a growing number of crowdtesting platforms [8, 34, 22]. An added benefit of crowdsourced testing, illustrated by previous work, is that integrating crowd-based app usage data and automated input generation techniques can markedly increase their effectiveness [135, 142]. Currently, to collect crowd data from users, developers typically must integrate a third-party library into their app, which can be expensive both financially and in terms of added development effort.

Conversely, we aim to allow crowd workers to simply submit videos of application usages depicting bugs or faulty behaviors that could be later automatically translated into replayable scenarios.

1.2.4 Bug Reporting

Current state of the art tools such as CrashScope [157, 153] and Sapienz [141] assist developers in the generation of inputs to find faulty behaviors while creating detailed bug reports. These bug reports are relatively robust for the apps they test. Nevertheless, they fall into the oracle problem which might be successful at reporting crashes but might miss bugs that are more subtle.

We present an approach that helps developers and testers to report faulty behaviors in apps by recording sensors, actions, and screenshots data in a very detailed crash-report.

1.3 Outline

There is a large amount of research that seeks possible solutions for software mobile development. However, there is a lack of understanding of testing practices by developers and testers. Also, there is a need for facilitating the evaluation of robustness in Android test cases that could potentially give developers advantage over other competitor apps. Moreover, developers face different problems when trying to reproduce bug reports from users such as incomplete descriptions, amount of feedback from multiple sources, lack of steps to reproduce, and lack of time to validate video bug reports among other problems. We seek to tackle these challenges in this dissertation.

In **Chapter 2** we present a survey with an in-depth analysis of developers and testers needs. More specifically, the purpose of this survey is to benefit the perspective of researchers and practitioners interested in designing approaches and tools for automated testing in mobile apps. This survey involves a total of 102 practitioners that contributed to Android open source projects. As a result we concluded that developers: (i) rely on usage models for designing test cases, (ii) prefer expressive automated generated test cases organized around use cases, (iii) prefer manual testing over automation due to reproducibility issues, and (iv) do not perceive code coverage as an important measure of test case quality. The content of this chapter is based primarily on the paper [130].

In **Chapter 3** we present a mutation testing framework to facilitate the evaluation of Android test cases. Moreover, it includes a taxonomy of 14 categories of faults on Android apps derived from a total of 2,023 artifacts. These artifacts were collected from sources for Android such as: (i) bug reports from open source apps, (ii) bug-fixing commits of open source apps, (iii) Android-related Stack Overflow discussions, (iv) exception hierarchy of Android APIs, (v) bugs and crashes described in previous studies, and (vi) reviews posted in the Google Play store. As a result of the taxonomy, we proposed 38 Android-specific mutation operators implemented in an infrastructure called MDroid+. Additionally, we executed a comparative analysis with previously proposed tools that are Java- and Android-specific, leaving MDroid+ with more representative of Android faults. MDroid+ has a better performance in terms of non-compilable and trivial mutants. The work described in this chapter is based primarily on work on the MDroid+ framework [128].

In **Chapter 4** we propose a new approach to automatically translate video bug reports into replayable scenarios called V2S. This approach relies on deep learning (DL) techniques to facilitate the detection and segmentation of users' actions in video bug reports by analyzing every single frame of the video. V2S is divided in three main phases: (i) the first phase is *touch detection* that uses Faster R-CNN and AlexNet to detect and estimate opacity of touch indicators respectively; (ii) the second phase is *action classification* that takes the frames and opacity levels to segment and assign a type of action to the detected events; (iii) the third phase is *scenario generation* that takes all the actions and locations to accurately generate a low-level script that can be replayed in an Android device. As a result of this research, V2S is capable of replaying actions with high accuracy (*i.e.,* $\approx 89\%$) and illustrates the potential usefulness in an industry setup. The work described in this chapter is based primarily on work on the V2S approach [55].

In **Chapter 5** we present an approach called ODBR to assist developers and testers in reporting bugs. ODBR records per user interaction low level events as well as screen-shots with the current state of the app to facilitate precise bug reporting. ODBR has four major components which includes: *Getevent manager* to detect when new inputs are

received, *user-interface-hierarchy manager* that captures the hierarchy and information from graphical-user-interface-components, *Event manager* that samples a stream of sensors (*e.g.,* accelerometer, and GPS), and *Screenshot manager* that takes a screenshot of the current app state. Then, ODBR Android app sends this information to a web application that generates a fully expressive bug report in `html`. The work described in this chapter is based primarily on work on the ODBR approach [150]

In addition to the contributions outlined in this dissertation, the author has worked on a wide variety of research areas in software engineering over the course of his graduate studies including publications on: (i) Android program analysis [132, 52], (ii) improving energy consumption of Android apps [124, 125, 129, 126], (iii) improving Android bug reporting [193, 155, 157, 153, 59, 152], (iv) automated testing of Android apps [135], (v) improving GUI design and implementation of Android apps [151, 56], and (vi) security and traceability in software [168, 154].

Chapter 2

How do Developers Test Android Applications?

2.1 Introduction

Mobile devices have quickly become the most accessible and popular computing devices in the world [13] due to their affordability and intuitive, touch-based user interfaces. The ubiquity of smartphones and tablets has led to sustained developer interest in creating "apps" and releasing them on increasingly competitive marketplaces such as Apple's App Store [48] or Google Play [17]. Due to their highly gesture-driven nature, GUI-based testing of mobile apps is paramount to ensuring proper functionality, performance, and an intelligent user experience. However, GUI-based testing activities are typically costly, and in the context of mobile apps, often performed manually [105, 114]. Given the additional constraints on the mobile application development process including pressure for frequent releases [97, 104], rapid platform evolution and API instability [52, 123], and parallel development across different platforms [105, 40] it can be difficult for developers to budget time for effective testing. Thus, the challenge of automating mobile app testing has captured the interest of the software engineering and systems research communities, and has led to the development different types of automated techniques that assist in various testing

tasks.

Research-oriented tools aimed at improving mobile testing span a diverse range, from record & replay approaches [82, 99], to bug reporting aids [155, 152], to automated input generation techniques [6, 138, 181, 175, 43, 44, 50, 157, 15, 42, 50, 63, 197, 195, 135, 200, 102, 146, 140, 141]. Perhaps the most interesting and valuable of these techniques from a developer's or tester's perspective are the automated input generation (AIG) techniques. The high-level goal of such techniques is relatively simple: given a mobile application under test (AUT), generate a series of program *inputs* according to a pre-defined *testing goal*. For the vast majority of these techniques, the generated *inputs* are simulated touch events on the screen of a device, and the *testing goal* is typically either achieving the highest possible code-coverage or uncovering the highest number of faults (e.g., crashes).

However, despite the large amount of research effort dedicated to building AIG techniques and other automated approaches, recent studies seem to indicate that these approaches are typically not used in practice [114]. Choudhary *et al.* offer a set of potential reasons for this lack of adoption as part of an experience report analyzing several AIG tools [64], and among the reasons reported are (i) lack of reproducible test cases, (ii) side effects across different testing runs, and (iii) lack of debugging support. While this study offers some insight, researchers and practitioners who aim to build these tools with the intention of them gaining adoption and positively impacting developers do not have a clear understanding of the testing needs and preferences of real developers.

This lack of guiding direction for this particular topic of mobile software engineering research is somewhat troubling given the highly practical impact that such tools could have on daily development and testing workflows. Conversely, it is unsurprising that many tools have failed to make an impact without taking into account developer preferences, as *"Automation applied to an inefficient operation will magnify the inefficiency"*[1]. If the *operation* or *goals* of automation techniques do not match developer needs, preferences, and expectations (and are thus *inefficient*), there is little chance that these will have a

[1]Bill Gates, co-founder of Microsoft, in reference to automation in business settings

meaningful impact. Therefore, there is a very clear demand to align the goals of research on automated testing techniques with the needs of developers in order to allow for practical impact.

In this project, we aim to bridge this gap through a survey, that at its core, aims to examine the testing preferences of open source developers with the intended purpose of providing actionable information to researchers and practitioners working on approaches to automate different aspects of mobile testing.

In summary, this project makes the following noteworthy contributions:

- To the best of our knowledge, this is the first work aimed at analyzing mobile testing preferences of real open source developers with a focus on (i) typical preferences when designing test cases for mobile apps, (ii) preferred characteristics for automatically generated test cases, and (iii) preferred effectiveness metrics.

- This study complements previous work that has identified and speculated upon potential reasons for lack of adoption of automated mobile testing approaches by collecting information from open source developers and providing a set of learned lessons to guide future research.

- Our general findings indicate that developers (i) rely heavily on usage models of their applications when designing test cases, (ii) prefer high-level expressive automatically generated test cases organized around use-cases, and (iii) prefer manual testing over automation due factors including test case representation and issues with reproducibility (iv) do not hold the perception that code coverage is an important measure of test case quality, as indicated by a large portion ($\approx 64\%$) of study participants, instead citing other measures of quality such as feature coverage or fault-detection as more useful.

2.2 Design of Empirical Study

The main *goal* of this study is to identify and analyze practices and preferences of mobile developers (MDs) toward testing related activities. To this end, we explored MDs (i) practices in documenting requirements and designing test cases, (ii) preferences toward features of automated testing approaches, (iii) use of existing automated tools, and (iv) preferences for testing-realted quality measures. The study is intended to benefit the *perspective* of researchers and practitioners interested in designing approaches and tools for automated testing of mobile apps.

While common wisdom and best practices suggest that test cases should be derived from requirements artifacts, to the best of our knowledge, previous studies focused on the challenges and tools used for testing but without analyzing the details of the strategies used by MDs for designing manual test cases or exploring preferences for automatically generated test cases [105, 114][2]. These aspects are important, as learning the preferences of developers' manual testing practices can inform automated techniques to best meet these needs.

Consequently, we aim to fill this "gap" in recent work by surveying contributors of open source Android apps hosted on GitHub. We are most interested in understanding testing practices of mobile developers from the viewpoint of test case design, preferred testing strategies, reasons for the prevalence of manual testing (as suggested by previous work [105, 114]), and preferred information/features of ideal automatically generated test cases. We also wanted to survey MDs about the usage of widely used techniques in the research community such as random testing and coverage analysis.

[2]Kochhar *et al* [114] investigated also with a survey with 83 open source Android developers the tools they use and challenges they face while testing Android apps. It is worth noting that our survey also includes a question concerning the tools used for automated testing (See **SQ8** in Table 2.1). Kochhar *et al* [114] list a reduced set of tools (i.e., 10), however, we complement their findings with a list of 55 tools used by our surveyed participants. We also complement their findings with a specific question designed to understand experiences and issues of mobile developers when using random testing tools (See **SQ9** in Table 2.1)

Table 2.1: Survey Questions for our Empirical Study

Id	Question (Type)
SQ_1	What type of documentation do you use for specifying the requirements in your apps? (Multiple choice)
SQ_2	How do you usually distribute your testing time among these different activities (e.g., manual testing 20%, Junit testing 50%, cloud testing 30%)? (Open question)
SQ_3	Please provide rationale for your answers to SQ3. (Open question)
SQ_4	How do you design the test cases for your apps? (Open question)
SQ_5	What is the target of the test cases you design for your apps? (Multiple choice)
SQ_6	If you are using (or intend to use) tools for automatic generation of test cases, what format for the test cases do you prefer? (Single choice)
SQ_7	Assuming you have test cases in natural language, what type of information would you like to have in them? (Multiple choice)
SQ_8	What tools do you use for automated testing? (Open question)
SQ_9	What are your experiences with random testing tools such as Android Monkey? Are random testing tools useful for your needs? Did you experience any issues with the sequences of events generated by a random testing tool like Android Monkey? (Open question)
SQ_{10}	Do you use code coverage as a metric for measuring the quality of your test cases? Why? (If you answer is No, please describe how else you measure or ensure the quality of your test cases) (Open question)

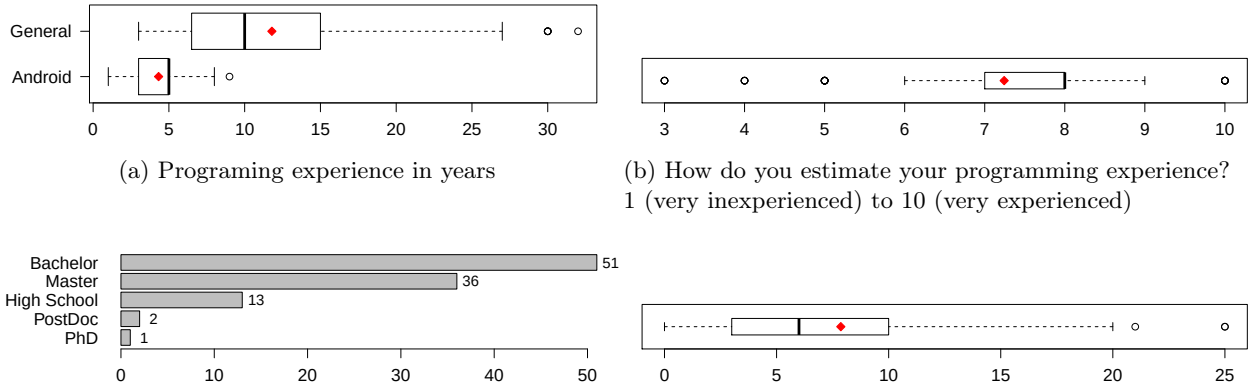
2.2.0.1 Research Questions

In particular, we aimed at answering the following research questions (RQs):

- ***RQ***$_1$*: What are the strategies used by MDs to design test cases?*
- ***RQ***$_2$*: What are the MDs' preferences for automatically generated test cases?*
- ***RQ***$_3$*: What tools are used by MDs for automated testing?*
- ***RQ***$_4$*: Do MDs consider code coverage as a useful metric for evaluating test cases effectiveness?*

2.2.0.2 Data Collection

Table 2.1 lists the questions in the survey. SQ_1-SQ_5 were used to answer $\mathbf{RQ}_1$; SQ_6 and SQ_7 aim at answering $\mathbf{RQ}_2$; SQ_8 and SQ_9 were designed to answer $\mathbf{RQ}_3$; and SQ_{10}



(a) Programing experience in years

(b) How do you estimate your programming experience? 1 (very inexperienced) to 10 (very experienced)

(c) What is your academic level

(d) How many years of industrial experience do you have (programmer, software engineer, etc.)?

Figure 2.1: Results of demographic questions for the 102 survey participants

served to answer $\mathbf{RQ}_4$. We also collected demographic background information to filter participants with short or over claimed experience in Android development, and to measure the diversity of our sample.

The survey was hosted online on the Qualtrics platform [26], and the participants were contacted via email. To select the potential participants, we followed the same procedure from previous work [134] — that also surveyed open source developers — to extract contributors' emails from GitHub. After the extraction and filtering, we emailed the survey to 10,000 email addresses from which we got 485 survey responses. We discarded 5 responses in which the participant reported 0 years of experience in Android programming, 3 with invalid answers, and 370 unfinished surveys. In the end, we obtained 102 valid responses.

The demographics of the participants are depicted in Figure 2.1. Our sample (102) is comparable to final numbers of mobile developers (with industrial experience) surveyed/interviewed in previous studies investigating other software engineering phenomenon: 3 in [125], 9 in [145], 45 in [52], 83 in [114], 200 (188 developers + 12 experts) in [105], and 485 in [134]. In addition, the claimed programming experience is diverse for the three cases: general programming, Android programming, and industrial experience.

The answers to multiple/single-choice questions were analyzed using descriptive statis-

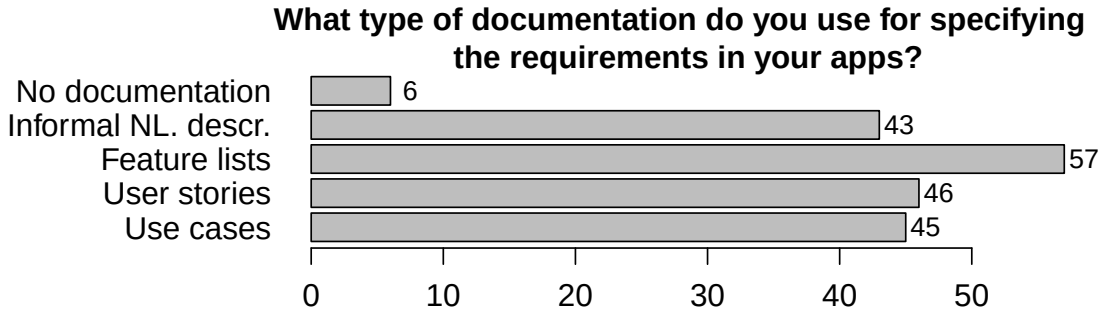


Figure 2.2: Artifacts used by Android developers to document apps requirements. The bar plots show the number of times each artifact was selected by the participants.

tics. In the case of open questions, we categorized the answers manually following a grounded theory-based approach [69]. Three of the authors went through all of the free-text answers and performed one round of open coding by independently creating categories for the answers. After the round of open-coding, the codes were standardized. In the cases of non-agreement between the three coders, corresponding answers were marked as "Unclear".

2.3 Results and Discussion

In this section we report the responses by the participants and provide answers to the aforementioned research questions. We describe the results using descriptive statistics and through summaries and discussion of examples of free-text answers.

2.3.1 RQ_1: What are the strategies used by MDs to design test cases?

2.3.1.1 Artifacts for documenting app requirements

Android developers use a diverse set of artifacts to document requirements, including artifacts from disciplined and agile methods. Fig. 2.2 depicts the answers for each of the options in our **SQ1** (note that this was a multiple-choice question). Although there is no large tendency towards a preferred artifact, feature lists are the top-used artifact. Surprisingly, only six participants do not document requirements, and the usage of the

other options (i.e., use cases, user stories, and informal natural language descriptions) is balanced across the participants. When analyzing the most popular answers (including combination of choices as a whole answer provided by participants), there is no clear preference; however, we found that the most popular responses reporting the usage of only a single type of artifact as the documentation practice are distributed as follows: user stories (14 participants), feature lists (12), informal natural language descriptions (10), use cases (9). Additionally some participants selected the combination of all 4 artifacts (8) as a single answer.

2.3.1.2 Distribution of testing efforts

Previous studies have reported that manual testing is preferred over automated appraoches [105, 114]. In the survey, we asked participants about how they distribute their testing effort and time across different testing activities (**SQ2**). In particular we asked about the following activities: manual testing, random testing using Monkey, JUnit testing, Record & Replay-based (R&R) testing, GUI ripping-based tools, automated testing with automated testing APIs (ATA), cloud testing services, and others. The answers provided by the participants are depicted with boxplots in Fig. 2.3.

As expected, manual testing is the preferred testing activity with an average of 58.18% of the testing efforts dedicated by the participants, an interquartile range (IQR) of $[25\%, 97.25\%]$ and a maximum of 100%; 96 out of 102 participants reported more than 5% of testing effort devoted to manual testing with 35 of them reporting more than 90% of dedication to manual testing. The second most popular activity in which the developers dedicate their testing efforts is JUnit-based testing with and average dedication of 18.96%, an IQR of $[0\%, 30\%]$ and a maximum of 90%; 58 participants reported more than 5% of testing effort devoted to unit testing. The third most popular testing activity is automated testing with APIs such as Expresso and Robotium, with an average dedication of 8.29%, an IQR of $[0\%, 10\%]$ and a maximum of 80%; 34 participants reported more than 5% of effort devoted to manual testing.

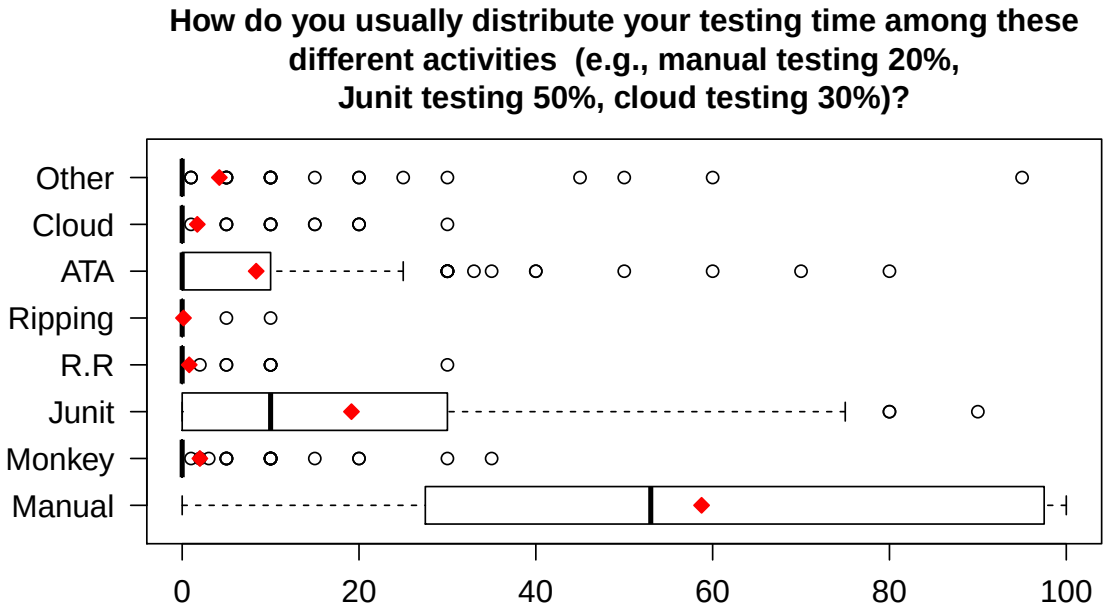


Figure 2.3: Distribution of testing activities reported by our participants. *ATA means Automated Testing APIs, and R.R means Record & Replay.

Fuzz/Random testing with Android Monkey is widely used as the baseline for comparing new automated testing tools proposed by the research community [138, 135, 64, 157]. However, responses from our participants suggest that fuzz/random testing is not widely used in-the-wild. Only 11 participants reported more than 10% of testing effort with Android Monkey with an average of 16.36% (for those 11 participants) and a maximum of 35%. One automated strategy that is widely used in the research community and serves as the foundation for multiple methods of AIG is GUI ripping [43, 50, 42, 159]; however, it seems that developers in-the-wild are not aware of such tools or do not find them useful. Only two participants reported the usage of automated ripping tools with 5% and 10% of their testing efforts. Concerning the case of Record & Replay, five participants reported this technique with more than 5% of testing effort and a maximum of 30%.

Despite of the availability of services such as Xamarin test cloud [194], Saucelabs [31], and Perfecto [23], only 14 participants mentioned the usage of cloud services for testing with an average effort of 12.57% (for those 14 participants) and a maximum of 30%.

Finally, under the "Others" option, we got 14 answers in which the participants claimed more than 10% of testing effort with an average of 29.29% and maximum of 95%. The participants further explained in the answer to **SQ3** that this is mostly because they use customized tools or strategies, instrumentation-based testing, or beta testing with users.

Regarding the rationale provided by the participants for their choices, the preference for manual testing is supported by several reasons such as (i) changing requirements, (ii) lack of time for testing and process decisions, (iii) size of the apps, (iv) lack of knowledge of automated tools and techniques, (v) usability and learning curve of available tools for automated testing, and (vi) the cost of maintaining automated testing artifacts. For instance, the following rationale provided by some participants illustrate their preference for manual testing as a consequence of changing requirements:

" *our app changes very frequently and we can't afford unit testing and automated testing* "

" *it's hard to right the useful test case for current project, because the requirement is changed very often, and the schedule is very tiny. So we still prefer hire some tester to do manually testing. In the other hand, the android test framework is not good enough yet, I tried study roboeletric, it's a little bit hard to understand.* "

" *A lot of what I do is related to how the app looks and feels. Therefore a lot of my testing is done manually. When I have complexity in my classes I use junit. Sometimes I use the Instrumentation testing classes from Android, but not much as manually testing feels faster. Also the requirements tend to change a lot so manually testing unfortunately is the best option for me.* "

The survey participants also justified the usage of manual testing because of time-related issues and project management decisions. For example:

" *Usually customers doesn't provide enough time for development of automated tests.* "

" *Too much time required in configuring the components for automated testing. Partly because I was working in a consultancy firm so there was no incentive to spend time on automated testing (not chargable).* "

" *I do not agree about this method but this is management decision. I repeatedly expressed by disconformity with this methodology.* "

" *Although I strongly disagree with this: the institution I work at does not provide the atmosphere to make testing a vital part of our development.* "

" *My previous work didn't have any testing requirements for the apps, and writing tests takes time that the budget didn't account for.* "

Cost, in terms of money and time, for creating and maintaining automated testing artifacts is also another reason for preferring manual testing. This case is illustrated by the following examples of rationale provided by the survey participants:

" *I'm faster by testing the app and all it's possibilities on the device itself, instead of writing separate Test Cases.* "

" *Quickly testing the product is important and writing automated tests can't be done quick enough. So a majority of my time is spent making sure the product works manually, then spending time automating what I can. jUnit is wrote by developers, so I've added a couple of tests, but not much. Have just played around with Android Monkey. Our product uses the Cloud, but another person takes care of the majority of the testing.* "

" *We prioritize feature work over automated testing. Automated tests have done little to prevent bugs, but incur significant overhead when creating new features or refactoring old.* "

" *For Android traditionally it has been very difficult to write tests for. Also UI tests can be brittle and take significant effort to maintain. In an environment where development resources are constrained and features / bug fixes works takes priority, it is very difficult to have sufficient tests. Manual testing with a dedicated QA team is more practical to maintain.* "

" *I'm skeptical of UI testing because I've found that the tests are fragile, require a lot of maintenance, and are generally more work than worth.* "

The survey participants also claimed a general lack of knowledge of existing automated testing techniques, along with difficulties related to the usability of the tools as factors for

not performing automated testing:

" *We don't really know how to use other tools.* "

" *I did not know about Android Monkey prior to this survey. I will try it out.* "

" *I have not found any easy-to-use testing solutions for Android* "

Finally, the size and maturity of the apps is also a factor that influences the preference for manual testing:

" *I never got involved in an Android project big enough to require unit testing, it wasn't worth investing in that.* "

" *I mostly do manual testing due to the limited size of my apps. I sometimes use a custom replay system (built into the app) to duplicate bugs after I come across them. This method is usually combined with manual testing (printing debug information to the log) to pinpoint the cause.* "

" *I'm mostly just building toys or research prototypes, never built Android apps professionally. So I test pretty informally (and poorly) because I just want to build a thing quick and don't care if it's robust.* "

2.3.1.3 Test case design strategies

After the open coding for the responses to **SQ4**, 34 answers were not considered because (i) the participants explicitly mentioned they do not perform testing or do not design test cases, and (ii) for some answers we were not able to understand/codify the textual answer. From the valid/accepted answers, the top strategy reported by the participants to design test cases is follow the usage model of the app as a guide (30 answers). The next in the list is designing unit tests for individual components/methods (10 answers), followed by negative testing and edge cases (9 answers), and testing expected outputs (6 answers). Bugs, changes in the last version, and regression account for 9 answers. Three participants claimed they follow the Behavior-Driven-Development philosophy (BDD). In addition, three participants mentioned they perform ad-hoc testing. Finally, two of the participants combine the usage model with feedback from the end users.

Non-functional requirements were mentioned only in few cases as the drivers for designing test cases: robustness (2 answers), performance (1 answer), usability (1 answer), and different device configurations (1 answer). Other strategies mentioned by only one participant each are: code coverage, defining assertions in code, dependencies, testing the GUI model, and testing the business logic.

We also found that developers tend to prefer a single criteria for designing test cases, as very few respondents reported more than one preference. Only 15 participants reported mixed strategies, as those described in the following responses:

" *l look for boundary conditions - i try to work out what happens in the Grey Areas - i look for ways of breaking it. i also test a range of use cases, how Can the user interact with the app? when combinations are possible? i try to test the most probable Scenarios and some strange ones.* "

" *1) specific tests depending on what the app should do. For instance: schedule should be as precise as is required for a scheduling app. 2) robustness: find the limits on the app by constantly changing the aspect ratio of the screen or switching app on and off. 3) look at CPU utilization while testing the app... app should not drain battery unnecessary* "

" *Based on the user stories or Use cases. Define initial state (local data). Perform scenario (call a rest service / perform an activity, etc). Validate final screen or rest service result.* "

The answers to **SQ4** were complemented by participants reporting the testing goals they have when designing test cases (**SQ5**). Fig. 2.4 depicts the responses for **SQ5**. As shown in Fig. 2.4, the developers prefer to design test cases that target individual uses cases/features (77 answers), or combinations of multiple uses cases/features (49 answers). Random events were mentioned only by 16 participants. The option "Other" was selected by nine participants; four of the "Other" answers were "none", two participants mentioned non-functional attributes (i.e., performance and robustness), one participant responded "corner cases", and one mentioned "unit test".

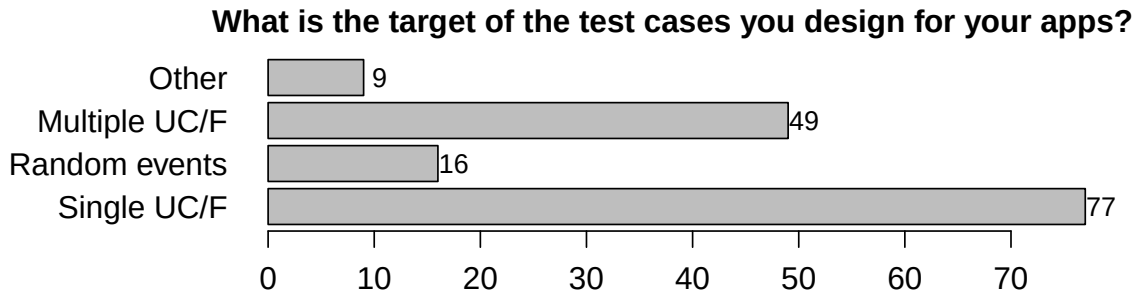


Figure 2.4: Target of test cases designed by the survey participants. The bar plots show the number of times each target was selected by the participants. *UC means Use Case, and F means Feature.

> **Answer to RQ_1.** Mobile developers (as represented by our survey sample) heavily rely on usage models to document and design test cases. First, requirements are documented using different artifacts such as feature lists, informal natural language descriptions, user stories and use cases; only six out of 102 participants reported they do not use any artifact to document requirements. Second, the surveyed participants mostly rely on manual testing and unit testing for their testing strategies. The rationale provided by the participants for their preference and effort dedication to manual testing is supported by several reasons such as (i) changing requirements, (ii) lack of time for testing and process decisions, (iii) size of the apps, (iv) lack of knowledge of the tools and techniques, (v) usability and learning curve of available tools, and (vi) the cost of maintaining automated testing artifacts. Finally, the surveyed developers mostly focus on the usage model to design test cases, and use one or a combination of use cases/features as the target for their test cases.

2.3.2 RQ_2: What are MDs preferences for automatically generated test cases?

Concerning **SQ6**, natural language is the format preferred by the survey participants (See Figure 2.5) for automated test cases, with 33 out of 102 participants selecting this

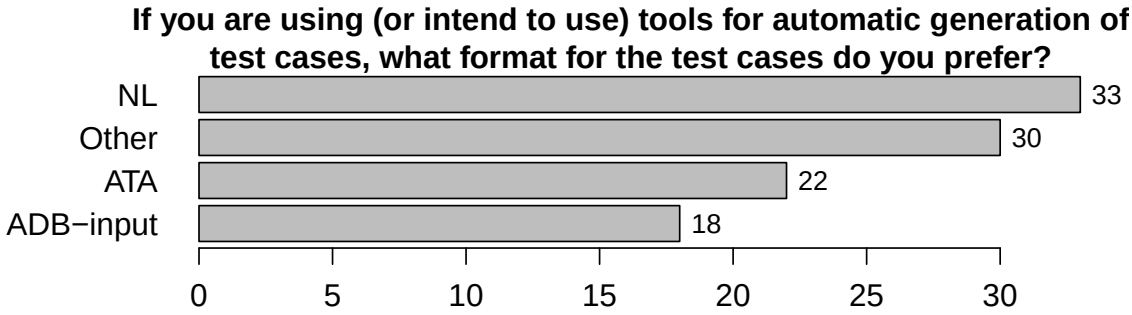


Figure 2.5: Automatically generated test cases format preferred by the survey participants. NL means Natural Language, ATA means Automated Test API, and ADB-input means input commands generated via the Android Debug Bridge (ADB).

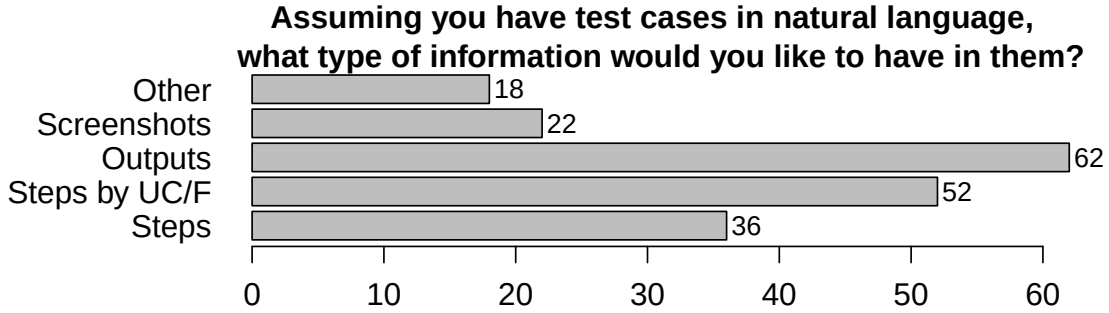


Figure 2.6: Information preferred by survey participants in automatically generated test cases. UC means Use Case and F means Feature.

option. The second most popular answer was the "Other" option (30 participants), which represented mostly the lack of participant knowledge about tools for automatic generation of test cases or the lack of preference for any format. Unfortunately, the option "Other" did not provide us with actionable knowledge; 17 textual answers for the "Other" option are empty or claim no preference for any format; ten participants reported that they do not use/do not like/are not aware of tools for automatic generation of test cases; one answer was unclear; one answer mentioned scripts augmented with comments; and one participant responded "Replayable event streams, starting from a known good state".

The third most popular option for **SQ6** was test cases written with automated testing APIs such as Espresso and Robotium (22 participants), and the least selected choice was

ADB input commands (18 participants). Therefore by combining the number of participants voting for natural language test cases and test cases written with ATAs, the results suggest that Android developers prefer expressive test cases over low level scripts with input commands.

Concerning the preferred information that developers would like to have in an ideal automatically generated test case (**SQ7**), expected outputs and reproduction steps organized by use case or feature are preferred over the other options. The answers for **SQ7** are depicted in Fig. 2.6. 82 out of 102 participants (80.39%) selected either "Expected outputs" or "Reproduction steps grouped by use case/feature". Only 21.7% of the participants agreed on having screenshots as part of test cases. And, the textual answers to the "Other" option include: reason/motivation for the test case, device and contextual information (e.g., Android OS version, display dimension, internet connection status), malicious user inputs, and specifications like in the RSpec framework for Ruby [30].

> **Answer to RQ_2.** Automatically generated test cases in natural language or expressed using automated testing APIs (e.g., Robotium or Espresso) are preferred by the participants. This suggest a preference for high-level languages instead of low level events (e.g., using ADB input commands). In addition, the surveyed developers prefer to have test cases that include expected outputs and reproduction steps organized/grouped by use cases/features.

2.3.3 RQ_3: What tools are used by MDs for automated testing?

2.3.3.1 Tools for automated testing

55 different tools have been used by our survey participants (**SQ8**); the tools and frequencies are depicted as a word cloud in Fig. 2.7. The most used tool is JUnit [20] (45 participants), followed by Roboelectic [27] with 16 answers, and Robotium [28] with 11 answers. 28 participants explicitly mentioned they have not used any automated tool for

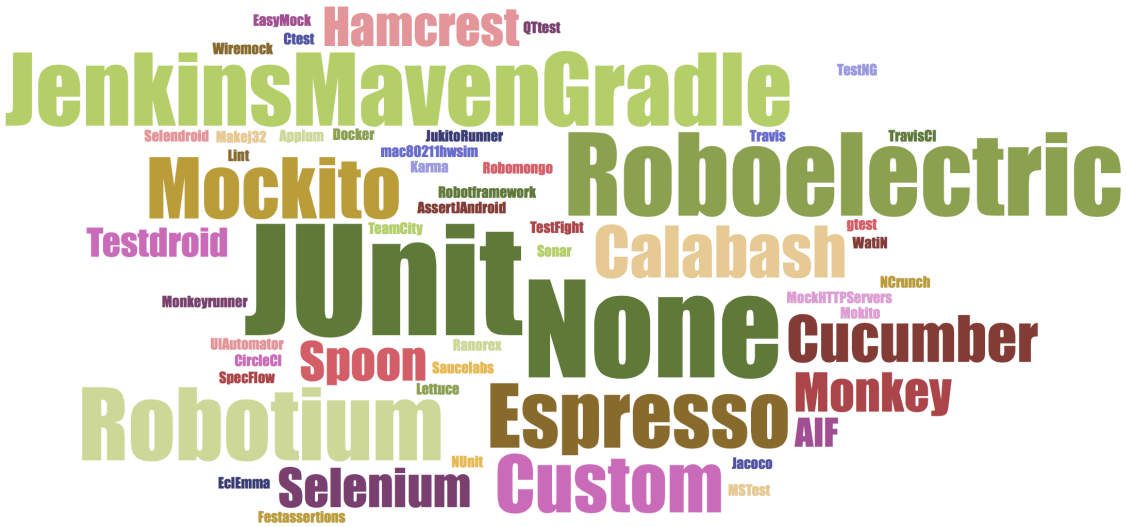


Figure 2.7: Tools used by the participants for automated testing of Android apps.

testing mobile apps. 39 out of 55 tools were mentioned only by one participant each, which suggests that mobile developers do not use a well established set of tools for automated testing. In addition, surprisingly, Monkey [86], the state-of-the-art tool for fuzz/random testing, was mentioned only by three out of 102 participants, and the results are similar for other tools designed/promoted by Google: Espresso [85] (eight participants), MonkeyRunner [84](one participant), Lint [83] (one participant), and UIAutomator [7] (one participant). None of the mentioned tools allow for automatic derivation of test cases from source code, real app usages, or requirements specifications[3]

2.3.3.2 Experiences with random testing tools

60.78% (62 out of 102) of the participants reported no experience with Monkey or tools for random testing (**SQ9**). 15 participants (14.71%) provided no-rationale/unclear answers. Concerning usefulness, 13 participants (12.75%) consider Monkey and random testing as useful tools for corner cases or stress testing, and in some cases for finding performance issues. For instance:

[3]It is worth nothing that the Android Monkey tool generates random sequences of events, however, the sequences are not easy to document or describe in terms of use cases.

“ *Yes, it's useful to detect minor/stability issues. For instance it sometimes finds issues happened when tapping buttons many times quickly.* ”

“ *Monkey is very useful for stress testing the application or to verify that there are no leaks (typically memory leaks) that build up over time. Sometimes, it also catches the odd bug as well.* ”

However, eight of those 15 participants pointed out that some issues or limitations are related to low impact of the discovered bugs and reproducibility:

“ *They can be useful in finding memory leaks, ANRs, bad navigation flows and the like. They can be problematic in doing unexpected things, e.g. exiting your application during a test run.* ”

“ *Good for stress testing, not very consistent results.* ”

“ *They add a lot of noise for very little signal. They are good at finding some really weird cases, but mostly it seems like the cases they find wouldn't normally be hit by a user and that time could be better spent adding features.* ”

“ *I used it long time ago and find that it works as a solution only for low-quality fragile code. It's rarely helps to improve overall quality of the app.* ”

11 participants (10.78%) provided answers in which the main message is that Monkey/Random testing tools are not useful because of reproducibility issues, maintenance costs of the scripts, and lack of tangible benefits: “ *hard to reproduce bugs, steps hardly reproducible by human beings* ” “ *I've ran Android Monkey and it found some defects, but the developers said "that barely happens" or "that never happens". So the defects weren't looked into.* ” “ *I don't think random tests provide any value. The ideal is to have a tool able to perform an entire session (run several scenarios) / without having the test-runner to kill and initialize the app before running each scenario.* ”

Although random testing has been proven to be “useful" by the research community [76, 49, 65, 41], available random testing tools for mobile apps (e.g., Monkey) have issues such as lack of expressiveness. For instance, the Android Monkey tool allows for reproducibility of event streams (by using the same seed), but it does not have log capabilities for creating

a higher level representation of the streams. In addition, it seems to be the only tool available for the community for random testing of mobile apps.

Researchers have also designed tools for mobile random testing (e.g., Dynodroid [138]), however those tools are still only known by the research community or have impediments for industrial usage (e.g., applicability only under certain conditions). In general, the main finding here is the lack of experience with and lack of usage of random testing tools by the surveyed participants. Regarding the usefulness, few participants reported cases in which the tools are useful for finding corner cases and stress testing; and few participants reported that the tools are not useful at all. However, common complaints in both cases (from both participants who saw benefits and those who didn't) are lack of reproducibility of the event sequences. These results suggest, that current tools used by Android developers need to be improved to allow expressiveness of the generated streams. Also more effort in the research community should be dedicated to promote the usage of random testing tools generated as part of research, and to deliver tools that can be easily adopted by the industry.

Answer to RQ_3. The surveyed participants rely on a diverse set of tools for supporting automated testing of mobile apps. In particular, 55 different tools were reported showing a preference for APIs such as JUnit, Roboelectric and Robotium. Compared to the study by Kochhar *et al.* [114], we report a larger set of tools answered by a larger set of participants. However, both studies agree on listing JUnit, Roboelectric and Robotium as part of the top-4 used tools. Record & replay, and random testing tools are used only by few participants. In the case of random testing tools, few participants claimed some benefits such as stress testing, execution/discovery of corner cases, and execution of events that are hard to generate by humans. However, impediments for increased adoption of random testing tools (e.g., the Android Monkey tool) are the lack of expressiveness of the generated event streams, and difficulty reproducing scenarios.

2.3.4 RQ_4: Do MDs consider code coverage as a useful metric for evaluating test cases quality?

14 out of 102 participants reported they do not use code coverage, do not use automation tools, or were unaware of code-coverage as a quality metric (**SQ10**); and six out of 102 participants provided no valid answers. From the remaining 82 participants, 51 answered "No" (i.e., code coverage is not useful), 29 answered "Yes" (i.e., code coverage is useful), and two participants provided a "yes-no" answer. In the case of the "No" answers, 19 out of 51 augmented the answer claiming that code coverage is not a good metric for measuring quality of test cases because there are other useful and better methods/metrics such as code (test cases) reviews, number of faults detected by the test cases (fault-detection capability), features covered by the test cases (feature coverage), or the "works for me" criteria[4]. Examples of the answers claiming that code coverage is not a useful metric for evaluating test case quality include the following:

" *No. We measure the number of uncaught bugs and regressions over time that devs had to spend time fixing* "

" *No, calculate total coverage based on features, covered elements etc.* "

" *I don't usually participate in the testing side of things, but I wouldn't use code coverage as a metric for quality as the two are completely distinct and different things.* "

" *Code coverage categorically does not measure the quality of tests. It is useful to show that code is not currently tested but it says nothing further about the code that is already under test. Many people – probably most – are quite skilled at writing useless tests. Education is the only tool for producing high-quality tests and code review is the only tool for ensuring that quality. That said, fuzz testing, for instance, is a very powerful tool for certain kinds of testing. Knowing how to use the tools available to us is part of that education.* "

29 out of 102 participants found code coverage a useful metric for measuring test cases

[4]Note that the effectiveness of code coverage for measuring the quality of test suites has been already questioned by the software engineering community [101, 109, 202].

quality, in particular for identifying code entities that have not been tested. Examples of their answers are as in the following:

" *I use code coverage mainly as a tool to ensure I haven't forgotten any major areas of testing. Most of my projects have a minimum coverage requirement of 75-80%* "

" *Yes, I try to keep code coverage at an acceptable level. This is definitely not the only thing that matters, but I think it does matter.* "

" *We use code coverage because it's easy to measure, it's a good enough metric, and because if developers feel they are being measured they are more likely to write more tests, thus generally producing the desired outcome.* "

Finally, the "Yes-no" answers claim code coverage is not useful at all, but they help to identify parts of the code that have not been tested:

" *Yes and no. It's an indication if something is tested, not that the test is correct.* "

" *No. We use code coverage more as a guide to which part of the code base might need more attention in terms of writing more tests. We don't really have other metric for measuring quality of the test cases.* "

> **Answer to RQ_4.** Code coverage is not used or not considered as useful for measuring the quality of test cases by 63.73% (i.e., 65 out of 102) of the surveyed participants. Some of the reasons explaining the lack of confidence in the metric is that they prefer fault-detection or feature-coverage capabilities of test cases as a measure of quality, or they prefer to measure test cases quality by performing code reviews. On the other side, some participants consider that code measure is a useful metric because it helps to identify parts of the code that are not tested.

2.4 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation, and relate to possible measurement imprecision when extracting data used in a study. To

minimize this threat we filtered out incomplete surveys, participants with zero years of experience, and surveys with invalid answers. Moreover, to minimize a source of inexactitude in our study on the open questions, we followed a grounded theory-based approach [69]. In particular three of the authors performed an open coding by independently creating categories, then the codes were standardized and in case of no-agreement the answers were marked as “Unclear".

Threats to *external validity* concern the generalizability of our findings. The results in our study may not be generalizable to developers on other platforms, moreover our study only focuses on developers from open source projects on Github and we can not guarantee that all participants are commercial developers (although the average industrial experience for participants is between 5-10 years). In addition, the testing practices and automated tools used by our sample set may not generalize across all mobile developers. However, despite this fact, we believe the information provided by our respondent pool can be used to effectively provide guidelines for the research community towards devising more practical automated testing approaches.

Another threat related to generalizability is that the results of our study are based on 102 respondents which might not be representative of the global community of Android developers. However, this study surveys a comparable number of Android developers to other studies [105, 114, 39].

2.5 Related Work

In this section we present the related work and we differentiate the outcomes of our study on how developers test android applications compared to other studies concerned with investigating the topic of mobile testing.

2.5.1 Surveys on Android Testing

Erfani *et al.* [105] performed a study to understand the challenges that developers face during the life cycle of mobile software development. The study comprises interviews and a semi-structured survey targeted to 12 experts on mobile development and 188 people from the general mobile community respectively. Therefore, the findings can be categorized in four main topics: (i) general challenges such as fragmentation, testing support, open/closed platforms, data intensive, and frequent code changes; (ii) development across multiple platforms with problems such as native vs. hybrid apps, capabilities of platforms, code reuse vs. writing from scratch, behavioral consistency cross platform, and effort on migration across platform; (iii) current testing practices like manual testing, developers as testers, platform specific testing, levels of testing, beta testers; and finally (iv) testing challenges including limited unit test support for mobile specific features, better monitoring support, crash reports, emulators, missing platform-supported tools, rapid changes, multiple scenarios to validate, app stores and usability testing.

The study concluded that one of the most important challenges for developers is having to deal with multiple platforms, since the knowledge of one platform typically can't be transferred to another. In addition, tools to monitor and measure the performance of mobile apps are important for developers as are testing frameworks and tools. Our study differs from the goals of this paper in that we focus on examining the testing practices and preferences of open source developers whereas Erfani *et al.* [105] analyzed a variety of aspects of the entire software development process.

Kochhar *et al.* [114] conducted an empirical investigation into open source apps and two different surveys, the first one comprising three questions asked to 83 android developers, and the second comprising five questions as an improved version of the first study, posed to 127 windows app developers at Microsoft. The study includes questions to investigate techniques used to test apps, frameworks used, types of testing used, reasons for using testing tools, and challenges encountered during testing process. The authors concluded

that Android apps are not properly tested since around 86% of the apps do not contain any test cases. In addition, existing automated tools are not able to reach certain parts of code in mobile apps and are typically prohibitively difficult to use. Finally, the study found that developers are not aware of many existing testing tools. Our study differentiates itself in the fact that we attempted to distill developer's testing preferences and practices for both manual and automated practices in order to inform the development of more practical automated tools, and our participants were Android developers.

Choudhary *et al.* [64] presented a comparison between test input generation techniques for Android applications. Choudhary et al. studied these tools applied to 60 real-world applications considering four different criteria: (i) ease of use, (ii) android framework compatibility, (iii) code coverage achieved, and (iv) fault detection. The authors concluded that random testing (specifically Android Monkey) surpasses all other automated techniques. In contrast to this study we surveyed developers to investigate trends on usage of automated testing tools and experiences with random testing tools.

Linares-Vásquez *et al.* [133], recently conducted a survey of current tools, frameworks, and services available to support mobile testing practices. This survey draws comparisons between different testing techniques and solutions, describing the benefits, and delineating drawbacks and trade-offs between different approaches/tools. Additionally, the work offers a forward-thinking vision for effective mobile testing along three principles: Continuous, Evolutionary, and Large-Scale. While this work offers a valuable perspective on the current state and potential future of mobile app testing, it does not survey developers to understand current mobile testing trends.

Aho *et al.* [39], presented an industrial evaluation of the *Murphy tool* that models the graphical user interface to support several testing tasks during the software development cycle. The experiences presented in the paper were based on the evaluation of three software systems and three test engineers from industry. The Murphy tool decreased the time and effort of generating test cases from the model. The authors concluded that Murphy helped to minimize the tedious and repetitive work while creating manual test

cases that involves analysis and verification from the tester. Compared to this study, we do not focus on the evaluation of one particular approach rather we surveyed open source developers about the usage of different automated testing tools, and preferences for ideal automated testing techniques.

2.6 Lessons Learned

In this work we presented the results of an empirical study with 102 contributors of open source mobile apps hosted at GitHub. In particular, the study was conducted with a survey aimed at gathering information about their practices in-the-wild and preferences for (i) designing and generating test cases, (ii) using automated approaches, and (iii) assessing the quality of test suites.

Our survey reveals highly relevant opinions of open source developers such as they rely primarily on usage models (e.g, use cases, user stories) of their applications when designing test cases, and they prefer high-level expressive automatically generated test cases organized around use-cases. As of today, little effort has been devoted to include usage models [188, 135, 115] during automated test cases generation for mobile apps; thus, usage models and expressive test cases should be considered as an important goal for automated approaches/tools for mobile testing. The survey results support the need for multi-models in model-based testing as suggested by Linares-Vásquez *et al.* [133].

Another result we would like to highlight is the fact that code coverage is not perceived by the survey participants as an important measure of test cases quality. While code coverage has been widely used by researchers to validate automated approaches for testing mobile apps, this result could be used as an insight that reinforce the discussion regarding code coverage utility [101, 109]. Additionally, this should spur the discussion and creation of new evaluation models — for new testing approaches/tools — that consider other criteria such as relevant fault detection capability (e.g. faults along heavily traversed parts of the app) and feature coverage.

Finally, our survey confirms the fact that despite the plethora of tools proposed by the research community, the state-of-the-practice for automated testing are automation APIs; manually written test cases with automation APIs are very fragile to changes in the GUI of the app under test [105, 133]; even the official Google tool for random/fuzz testing (i.e., Monkey) has a low usage rate. In order to aid in technology transfer, researchers should consider developer preferences and workflows when designing and evaluating their approaches. Such preferences can be gleaned from the developer responses in this work, and include among others: (i) A need for automatically generated test cases to co-evolve with apps and features, (ii) low-overhead tools that tightly integrate into current (agile) development workflows, (iii) expressive test cases that allow for easier debugging and traceability between test cases and features, and (iv) alternative metrics and techniques to validate effectiveness and robustness of test suits. By taking such preferences into consideration, researchers should be able to design approaches that make a meaningful impact during real mobile testing practices.

2.7 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary:

Linares-Vásquez, M., **Bernal-Cárdenas, C.**, Moran, K. and Poshyvanyk, D., "How do developers test android applications?", in Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME'17), Shanghai, China, September 20-22, 2017, pp. 613-622

Chapter 3

Enabling Mutation Testing for Android Apps

3.1 Introduction

In the last few years mobile apps have become indispensable in our daily lives. With millions of mobile apps available for download on Google Play [88] and the Apple App Store [48], mobile users have access to an unprecedentedly large set of apps that are not only intended to provide entertainment but also to support critical activities such as banking and health monitoring. Therefore, given the increasing relevance and demand for high quality apps, industrial practitioners and academic researchers have been devoting significant effort to improving methods for measuring and assuring the quality of mobile apps. Manifestations of interest in this topic include the broad portfolio of mobile testing methods ranging from tools for assisting record and replay testing [82, 99], to automated approaches that generate and execute test cases [138, 64, 135, 157, 141], and cloud-based services for large-scale multi-device testing [23].

Despite the availability of these tools/approaches, the field of mobile app testing is still very much under development; as evidenced by limitations related to test data generation [64, 133], and concerns regarding effective assessment of the quality of mobile apps' test

suites. One way to evaluate test suites is to seed small faults, called mutants, into source code and asses the ability of a suite to detect these faults [92, 71]. Such mutants have been defined in the literature to reflect the typical errors developers make when writing source code [136, 113, 147, 166, 199, 161, 178].

However, existing literature lacks a thorough characterization of bugs exhibited by mobile apps. Therefore, it is unclear whether such apps exhibit a distribution of faults similar to other systems, or if there are types of faults that require special attention. As a consequence, it is unclear whether the use of traditional mutant taxonomies [136, 113] is enough to asses test quality and drive test case generation/selection of mobile apps.

In this paper, we explore this topic focusing on apps developed for Android, the most popular mobile operating system. Android apps are characterized by GUI-centric design/interaction, event-driven programming, Inter Processes Communication (IPC), and interaction with backend and local services. In addition, there are specific characteristics of Android apps—such as permission mechanisms, Software Development Kit (SDK) version compatibility, or features of target devices—that can lead to a failure. While this set of characteristics would demand a specialized set of mutation operators that can support mutation analysis and testing, there is no available tool to date that supports mutation analysis/testing of Android apps, and relatively few (eight) mutation operators have been proposed by the research community [72]. At the same time, mutation tools for Java apps, such as Pit [24] and Major [107, 111] lack any Android-specific mutation operators, and present challenges for their use in this context, resulting in common problems such as trivial mutants that always crash at runtime or difficulties automating mutant compilation into Android PacKages (APKs).

Paper contributions. This paper aims to deal with the lack of (i) an extensive empirical evidence of the distribution of Android faults, (ii) a thorough catalog of Android-specific mutants, and (iii) an analysis of the applicability of state-of-the-art mutation tools on Android apps. We then propose a framework, MDroid+, that relies on a catalog of mutation operators inspired by a taxonomy of bugs/crashes specific for Android apps, and

a profile of potential failure points automatically extracted from APKs.

As a first step, we produced a taxonomy of Android faults by analyzing a statistically significant sample of 2,023 candidate faults documented in (i) bug reports from open source apps, (ii) bug-fixing commits of open source apps; (iii) Stack Overflow discussions, (iv) the Android exception hierarchy and APIs potentially triggering such exceptions; and (v) crashes/bugs described in previous studies on Android [138, 175, 201, 197, 120, 38, 64, 155, 157]. As a result, we produced a taxonomy of 262 types of faults grouped in 14 categories, four of which relate to Android-specific faults, five to Java-related faults, and five mixed categories (Fig. 3.1). Then, leveraging this fault taxonomy and focusing on Android-specific faults, we devised a set of 38 Android mutation operators and implemented a platform to automatically seed 35 of them. Finally, we conducted a study comparing MDroid+ with other Java and Android-specific mutation tools. The study results indicate that MDroid+, as compared to existing competitive tools, (i) is able to cover a larger number of bug types/instances present in Android app, (ii) is highly complementary to the existing tools in terms of covered bug types, and (iii) generates fewer trivial and stillborn mutants.

3.2 Related Work

This section describes related literature and publicly available, state-of-the-art tools on mutation testing. We do not discuss the literature on testing Android apps [43, 94, 140, 138, 64, 135, 157, 141, 133], since proposing a novel approach for testing Android apps is not the main goal of this work. For further details about the concepts, recent research, and future work in the field of mutation testing, one can refer to the survey by Jia and Harman [103].

Mutation Operators. Since the introduction of mutation testing in the 70s [92, 71], researchers have tried not only to define new mutation operators for different programming languages and paradigms (*e.g.,* mutation operators have been defined for Java [136] and

Python [73]) but also for specific types of software like Web applications [173] and data-intensive applications [47, 203] either to exercise their GUIs [165] or to alter complex, model-defined input data [74]. The aim of our research, which we share with prior work, is to define customized mutation operators suitable for Android applications, by relying on a solid empirical foundation.

To the best of our knowledge, the closest work to ours is that of Deng *et al.*, [72], which defined eight mutant operators aimed at introducing faults in the essential programming elements of Android apps, *i.e.,* intents, event handlers, activity lifecycle, and XML files (*e.g.,* GUI or permission files). While we share with Deng *et al.* the need for defining specific operators for the key Android programming elements, our work builds upon it by (i) empirically analyzing the distribution of faults in Android apps by manually tagging 2,023 documents, (ii) based on this distribution, defining a mutant taxonomy—complementing Java mutants—which includes a total of 38 operators tailored for the Android platform.

Mutation Testing Effectiveness and Efficiency. Several researchers have proposed approaches to measure the effectiveness and efficiency of mutation testing [163, 45, 89, 110] to devise strategies for reducing the effort required to generate effective mutant sets [37, 108, 90], and to define theoretical frameworks [103, 185]. Such strategies can complement our work, since in this paper we aim at defining new mutant operators for Android, on which effectiveness/efficiency measures or minimization strategies can be applied.

Mutation Testing Tools. Most of the available mutation testing tools are in the form of research prototypes. Concerning Java, representative tools are μJava [137], Jester [148], Major [107], Jumble [189], PIT [24], and javaLanche [182]. Some of these tools operate on the Java source code, while others inject mutants in the bytecode. For instance, μJava, Jester, and Major generate the mutants by modifying the source code, while Jumble, PIT, and javaLanche perform the mutations in the bytecode. When it comes to Android apps, there is only one available tool, namely muDroid [196], which performs the mutations at byte code level by generating one APK (*i.e.,* one version of the mobile app) for each mutant.

The tools for mutation testing can be also categorized according to the tool's capabilities (*e.g.,* the availability of automatic tests selection). A thorough comparison of these tools is out of the scope of this paper. The interested reader can find more details on PIT's website [68] and in the paper by Madeysky and Radyk [139].

Empirical Studies on Mutation Testing. Daran and Thévenod-Fosse [70] were the first to empirically compare mutants and real faults, finding that the set of errors and failures they produced with a given test suite were similar. Andrews *et al.* [45, 46] studied whether mutant-generated faults and faults seeded by humans can be representative of real faults. The study showed that carefully-selected mutants are not easier to detect than real faults, and can provide a good indication of test suite adequacy, whereas human-seeded faults can likely produce underestimates. Just *et al.* [109] correlated mutant detection and real fault detection using automatically and manually generated test suites. They found that these two variables exhibit a statistically significant correlation. At the same time, their study pointed out that traditional Java mutants need to be complemented by further operators, as they found that around 17% of faults were not related to mutants (i.e., due to algorithmic changes and code deletion).

3.3 A Taxonomy of Crashes/Bugs in Android apps

To the best of our knowledge there is currently no (i) large-scale study describing a taxonomy of bugs in Android apps, or (ii) comprehensive mutation framework including operators derived from such a taxonomy and targeting mobile-specific faults (the only framework available is the one with eight mutation operators proposed by Deng *et al.* [72]). In this section, we describe a taxonomy of bugs in Android apps derived from a large manual analysis of (un)structured sources. Our work is the first large-scale data driven effort to design such a taxonomy. Our purpose is to extend/complement previous studies analyzing bugs/crashes in Android apps and to provide a large taxonomy of bugs that can be used to design mutation operators. In all the cases reported below the manually analyzed sets of

sources—randomly extracted—represent a 95% statistically significant sample with a 5% confidence interval.

3.3.1 Design

To derive such a taxonomy we manually analyzed six different sources of information described below:

1. *Bug reports of Android open source apps.* Bug reports are the most obvious source to mine in order to identify typical bugs affecting Android apps. We mined the issue trackers of 16,331 open source Android apps hosted on GitHub. Such apps have been identified by locally cloning all Java projects (381,161) identified through GitHub's API and searching for projects with an *AndroidManifest.xml* file (a requirement for Android apps) in the top-level directory. We then removed forked projects to avoid duplicated apps and filtered projects that did not have a single star or watcher to avoid abandoned apps. We utilized a web crawler to mine the GitHub issue trackers. To be able to analyze the bug cause, we only selected closed issues (*i.e.,* those having a fix that can be inspected) having "Bug" as type. Overall, we collected 2,234 issues from which we randomly sampled 328 for manual inspection.

2. *Bug-fixing commits of Android open source apps.* Android apps are often developed by very small teams [105, 158]. Thus, it is possible that some bugs are not documented in issue trackers but quickly discussed by the developers and then directly fixed. This might be particularly true for bugs having a straightforward solution. Thus, we also mined the versioning system of the same 16,331 Android apps considered for the bug reports by looking for bug-fixing commits not related to any of the bugs considered in the previous point (*i.e.,* the ones documented in the issue tracker). With the cloned repositories, we utilized the *git* command line utility to extract the commit notes and matched the ones containing lexical patterns indicating bug fixing activities, *e.g.,"fix issue"*, *"fixed bug"*, similarly to the approach proposed by Fischer

et al. [78]. By exploiting this procedure we collected 26,826 commits, from which we randomly selected a statistically significant sample of 376 commits for manual inspection.

3. *Android-related Stack Overflow (SO) discussions.* It is not unusual for developers to ask help on SO for bugs they are experiencing and having difficulty fixing [131, 58, 112, 179]. Thus, mining SO discussions could provide additional hints on the types of bugs experienced by Android developers. To this aim, we collected all 51,829 discussions tagged "Android" from SO. Then, we randomly extracted a statistically significant sample of 377 of them for the manual analysis.

4. *The exception hierarchy of the Android APIs.* Uncaught exceptions and statements throwing exceptions are a major source of faults in Android apps [201, 67]. We automatically crawled the official Android developer JavaDoc guide to extract the exception hierarchy and API methods throwing exceptions. We collected 5,414 items from which we sampled 360 of them for manual analysis.

5. *Crashes/bugs described in previous studies on Android apps.* 43 papers related to Android testing[1] were analyzed by looking for crashes/bugs reported in the papers. For each identified bug, we kept track of the following information: app, version, bug id, bug description, bug URL. When we were not able to identify some of this information, we contacted the paper's authors. In the 43 papers, a total of 365 bugs were mentioned/reported; however, we were able (in some cases with the authors' help) to identify the app and the bug descriptions for only 182 bugs/issues (from nine papers [138, 175, 201, 197, 120, 38, 64, 155, 157]). Given the limited number, in this case we considered all of them in our manual analysis.

6. *Reviews posted by users of Android apps on the Google Play store.* App store reviews have been identified as a prominent source of bugs and crashes in mobile

[1]The complete list of papers is provided with our online appendix [127].

apps [167, 100, 112, 172, 170, 134]. However, only a reduced set of reviews are in fact informative and useful for developers [62, 170]. Therefore, to automatically detect informative reviews reporting bugs and crashes, we leverage CLAP, the tool developed by Villarroel *et al.* [191], to automatically identify the bug-reporting reviews. Such a tool has been shown to have a precision of 88% in identifying this specific type of review. We ran CLAP on the Android user reviews dataset made available by Chen *et al.* [61]. This dataset reports user reviews for multiple releases of ∼21K apps, in which CLAP identified 718,132 reviews as bug-reporting. Our statistically significant sample included 384 reviews that we analyzed.

The data collected from the six sources listed above was manually analyzed by the eight authors following a procedure inspired by open coding [144]. In particular, the 2,007 documents (*e.g.,* bug reports, user reviews, *etc.*) to manually validate were equally and randomly distributed among the authors making sure that each document was classified by two authors. The goal of the process was to identify the exact reason behind the bug and to define a tag (*e.g., null GPS position*) describing such a reason. Thus, when inspecting a bug report, we did not limit our analysis to the reading of the bug description, but we analyzed (i) the whole discussion performed by the developers, (ii) the commit message related to the bug fixing, and (iii) the patch used to fix the bug (*i.e.,* source code diff). The tagging process was supported by a Web application that we developed to classify the documents (*i.e.,* to describe the reason behind the bug) and to solve conflicts between the authors. Each author independently tagged the documents assigned to him by defining a tag describing the cause behind a bug. Every time the authors had to tag a document, the Web application also shows the list of tags created so far, allowing the tagger to select one of the already defined tags. Although, in principle, this is against the notion of open coding, in a context like the one encountered in this work, where the number of possible tags (*i.e.,* cause behind the bug) is extremely high, such a choice helps using consistent naming and does not introduce a substantial bias.

In cases for which there was no agreement between the two evaluators (∼43% of the classified documents), the document was automatically assigned to an additional evaluator. The process was iterated until all the documents were classified by the absolute majority of the evaluators with the same tag. When there was no agreement after all eight authors tagged the same document (*e.g.,* four of them used the tag t_1 and the other four the tag t_2), two of the authors manually analyzed these cases in order to solve the conflict and define the most appropriate tag to assign (this happened for ∼22% of the classified documents). It is important to note that the Web application did not consider documents tagged as *false positive* (*e.g.,* a bug report that does not report an actual bug in an Android app) in the count of the documents manually analyzed. This means that, for example, to reach the 328 bug reports to manually analyze and tag, we had to analyze 400 bug reports (since 72 were tagged as false positives).

It is important to point out that, during the tagging, we discovered that for user reviews, except for very few cases, it was impossible (without internal knowledge of an app's source code) to infer the likely cause of the failure (fault) by only relying on what was described in the user review. For this reason, we decided to discard user reviews from our analysis, and this left us with 2,007-384=1,623 documents to manually analyze.

After having manually tagged all the documents (overall, 2,023 = 1,623 + 400 additional documents, since 400 false positives were encountered in the tagging process), all the authors met online to refine the identified tags by merging similar ones and splitting generic ones when needed. Also, in order to build the fault taxonomy, the identified tags were clustered in cohesive groups at two different levels of abstraction, *i.e.,* categories and subcategories. Again, the grouping was performed over multiple iterations, in which tags were moved across categories, and categories merged/split.

Finally, the output of this step was (i) a taxonomy of representative bugs for Android apps, and (ii) the assignment of the analyzed documents to a specific tag describing the reason behind the bug reported in the document.

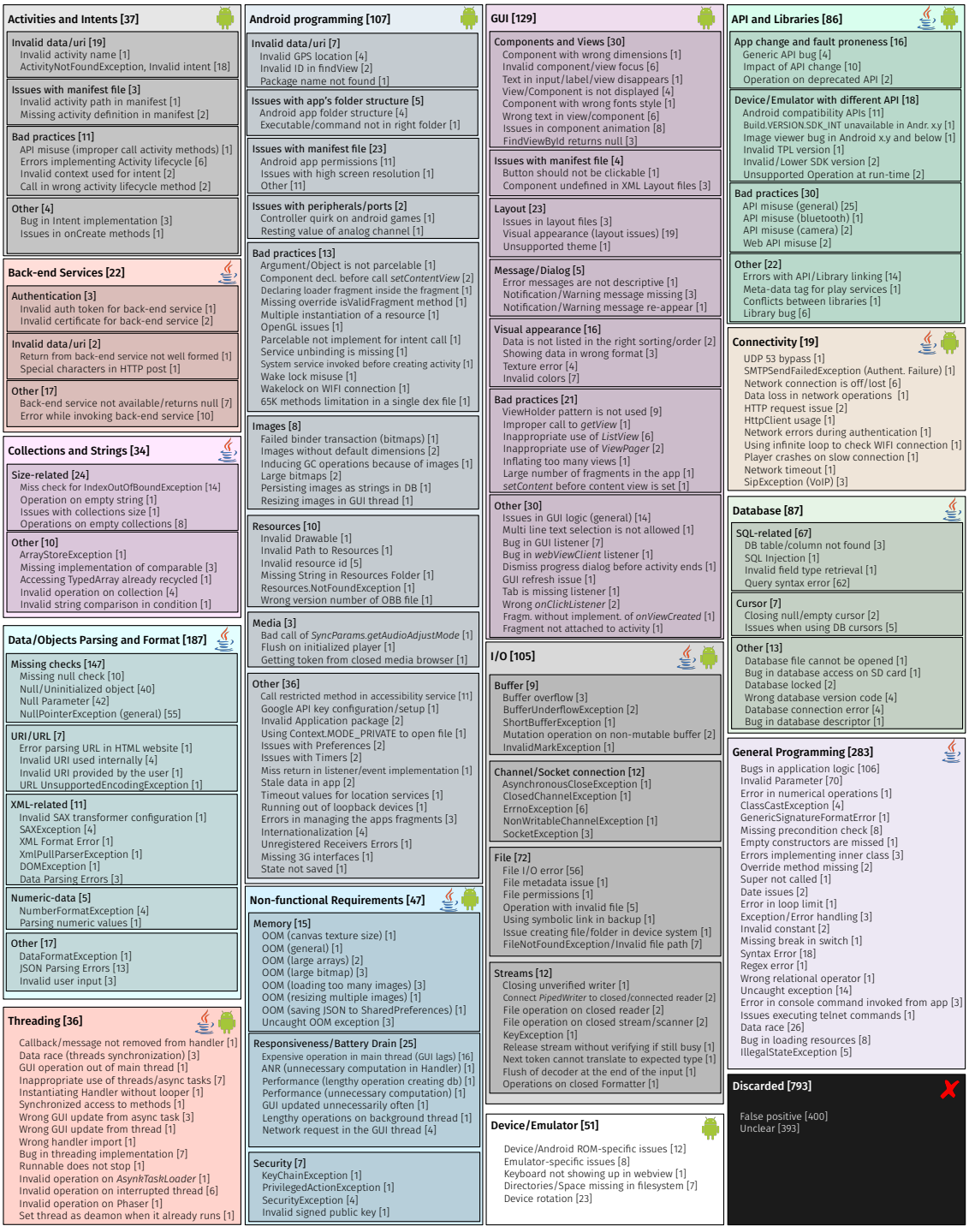


Figure 3.1: The defined taxonomy of Android bugs.

3.3.2 The Defined Taxonomy

Fig. 3.1 depicts the taxonomy that we obtained through the manual coding. The black rectangle in the bottom-right part of Fig. 3.1 reports the number of documents tagged as *false positive* or as *unclear*. The other rectangles—marked with the Android and/or with the Java logo—represent the 14 high-level categories that we identified. Categories marked with the Android logo (*e.g.,* Activities and Intents) group together Android-specific bugs while those marked with the Java logo (*e.g.,* Collections and Strings) group bugs that could affect any Java application. Both symbols together indicate categories featuring both Android-specific and Java-related bugs (see *e.g.,* I/O). The number reported in square brackets indicates the bug instances (from the manually classified sample) belonging to each category. Inner rectangles, when present, represent sub-categories, *e.g., Responsiveness/Battery Drain* in *Non-functional Requirements*. Finally, the most fine-grained levels, represented as lighter text, describe the specific type of faults as labeled using our manually-defined tags, *e.g.,* the *Invalid resource ID* tag under the sub-category *Resources*, in turn part of the *Android programming* category. The analysis of Fig. 3.1 allows to note that:

1. *We were able to classify the faults reported in 1,230 documents (e.g., bug reports, commits, etc.).* This number is obtained by subtracting from the 2,023 tagged documents the 400 tagged as *false positives* and the 393 tagged as *unclear*.

2. *Of these 1,230, 26% (324) are grouped in categories only reporting Android-related bugs.* This means that more than one fourth of the bugs present in Android apps are specific of this architecture, and not shared with other types of Java systems. Also, this percentage clearly represents an underestimation. Indeed, Android-specific bugs are also present in the previously mentioned "mixed" categories (*e.g.,* in *Non-functional requirements* 25 out of the 26 instances present in the *Responsiveness/-Battery Drain* subcategory are Android-specific—all but *Performance (unnecessary computation)*). From a more detailed count, after including also the Android-specific

bugs in the “mixed" categories, we estimated that 35% (430) of the identified bugs are Android-specific.

3. *As expected, several bugs are related to simple Java programming.* This holds for 800 of the identified bugs (65%).

Take-away. Over one third (35%) of the bugs we identified with manual inspection are Android-specific. This highlights the importance of having testing instruments, such as mutation operators, tailored for such a specific type of software. At the same time, 65% of the bugs that are typical of any Java application confirm the importance of also considering standard testing tools developed for Java, including mutation operators, when performing verification and validation activities of Android apps.

3.4 Mutation Operators for Android

Given the taxonomy of faults in Android apps and the set of available operators widely used for Java applications, a catalog of Android-specific mutation operators should (i) complement the classic Java operators, (ii) be representative of the faults exhibited by Android apps, (iii) reduce the rate of still-born and trivial mutants, and (iv) consider faults that can be simulated by modifying statements/elements in the app source code and resources (*e.g.,* the `strings.xml` file). The last condition is based on the fact that some faults cannot be simulated by changing the source code, like in the case of device specific bugs, or bugs related to the API and third-party libraries.

Following the aforementioned conditions, we defined a set of 38 operators, trying to cover as many fault categories as possible (10 out of the 14 categories in Fig. 3.1), and complementing the available Java mutation operators. The reasons for not including operators from the other four categories are:

1. API/Libraries: bugs in this category are related to API/Library issues and API misuses. The former will require applying operators to the APIs; the latter requires

a deeper analysis of the specific API usage patterns inducing the bugs;

2. Collections/Strings: most of the bugs in this category can be induced with classic Java mutation operators;

3. Device/Emulator: because this type of bug is Device/Emulator specific, their implementation is out of the scope of source code mutations;

4. Multi-threading: the detection of the places for applying the corresponding mutations is not trivial; therefore, this category will be considered in future work.

The list of defined mutation operators is provided in Table 3.1 and these operators were implemented in a tool named MDroid+. In the context of this paper, we define a Potential Failure Profile (PFP) that sipulates locations of the analyzed apps—which can be source code statements, XML tags or locations in other resource files—that can be the source of a potential fault, given the faults catalog from Sec. 3.3. Consequently, the PFP lists the locations where a mutation operator can be applied.

In order the extract the PFP, MDroid+ statically analyzes the targeted mobile app, looking for locations where the operators from Table 3.1 can be implemented. The locations are detected automatically by parsing XML files or through AST-based analysis for detecting the location of API calls. Given an automatically derived PFP for an app, and the catalog of Android-specific operators, MDroid+ generates a mutant for each location in the PFP. Mutants are initially generated as clones (at source code-level) of the original app, and then the clones are automatically compiled/built into individual Android Packages (APKs). Note that each location in the PFP is related to a mutation operator. Therefore, given a location entry in the PFP, MDroid+ automatically detects the corresponding mutation operator and applies the mutation in the source code. Details of the detection rules and code transformations applied with each operator are provided in our replication package [127].

Table 3.1: Proposed mutation operators. The table lists the operator names, detection strategy (AST or TEXTual), the fault category (Activity/Intents, Android Programming, Back-End Services, Connectivity, Data, DataBase, General Programming, GUI, I/O, Non-Functional Requirements), and a brief operator description. The operators indicated with * are not implemented in MDroid+ yet.

Mutation Operator	Det.	Cat.	Description
ActivityNotDefined	Text	A/I	Delete an activity <android:name="Activity"/> entry in the Manifest file
DifferentActivityIntentDefinition	AST	A/I	Replace the Activity.class argument in an Intent instantiation
InvalidActivityName	Text	A/I	Randomly insert typos in the path of an activity defined in the Manifest file
InvalidKeyIntentPutExtra	AST	A/I	Randomly generate a different key in an Intent.putExtra(key, value) call
InvalidLabel	Text	A/I	Replace the attribute "android:label" in the Manifest file with a random string
NullIntent	AST	A/I	Replace an Intent instantiation with null
NullValueIntentPutExtra	AST	A/I	Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable[0]
WrongMainActivity	Text	A/I	Randomly replace the main activity definition with a different activity
MissingPermissionManifest	Text	AP	Select and remove an <uses-permission /> entry in the Manifest file
NotParcelable	AST	AP	Select a parcelable class, remove"implements Parcelable" and the @override annotations
NullGPSLocation	AST	AP	Inject a Null GPS location in the location services
SDKVersion	Text	AP	Randomly mutate the integer values in the SdkVersion-related attributes
WrongStringResource	Text	AP	Select a <string /> entry in /res/values/strings.xml file and mutate the string value
NullBackEndServiceReturn	AST	BES	Assign null to a response variable from a back-end service
BluetoothAdapterAlwaysEnabled	AST	C	Replace a BluetoothAdapter.isEnabled() call with"tru"
NullBluetoothAdapter	AST	C	Replace a BluetoothAdapter instance with null
InvalidURI	AST	D	If URIs are used internally, randomly mutate the URIs
ClosingNullCursor	AST	DB	Assign a cursor to null before it is closed
InvalidIndexQueryParameter	AST	DB	Randomly modify indexes/order of query parameters
InvalidSQLQuery	AST	DB	Randomly mutate a SQL query
InvalidDate	AST	GP	Set a random Date to a date object
InvalidMethodCallArgument*	AST	GP	Randomly mutate a method call argument of a basic type
NotSerializable	AST	GP	Select a serializable class, remove "implements Serializable"
NullMethodCallArgument*	AST	GP	Randomly set null to a method call argument
BuggyGUIListener	AST	GUI	Delete action implemented in a GUI listener
FindViewByIdReturnsNull	AST	GUI	Assign a variable (returned by Activity.findViewById) to null
InvalidColor	Text	GUI	Randomly change colors in layout files
InvalidIDFindView	AST	GUI	Replace the id argument in an Activitity.findViewById call
InvalidViewFocus*	AST	GU	IRandomly focus a GUI component
ViewComponentNotVisible	AST	GUI	Set visible attribute (from a View) to false
InvalidFilePath	AST	I/O	Randomly mutate paths to files
NullInputStream	AST	I/O	Assign an input stream (*e.g.,* reader) to null before it is closed
NullOutputStream	AST	I/O	Assign an output stream (*e.g.,* writer) to null before it is closed
LengthyBackEndService	AST	NFR	Inject large delay right-after a call to a back-end service
LengthyGUICreation	AST	NFR	Insert a long delay (*i.e.,* Thread.sleep(..)) in the GUI creation thread
LengthyGUIListener	AST	NFR	Insert a long delay (*i.e.,* Thread.sleep(..)) in the GUI listener thread
LongConnectionTimeOut	AST	NFR	Increase the time-out of connections to back-end services
OOMLargeImage	AST	NFR	Increase the size of bitmaps by explicitly setting large dimensions

It is worth noting that from our catalog of Android-specific operators only two operators (DifferentActivityIntentDefinition and MissingPermissionManifest) overlap with the eight operators proposed by Deng *et al.*, [72]. Future work will be devoted to cover a larger number of fault categories and define/implement a larger number of operators.

3.5 Applying Mutation Testing Operators to Android Apps

The *goal* of this study is to: (i) understand and compare the ***applicability*** of MDroid+ and other currently available mutation testing tools to Android apps; (ii) to understand the ***underlying reasons*** for mutants—generated by these tools—that cannot be considered useful for the mutant analysis purposes, *i.e.,* mutants that do not compile or cannot be launched. This study is conducted from the *perspective* of researchers interested in improving current tools and approaches for mutation testing in the context of mobile apps. The study addresses the following research questions:

- **RQ_1:** *Are the mutation operators (available for Java and Android apps) representative of real bugs in Android apps?*
- **RQ_2:** *What is the rate of stillborn mutants (e.g., those leading to failed compilations) and trivial mutants (e.g., those leading to crashes on app launch) produced by the studied tools when used with Android apps?*
- **RQ_3:** *What are the major causes for stillborn and trivial mutants produced by the mutation testing tools when applied to Android apps?*

To answer $\mathbf{RQ}_1$, we measured the applicability of operators from seven mutation testing tools (Major [107], PIT [24], μJava [137], Javalanche [182], muDroid [196], Deng *et al.* [72], and MDroid+) in terms of their ability of representing real Android apps' faults documented in a sample of software artifacts not used to build the taxonomy presented in Sec. 3.3. To answer $\mathbf{RQ}_2$, we used a representative subset of the aforementioned tools to generate mutants for 55 open source Android apps, quantitatively and qualitatively

examining the stillborn and trivial mutants generated by each tool. Finally, to answer $\mathbf{RQ}_3$, we manually analyzed the mutants and their crash outputs to qualitatively determine the reasons for trivial and stillborn mutants generated by each tool.

3.5.1 Study Context and Data Collection

To answer $\mathbf{RQ}_1$, we analyzed the complete list of 102 mutation operators from the seven considered tools to investigate their ability to "*cover*" bugs described in 726 artifacts[2] (103 exceptions hierarchy and API methods throwing exceptions, 245 bug-fixing commits from GitHub, 176 closed issues from GitHub, and 202 questions from SO). Such 726 documents were randomly selected from the dataset built for the taxonomy definition (see Sec. 3.3.1) by excluding the ones already tagged and used in the taxonomy. The documents were manually analyzed by the eight authors using the same exact procedure previously described for the taxonomy building (*i.e.,* two evaluators per document having the goal of tagging the type of bug described in the document; conflicts solved by using a majority-rule schema; tagging process supported by a Web app—details in Sec. 3.3.1). We targeted the tagging of ∼150 documents per evaluator (600 overall documents considering eight evaluators and two evaluations per document). However, some of the authors tagged more documents, leading to the considered 726 documents. Note that we did not constrain the tagging of the bug type to the ones already present in our taxonomy (Fig. 3.1): The evaluations were free to include new types of previously unseen bugs.

We answer RQ_1 by reporting (i) the new bug types we identified in the tagging of the additional 726 documents (*i.e.,* the ones not present in our original taxonomy), (ii) the *coverage* level ensured by each of the seven mutation tools, measured as the percentage of bug types and bug instances identified in the 726 documents covered by its operators. We also analyze the complementarity of MDroid+ with respect to the existing tools.

Concerning $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we compare MDroid+ with two popular open source mutation testing tools (Major and PIT), which are available and can be tailored for Android

[2]With "cover" we mean the ability to generate a mutant simulating the presence of a give type of bug.

apps, and with one context-specific mutation testing tool for Android called muDroid [72]. We chose these tools because of their diversity (in terms of functionality and mutation operators), their compatibility with Java, and their representativeness of tools working at different representation levels: source code, Java bytecode, and smali bytecode (*i.e.,* Android-specific bytecode representation).

To compare the applicability of each mutation tool, we need a set of Android apps that meet certain constraints: (i) the source code of the apps must be available, (ii), the apps should be representative of different categories, and (iii) the apps should be compilable (*e.g.,* including proper versions of the external libraries they depend upon). For these reasons, we use the Androtest suite of apps [64], which includes 68 Android apps from 18 Google Play categories. These apps have been previously used to study the design and implementation of automated testing tools for Android and met the three above listed constraints. The mutation testing tools exhibited issues in 13 of the considered 68 apps, *i.e.,* the 13 apps did not compile after injecting the faults. Thus, in the end, we considered 55 subject apps in our study. The list of considered apps as well as their source code is available in our replication package [127].

Note that while Major and PIT are compatible with Java applications, they cannot be directly applied to Android apps. Thus, we wrote specific wrapper programs to perform the mutation, the assembly of files, and the compilation of the mutated apps into runnable Android application packages (*i.e., APKs*). While the procedure used to generate and compile mutants varies for each tool, the following general workflow was used in our study: (i) generate mutants by operating on the original source/byte/smali code using all possible mutation operators; (ii) compile or assemble the APKs either using the `ant`, `dex2jar`, or `baksmali` tools; (iii) run all of the apps in a parallel-testing architecture that utilizes Android Virtual Devices (AVDs); (iv) collect data about the number of apps that crash on launch and the corresponding exceptions of these crashes which will be utilized for a manual qualitative analysis. We refer readers to our replication package for the complete technical methodology used for each mutation tool [127].

To quantitatively assess the applicability and effectiveness of the considered mutation tools to Android apps, we used three metrics: **Total Number of Generated Mutants (TNGM)**, **Stillborn Mutants (SM)**, and **Trivial Mutants (TM)**. In this paper, we consider *stillborn mutants* as those that are syntactically incorrect to the point that the APK file cannot be compiled/assembled, and *trivial mutants* as those that are killed arbitrarily by nearly any test case. If a mutant crashes upon launch, we consider it as a trivial mutant. Another metric one might consider to evaluate the effectiveness of a mutation testing tool is the number of equivalent and redundant mutants the tool produces. However, in past work, the identification of equivalent mutants has been proven to be an undecidable problem [164], and both equivalent and redundant mutants require the existence of test suites (not available for the Androtest apps). Therefore, this aspect is not studied in our work.

After generating the mutants' `APKs` using each tool, we needed a viable methodology for launching all these mutants in a reasonable amount of time to determine the number of *trivial mutants*. To accomplish this, we relied on a parallel Android execution architecture that we call the Execution Engine (EE). EE utilizes concurrently running instances of Android Virtual Devices based on the `android-x86` project [87]. Specifically, we configured 20 AVDs with the `android-x86` v4.4.2 image, a screen resolution of 1900x1200, and 1GB of RAM to resemble the hardware configuration of a Google Nexus 7 device. We then concurrently instantiated these AVDs and launched each mutant, identifying app crashes.

3.5.2 Results

RQ_1: Fig. 3.2 reports (i) the percentage of bug types identified during our manual tagging that are covered by the taxonomy of bugs we previously presented in Fig. 3.1 (top part of Fig. 3.2), and (ii) the coverage in terms of bug types as well as of instances of tagged bugs ensured by each of the considered mutation tools (bottom part). The data shown in Fig. 3.2 refers to the 413 bug instances for which we were able to define the exact reason behind the bug (this excludes the 114 entities tagged as *unclear* and the 199 identified as

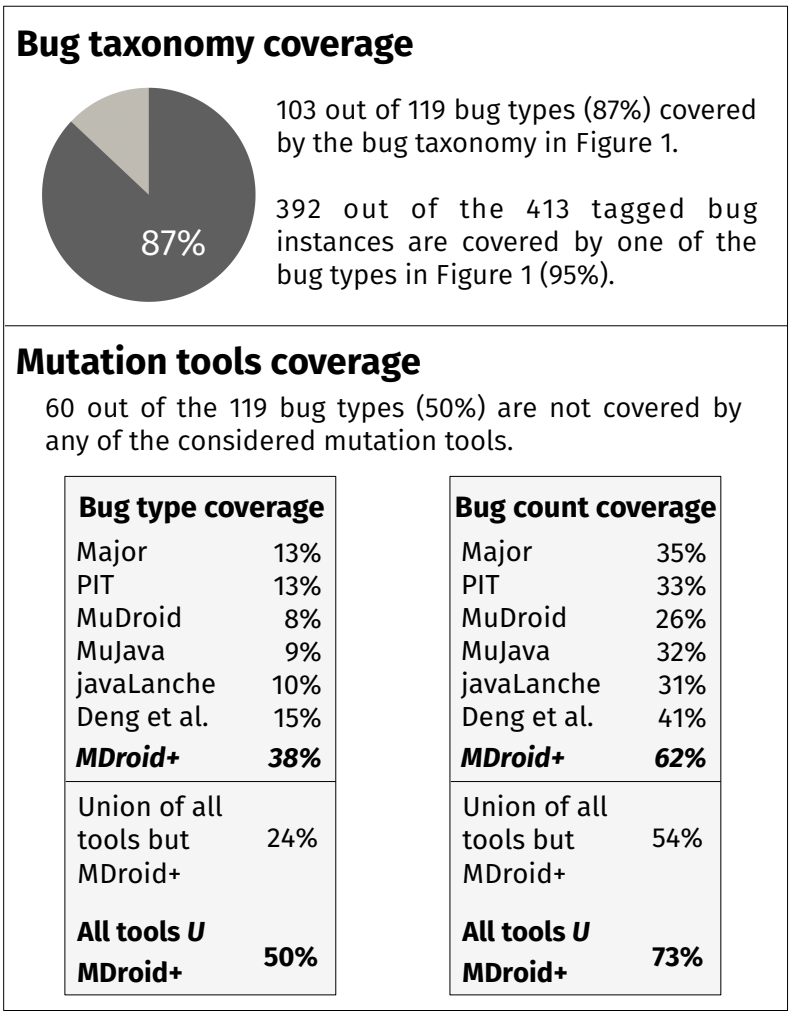


Figure 3.2: Mutation tools and coverage of analyzed bugs.

false positives).

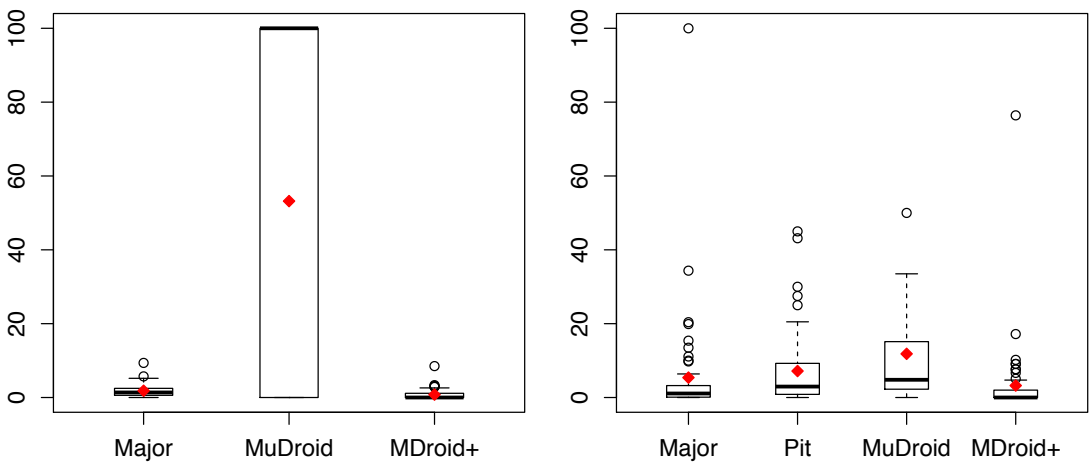
87% of the bug types are covered in our taxonomy. In particular, we identified 16 new categories of bugs that we did not encounter before in the definition of our taxonomy (Sec. 3.3). Examples of these categories (full list in our replication package) are: *Issues with audio codecs*, *Improper implementation of sensors as Activities*, and *Improper usage of the static modifier*. Note that these categories just represent a minority of the bugs we analyzed, accounting all together for a total of 21 bugs (5% of the 413 bugs considered). Thus, our bug taxonomy covers 95% of the bug instances we found, indicating a very good coverage.

Moving to the bottom part of Fig. 3.2, our first important finding highlights the limitations of the experimented mutation tools (including MDroid+) in potentially unveiling the bugs subject of our study. Indeed, for 60 out of the 119 bug types (50%), none of the considered tools is able to generate mutants simulating the bug. This stresses the need for new and more powerful mutation tools tailored for mobile platforms. For instance, no tool is currently able to generate mutants covering the *Bug in webViewClient listener* and the *Components with wrong dimensions* bug types.

When comparing the seven mutation tools considered in our study, MDroid+ clearly stands out as the tool ensuring the highest coverage both in terms of bug types and bug instances. In particular, mutators generated by MDroid+ have the potential to unveil 38% of the bug types and 62% of the bug instances. In comparison, the best competitive tool (*i.e.,* the catalog of mutants proposed by Deng *et al.* [72]) covers 15% of the bug types (61% less as compared to MDroid+) and 41% of the bug instances (34% less as compared to MDroid+). Also, we observe that MDroid+ covers bug categories (and, as a consequence, bug instances) missed by all competitive tools. Indeed, while the union of the six competitive tools covers 24% of the bug types (54% of the bug instances), adding the mutation operators included in MDroid+ increases the percentage of covered bug types to 50% (73% of the bug instances). Examples of categories covered by MDroid+ and not by the competitive tools are: *Android app permissions*, thanks to the MissingPermissionManifest operator, and the *FindViewById returns null*, thanks to the FindViewByIdReturnsNull operator.

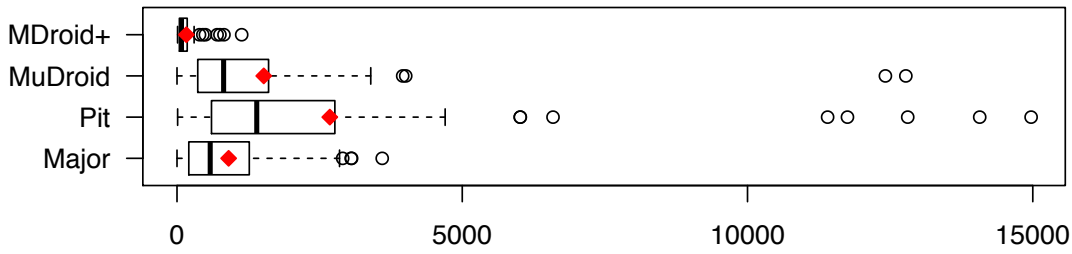
Finally, we statistically compared the proportion of bug types and the number of bug instances covered by MDroid+, by all other techniques, and by their combination, using Fisher's exact test and Odds Ratio (OR) [184]. The results indicate that:

1. The odds of covering bug types using MDroid+ are 1.56 times greater than other techniques, although the difference is not statistically significant (p-value=0.11). Similarly, the odds of discovering faults with MDroid+ are 1.15 times greater than



(a) %stillborn mutants

(b) %trivial mutants



(c) # of mutants per app

Figure 3.3: Stillborn and trivial mutants generated per app.

other techniques, but the difference is not significant (p-value=0.25);

2. The odds of covering bug types using MDroid+ combined with other techniques are 2.0 times greater than the other techniques alone, with a statistically significant difference (p-value=0.008). Similarly, the odds of discovering bugs using the combination of MDroid+ and other techniques are 1.35 times greater than other techniques alone, with a significant difference (p-value=0.008).

RQ$_2$: Figure 3.3 depicts the achieved results as percentage of (a) Stillborn Mutants (SM), and (b) Trivial Mutants (TM) generated by each tool on each app. On average, 167, 904, 2.6k+, and 1.5k+ mutants were generated by MDroid+, Major, PIT, and muDroid,

respectively for each app. The larger number of mutants generated by PIT is due in part to the larger number of mutation operators available for the tool. The average percentage of **stillborn mutants** (SM) generated by MDroid+, Major and muDroid over all the apps is 0.56%, 1.8%, and 53.9%, respectively, while no SM are generated by PIT (Figure 3.3a). MDroid+ produces significantly less SM than Major (Wilcoxon paired signed rank test p-value$<$ 0.001 – adjusted with Holm's correction [96], Cliff's d=0.59 - large) and than muDroid (adjusted p-value$<$ 0.001, Cliff's d=0.35 - medium).

These differences across the tools are mainly due to the compilation/assembly process they adopt during the mutation process. PIT works at Java bytecode level and thus can avoid the SM problem, at the risk of creating a larger number of TM. However, PIT is the tool that required the highest effort to build a wrapper to make it compatible with Android apps. Major works at the source code level and compiles the app in a "traditional" manner. Thus, it is prone to SM and requires an overhead in terms of memory and CPU resources needed for generating the mutants. Finally, muDroid operates on `APKs` and `smali` code, reducing the computational cost of mutant generation, but significantly increasing the chances of SM.

All four tools generated **trivial mutants** (TM) (*i.e.,* mutants that crashed simply upon launching the app). These instances place an unnecessary burden on the developer, particularly in the context of mobile apps, as they must be discarded from analysis. The mean of the distribution of the percentage of TM over all apps for MDroid+, Major, PIT and muDroid is 2.42%, 5.4%, 7.2%, and 11.8%, respectively (Figure 3.3b). MDroid+ generates significantly less TM than muDroid (Wilcoxon paired signed rank test adjusted p-value=0.04, Cliff's d=0.61 - large) and than PIT (adjusted p-value=0.004, Cliff's d=0.49 - large), while there is no statistically significant difference with Major (adjusted p-value=0.11).

While these percentages may appear small, the raw values show that the TM can comprise a large set of instances for tools that can generate thousands of mutants per app. For example, for the Translate app, 518 out of the 1,877 mutants generated by PIT

were TM. For the same app, muDroid creates 348 TM out of the 1,038 it generates. For the Blokish app, 340 out of the 3,479 mutants generated by Major were TM. Conversely, while MDroid+ may generate a smaller number of mutants per app, this also leads to a smaller number of TM, only 213 in total across all apps. This is due to the fact that MDroid+ generates a much smaller set of mutants that are specifically targeted towards emulating *real* faults identified in our empirically derived taxonomy, and are applied on specific locations detected by the PFP.

RQ$_3$: In terms of mutation operators causing the highest number of stillborn and TM we found that for Major, the Literal Value Replacement (LVR) operator had the highest number of TM, whereas the Relational Operator Replacement (ROR) had the highest number of SM. It may seem surprising that ROR generated many SM, however, we discovered that the reason was due to improper modifications of loop conditions. For instance, in the A2dp.Vol app one mutant changed this loop: `for (int i = 0; i < cols; i++)` and replaced the condition "$i < cols$" with "`false`", causing the compiler to throw an unreachable code error. For PIT, the Member Variable Mutator (MVM) is the one causing most of the TM; for muDroid, the Unary Operator Insertion (UOI) operator has the highest number of SM (although all the operators have relatively high failure rates), and the Relational Value Replacement (RVR) has the highest number of TM. For MDroid+, the WrongStringResource operator had that highest number of SM, whereas the FindViewByIdReturnsNull operator had the highest number of TM.

To qualitatively investigate the causes behind the crashes, three authors manually analyzed a randomly selected sample of 15 crashed mutants per tool. In this analysis, the authors relied on information about the mutation (*i.e.,* applied mutation operator and location), and the generated stack trace.

Major. The reasons behind the crashing mutants generated by Major mainly fall in two categories. First, mutants generated with the LVR operator that changes the value of a literal causing an app to crash. This was the case for the *wikipedia* app when changing the "1" in the invocation `setCacheMode(params.getString(1))` to "0". This passed

Table 3.2: Number of Generated, Stillborn, and Trivial Mutants created by MDroid+ operators.

Mutation Operators	GM	SM	TM	Mutation Operators	GM	SM	TM
WrongStringResource	3394	0	14	NullInputStream	61	0	4
NullIntent	559	3	41	WrongMainActivity	56	0	0
InvalidKeyIntentPutExtra	459	3	11	InvalidColor	52	0	0
NullValueIntentPutExtra	459	0	14	NullOuptutStream	45	0	2
InvalidIDFindView	456	4	30	InvalidDate	40	0	0
FindViewByIdReturnsNull	413	0	40	InvalidSQLQuery	33	0	2
ActivityNotDefined	384	1	8	NotSerializable	15	7	0
InvalidActivityName	382	0	10	NullBluetoothAdapter	9	0	0
DifferentActivityIntentDefinition	358	2	8	LengthyBackEndService	8	0	0
ViewComponentNotVisible	347	5	7	NullBackEndServiceReturn	8	1	0
MissingPermissionManifest	229	0	8	NotParcelable	7	6	0
InvalidFilePath	220	0	1	InvalidIndexQueryParameter	7	1	0
InvalidLabel	214	0	3	OOMLargeImage	7	4	0
ClosingNullCursor	179	13	5	BluetoothAdapterAlwaysEnabled	4	0	0
LengthyGUICreation	129	0	1	InvalidURI	2	0	0
BuggyGUIListener	122	0	2	NullGPSLocation	1	0	0
LengthyGUIListener	122	0	0	LongConnectionTimeOut	0	0	0
SDKVersion	66	0	2				
Total					8847	50	213

a wrong asset URL to the method `setCacheMode`, thus crashing the app. Second, the Statement Deletion (STD) operator was responsible for app crashes especially when it deleted needed methods' invocations. A representative example is the deletion of invocations to methods of the superclass when overriding methods, *e.g.,* when removing the `super.onDestroy()` invocation from the `onDestroy()` method of an `Activity`. This results in throwing of an `android.util.SuperNotCalledException`. Other STD mutations causing crashes involved deleting a statement initializing the main `Activity` leading to a `NullPointerException`.

muDroid. This tool is the one exhibiting the highest percentage of stillborn and TM. The most interesting finding of our qualitative analysis is that 75% of the crashing mutants lead to the throwing of a `java.lang.VerifyError`. A `VerifyError` occurs when Android tries to load a class that, while being syntactically correct, refers to resources that are not available (*e.g.,* wrong class paths). In the remaining 25% of the cases, several of the crashes were due to the Inline Constant Replacement (ICR) operator. An example is the crash observed in the `photostream` app where the "100" value has been replaced with "101" in `bitmap.compress(Bitmap.CompressFormat.PNG, 100, out)`. Since "100" represents the

quality of the compression, its value must be bounded between 0 and 100.

PIT. In this tool, several of the manually analyzed crashes were due to (i) the RVR operator changing the return value of a method to null, causing a `NullPointerException`, and (ii) removed method invocations causing issues similar to the ones described for Major.

MDroid+. Table 3.2 lists the mutants generated by MDroid+ across all the systems (information for the other tools is provided with our replication package). The overall rate of SM is very low in MDroid+, and most failed compilations pertain to edge cases that would require a more robust static analysis approach to resolve. For example, the ClosingNullCursor operator has the highest total number of SM (across all the apps) with 13, and some edge cases that trigger compilation errors involve cursors that have been declared `Final`, thus causing the reassignment to trigger the compilation error. The small number of other SM are generally other edge cases, and current limitations of MDroid+ can be found in our replication package with detailed documentation.

The three operators generating the highest number of TM are NullIntent(41), FindViewByIdReturnsNull(40), and InvalidIDFindView(30). The main reason for the NullIntent TM are intents invoked by the Main Activity of an app (*i.e.,* the activity loaded when the app starts). Intents are one of the fundamental components of Android apps and function as asynchronous messengers that activate Activities, Broadcast Receivers and services. One example of a trivial mutant is for the A2dp.Vol app, in which a bluetooth service, inteneded to start up when the app is launched, causes a NullPointerException when opened due to NullIntent operator. To avoid cases like this, more sophisticated static analysis could be performed to prevent mutations from affecting Intents in an app's MainActivity. The story is similar for the FindViewViewByIdReturnsNull and InvalidIDFindView operators: TM will occur when views in the MainActivity of the app are set to null or reference invalid Ids, causing a crash on startup. Future improvements to the tool could avoid mutants to be seeded in components related to the MainActivity. Also, it would be desirable to allow developers to choose the activities in which mutations should be injected.

Summary of the RQs. MDroid+ outperformed the other six mutation tools by achieving the highest coverage both in terms of bug types and bug instances. However, the results show that Android-specific mutation operators should be combined with classic operators to generate mutants that are representative of real faults in mobile apps ($\mathbf{RQ}_1$). MDroid+ generated the smallest rate of both stillborn and trivial mutants illustrating its immediate applicability to Android apps. Major and muDroid generate stillborn mutants, with the latter having a critical average rate of 58.7% stillborn mutants per app ($\mathbf{RQ}_2$). All four tools generated a relatively low rate of trivial mutants, with muDroid again being the worst with an 11.8% average rate of trivial mutants ($\mathbf{RQ}_3$). Our analysis shows that the PIT tool is most applicable to Android apps when evaluated in terms of the ratio between failed and generated mutants. However, MDroid+ is both practical and based on Android-specific operations implemented according to an empirically derived fault-taxonomy of Android apps.

3.6 Threats to Validity

This section discusses the threats to validity of the work related to devising the fault taxonomy, and carrying out the study reported in Sec. 3.5.

Threats to *construct validity* concern the relationship between theory and observation. The main threat is related to how we assess and compare the performance of mutation tools, *i.e.,* by covering the types, and by their capability to limit stillborn and trivial mutants. A further, even more relevant evaluation would explore the extent to which different mutant taxonomies are able to support test case prioritization. However, this requires a more complex setting which we leave for our future work.

Threats to *internal validity* concern factors internal to our settings that could have influenced our results. This is, in particular, related to possible subjectiveness of mistakes

in the tagging of Sec. 3.3 and for $\mathbf{RQ}_1$. As explained, we employed multiple taggers to mitigate such a threat.

Threats to *external validity* concern the generalizability of our findings. To maximize the generalizability of the fault taxonomy, we have considered six different data sources. However, it is still possible that we could have missed some fault types available in sources we did not consider, or due to our sampling methodology. Also, we are aware that in our study results of $\mathbf{RQ}_1$ are based on the new sample of data sources, and results of $\mathbf{RQ}_2$ on the set of 68 apps considered [64].

3.7 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary and researchers from the Universidad de los Andes, the University of Sannio, and the the Università della Svizzera italiana:

Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., **Bernal-Cárdenas, C.**, and Poshyvanyk, D., "Enabling Mutation Testing for Android Apps", in Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the 25th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'17), Paderborn, Germany, September 4-8, 2017, pp. 233-244

Chapter 4

Translating Video Recordings of Mobile App Usages into Replayable Scenarios

Mobile application developers rely on a diverse set of software artifacts to help them make informed decisions throughout the development process. These information sources include user reviews, crash reports, bug reports, and emails, among others. An increasingly common component of these software artifacts is graphical information, such as screenshots or screen recordings. This is primarily due to the fact that they are relatively easy to collect and, due to the GUI-driven nature of mobile apps, they contain rich information that can easily demonstrate complex concepts, such as a bug or a feature request. In fact, many crowd-testing and bug reporting frameworks have built-in screen recording features to help developers collect mobile application usage data and faults [36, 11, 10, 35]. *Screen recordings* that depict application usages are used by developers to: (i) help understand how users interact with apps [183, 21]; (ii) process bug reports and feature requests from end-users [57]; and (iii) aid in bug comprehension for testing related tasks [142]. However, despite the growing prevalence of visual mobile development artifacts, developers must still manually inspect and interpret screenshots and videos in order to glean relevant informa-

tion, which can be time consuming and ambiguous. The manual effort required by this comprehension process complicates a development workflow that is already constrained by language dichotomies [149] and several challenges unique to mobile software, including: (i) pressure for frequent releases [97, 104], (ii) rapidly evolving platforms and APIs [123, 52], (iii) constant noisy feedback from users [66, 75, 171, 169, 170], and (iv) fragmentation in the mobile device ecosystem [93, 192, 4] among others [133]. Automation for processing graphical software artifacts is necessary and would help developers shift their focus toward core development tasks.

To improve and automate the analysis of video-related mobile development artifacts, we introduce Video to Scenario (V2S), a lightweight automated approach for translating video screen recordings of Android app usages into replayable scenarios. We designed V2S to operate solely on a video file recorded from an Android device, and as such, it is based primarily on computer vision techniques. V2S adapts recent Deep Learning (DL) models for object detection and image classification to accurately detect and classify different types of user actions performed on the screen. These classified actions are then translated into replayable scenarios that can automatically reproduce user interactions on a target device, making V2S the first purely graphical Android record-and-replay technique.

In addition to helping automatically process the graphical data that is already present in mobile development artifacts, V2S can also be used for improving or enhancing additional development tasks that do not currently take full advantage of screen-recordings, such as: creating and maintaining automated GUI-based test suites; and crowdsourcing functional and usability testing.

We conducted a comprehensive evaluation of V2S using both videos collected from users reproducing bugs as well as general usage videos from the top-rated apps of 32 categories in the Google Play market. As part of this evaluation, we examined the effectiveness of the different components that comprise V2S as well as the accuracy of the generated scenarios. Additionally, we assessed the overhead of our technique and conducted a case study with three industrial partners to understand the practical applicability of V2S. The results of

our evaluation indicate that V2S is *accurate*, and is able to correctly reproduce 89% of events across collected videos. The approach is also *robust* in that it is applicable to a wide range of popular native and non-native apps currently available on Google Play. In terms of *efficiency*, we found that V2S imposes acceptable overhead, and is perceived as potentially useful by developers.

In summary, the main contributions of our work are as follows:

- V2S, the first record-and-replay approach for Android that functions purely on screen-recordings of app usages. V2S adapts computer vision solutions for object detection, and image classification, to effectively recognize and classify user actions in the video frames of a screen recording;
- An automated pipeline for dataset generation and model training to identify user interactions from screen recordings;
- The results of an extensive empirical evaluation of V2S that measures the *accuracy*, *robustness*, and *efficiency* across 175 videos from 80 applications;
- The results of a case study with three industrial partners, who develop commercial apps, highlighting V2S's potential usefulness, as well as areas for improvement and extension;
- An online appendix [54], which contains examples of videos replayed by V2S, experimental data, source code, trained models, and our evaluation infrastructure to facilitate reproducibility of the approach and the results.

4.1 The V2S Approach

This section outlines the V2S approach for automatically translating Android screen recordings into replayable scenarios. Fig. 4.1 depicts V2S's architecture, which is divided into three main phases: (i) the *Touch Detection* phase, which identifies user touches in

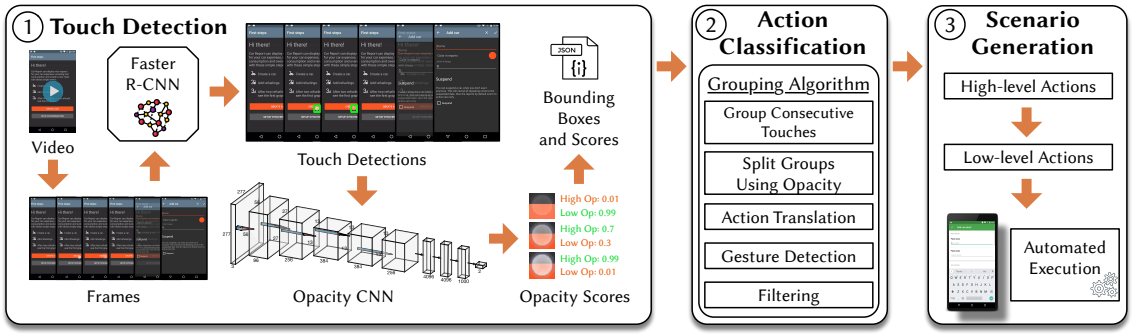


Figure 4.1: The V2S Approach and Components

each frame of an input video; (ii) the *Action Classification* phase that groups and classifies the detected touches into discrete user actions (*i.e.,* tap, long-tap, and swipe), and (iii) the *Scenario Generation* phase that exports and formats these actions into a replayable script. Before discussing each phase in detail, we discuss some preliminary aspects of our approach, input specifications, and requirements.

4.1.1 Input Video Specifications

In order for a video to be consumable by V2S, it must meet a few lightweight requirements to ensure proper functioning with our computer vision (CV) models. First, the video frame size must match the full-resolution screen size of the target Android device, in order to be compatible with a specified pre-trained object-detection network. This requirement is met by nearly every modern Android device that has shipped within the last few years. These videos can be recorded either by the built-in Android `screenrecord` utility, or via third-party applications [16]. The second requirement is that input videos must be recorded at least 30 "frames per second" (FPS), which again, is met or exceeded by a majority of modern Android devices. This requirement is due to the fact that the frame-rate directly corresponds to the accuracy with which "quick" gestures (*e.g.,* fast scrolling) can be physically resolved in constituent video frames. Finally, the videos must be recorded with the "Show Touches" option enabled on the device, which is accessed through an advanced settings menu [5], and is available by default on nearly all Android devices

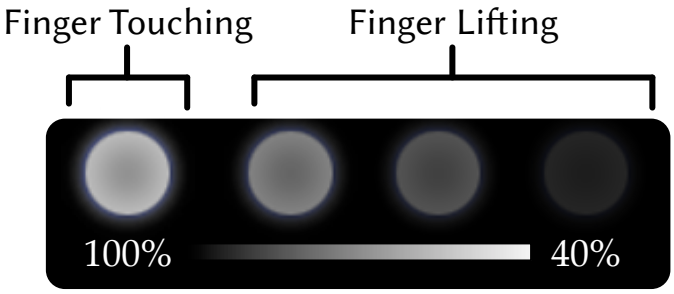


Figure 4.2: Illustration of touch indicator opacity levels

since at least Android 4.1. This option renders a *touch indicator*, which is a small semi-transparent circle, that gives a visual feedback when the user presses her finger on the device screen. The opacity of the indicator is fully solid from the moment the user first touches the screen and then fades from more to less opaque when a finger is lifted off the screen (Fig. 4.2).

4.1.2 Phase 1: Touch Detection

The *goal* of this phase is to accurately identify the locations where a user touched the device screen during a video recording. To accomplish this, V2S leverages the DL techniques outlined in Sec. 4.2 to both accurately find the position of the touch indicator appearing in video frames, and identify its opacity to determine whether a user's finger is being pressed or lifted from the screen. More specifically, we adapt an implementation of Faster R-CNN [176, 177], which makes use of VGGNet [186] for feature extraction of RPs in order to perform touch indicator detection. To differentiate between low and high-opacity detected touch indicators, we build an Opacity CNN, which is a modified version of AlexNet [117]. Given that we adapt well-known DL architectures, here we focus on describing our adaptions, and provide model specs in our appendix [54].

The *Touch Detection* Phase begins by accepting as input a video that meets the specifications outlined in Sec. 4.1.1. First, the video is *parsed* and decomposed into its constituent frames. Then the Faster R-CNN network is utilized to *detect* the presence of the touch indicator, if any, in every frame. Finally, the Opacity CNN *classifies* each detected touch

indicator as having either low or high-opacity. The output of this phase is a structured `JSON` with a set of touch indicator bounding boxes in individual video frames wherein each detected touch indicator is classified based on the opacity.

4.1.2.1 Parsing Videos

Before V2S parses the video to extract single frames, it must first normalize the frame-rate for those videos where it may be variable, to ensure a constant FPS. Certain Android devices may record variable frame-rate video for efficiency [32]. This may lead to inconsistencies in the time between frames, which V2S utilizes in the classification phase to synthesize the timing of touch actions. Thus, to avoid this issue, we normalize the frame rate to 30fps and extract individual frames using the `FFmpeg` [14] tool.

4.1.2.2 Faster R-CNN

After the individual frames have been parsed from the input video, V2S applies its object detection network to localize the bounding boxes of touch indicators. However, before using the object detection, it must be trained. As described in Sec. 4.2 the DL models that we utilize typically require large, manually labeled datasets to be effective. However, to avoid the manual curation of data, and make the V2S approach practical, we designed a fully automated dataset generation and training process. To bootstrap the generation of V2S's object detection training dataset, we make use of the existing large-scale ReDraw dataset of Android screenshots [156]. This dataset includes over 14k screens extracted from the most popular Android applications on Google Play using a fully-automated execution technique.

Next, we randomly sample 5k unique screenshots of different apps and programmatically superimpose an image of the *touch indicator* at a random location in each screenshot. During this process, we took two steps to ensure that our synthesized dataset reflects actual usage of the touch indicator: (i) we varied the opacity of the indicator icon between 40%-100% to ensure our model is trained to detect instances where a finger is lifted off

the screen; (ii) we placed indicator icons on the edges of the screen to capture instances where the indicator may be occluded. This process is repeated three times per screenshot to generate 15k unique images. We then split this dataset 70%/30% to create training and testing sets respectively. We performed this partitioning such that all screenshots expect one appear only in the testing set, wherein the one screenshot that overlapped had a different location and opacity value for the touch indicator. During testing, we found that a training set of 15k screens was large enough to train the model to extremely high levels of accuracy (*i.e.,* $> 97\%$). To train the model we use the *TensorFlow Object Detection API* [33] that provides functions and configurations of well-known DL architectures. We provide details regarding our training process for V2S's object detection network in Sec. 4.3.1. Note that, despite the training procedure being completely automated, it needs to be run only once for a given device screen size, after which it can be re-used for inference. After the model is trained, inference is run on each frame, resulting in a set of output *bounding box* predictions for each screen, with a confidence score.

4.1.2.3 Opacity CNN

Once V2S has localized the screen touches that exist in each video frame, it must then determine the opacity of each detected touch indicator to aid in the *Action Classification* phase. This will help V2S in identifying instances where there are multiple actions in consecutive frames with very similar locations (*e.g.,* double tapping). To differentiate between low and high-opacity touch indicators, V2S adopts a modified version of the AlexNet [117] architecture as an Opacity CNN that predicts whether a cropped image of the touch indicator is fully opaque (*i.e.,* finger touching screen) or low opacity (*i.e.,* indicating a finger being lifted off the screen). Similar to the object detection network, we fully automate the generation of the training dataset and training process for practicality. We again make use of the ReDraw dataset and randomly select 10k unique screenshots, randomly crop a region of the screenshot to the size of a touch indicator, and an equal number of full and partial opacity examples are generated. For the low-opacity examples, we varied the

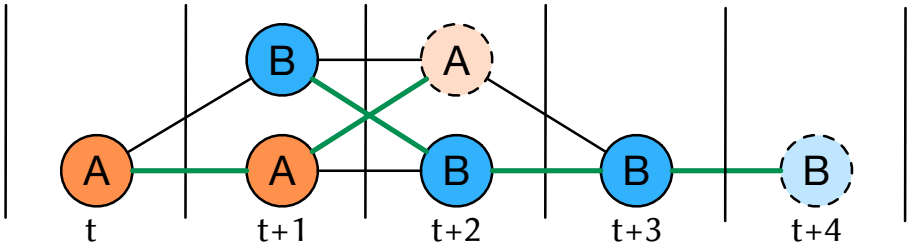


Figure 4.3: Illustration of the graph traversal problem for splitting discrete actions. Faded nodes with dotted lines represent touches where a finger is being lifted off the screen.

transparency levels between 20%-80% to increase the diversity of samples in the training set. During initial experiments, we found that our OPACITY CNN required fewer training samples than the object detection network to achieve a high accuracy (*i.e.,* $> 97\%$). Similar to the FASTER R-CNN model, this is a one-time training process, however, this model can be re-used across varying screen dimensions. Finally, V2S runs the classification for all the detected touch indicators found in the previous step. Then V2S generates a `JSON` file containing all the detected bounding boxes, confidence levels, and opacity classifications.

4.1.3 Phase 2: Action Classification

The `JSON` file generated by the *Touch Detection* phase contains detailed data about the bounding boxes, opacity information, and the frame of each detected touch indicator (note we use the term "touch indicator" and "touch" interchangeably moving forward). This `JSON` file is used as input into the *Action Classification* phase where single touches are grouped and classified as high-level actions. The classification of these actions involves two main parts: (i) a *grouping algorithm* that associates touches across subsequent frames as a discrete action; and (ii) *action translation*, which identifies the grouped touches as a single action type. The output of this step is a series of touch groups, each corresponding to an action type: (i) `Tap`, (ii) `Long Tap`, or (iii) `Gesture` (*e.g.,* swipes, pinches, etc).

4.1.3.1 Action Grouping

The first step of V2S's *action grouping* filters out detected touches where the model's confidence is lower than 0.7. The second step groups touches belonging to the same atomic action according to a tailored heuristic and a graph connection algorithm. This procedure is necessary because discrete actions performed on the screen will persist across several frames, and thus, need to be grouped and segmented accordingly.

Grouping Consecutive Touches. The first heuristic groups touch indicators present in consecutive frames into the same group. As a measure taken to avoid (the rare occurrence of) a falsely detected touch indicator, touches that exist across two or fewer frames are discarded. This is due to the fact that, we observed in practice, even the quickest of touchscreen taps last across at least five frames.

Discrete Action Segmentation. There may exist successive touches that were carried out extremely fast, such that there is no empty frame between two touches. In other cases, the touch indicator of one action may not fully disappear before the user starts a new action, leading to two or more touch indicators appearing in the same frame. These two situations are common when a user is swiping a list and quickly tapping an option, or typing quickly on the keyboard, respectively. However, it can be hard to determine where one action ends and another begins.

V2S analyzes groups of consecutive or overlapping touches and segments them into discrete actions using a heuristic-based approach. We model the grouping of touch indicators as a graph connectivity problem (see Fig. 4.3). In this formulation, touch indicators are represented as nodes, vertical lines separate consecutive frames, and edges are possible connections between consecutive touches that make up a discrete action. The goal is to derive the proper edges for traversing the graph such all touches for action A are in one group and all touches for action B are in another group (illustrated in green in Fig. 4.3). Our algorithm decomposes the lists of consecutive touches grouped together into a graph. Starting from the first node, our algorithm visits each subsequent node and attempts to

link it to the previous node. If there is only one node in a subsequent frame, then two successive nodes are linked. If there is more than one node in a subsequent frame, our algorithm looks at the spatial distance between the previous node and both subsequent nodes, and groups previous nodes to their closer neighbors (as shown between frame t and $t+1$). However, if multiple nodes in one frame are at similar distance from the previous node (as between frames $t+1$ and $t+2$ in Fig. 4.3), then the opacity of the nodes is used to perform the connection. For example, it is clear that node A in frame $t+2$ is a finger being lifted off the screen. Thus, it must be connected to the previously occurring action A (*i.e.,* in $t+1$), and not the action B that just started.

Finally, after this process, the opacity of all linked nodes are analyzed to determine further splits. There exist multiple actions with no empty frame between them. Therefore, if a low-opacity node is detected in a sequence of successively connected nodes, they are split into distinct groups representing different actions.

4.1.3.2 Action Translation

This process analyzes the derived groups of touches and classifies them based on the number and touch locations in each group. For touches that start and end in the same spatial location on the screen (*e.g.,* the center of subsequent touch indicator bounding boxes varies by less than 20 pixels) the action is classified as a `Tap` or a `Long Tap`. `Taps` are recognized if the action lasts across 20 or fewer frames, and `Long-Taps` otherwise. Everything else is classified as a `Gesture`.

4.1.3.3 Filtering

V2S removes actions that have touch indicators with low average opacity (*e.g.,* $< 0.1\%$) across a group, as this could represent a rare *series* of misclassified touch indicators from the Faster R-CNN. V2S also removes groups whose size is below or equals a threshold of two frames, as these might also indicate rare misclassifications. The result of the *Action*

Classification phase is a structured list of actions (*i.e.,* `Tap`, `Long Tap`, or `Gesture`), where each action is a series of touches associated to video frames and screen locations.

4.1.4 Phase 3: Scenario Generation

After all the actions have been derived by the *Action Classification* phase, V2S proceeds by generating commands using the Android Debug Bridge (`adb`) that replay the classified actions on a device. To accomplish this, V2S converts the classified, high-level actions into low-level instructions in the `sendevent` command format, which is a utility included in Android's Linux kernel. Then, V2S uses a modified RERAN [82] binary to replay the events on a device.

Generating the Scenario Script. The `sendevent` command uses a very limited instruction set in order to control the UI of an Android device. The main instructions of interest are the `start_event`, `end_event`, x and y coordinates where the user's finger touched the screen, and certain special instructions required by devices with older API levels. To create the script, each action is exported starting with the `start_event` command. Then, for actions classified as a `Tap`, V2S provides a single (x, y) coordinate pair, derived from the center of the detected bounding box of the `Tap`. For `Gestures`, V2S iterates over each touch that makes up the `Gesture` action and appends the (x, y) pairs of each touch indicator to the list of instructions. For `Long Taps`, V2S performs similar processing to that of `Gestures`, but instead uses only a single (x, y) pair from the initial detected touch indicator bounding box.

Then, V2S ends the set of instructions for an action with the appropriate `end_event` command. For `Gestures` and `Long Taps` the speed and duration of each instruction is extremely important in order to accurately replay the user's actions. To derive the speed and duration of these actions, V2S adds timestamps to each (x, y) touch location based on the timing between video frames (*i.e.,* for 30fps, there is a 33 millisecond delay between each frame), which will temporally separate each touch command sent to the device. The same concept applies for `Long Tap`, however, since this action type uses a single (x, y)

touch location, the timing affects the duration that the touch event lasts on the screen.

Finally, in order to determine the delays between successive actions, the timing between video frames is again used. Our required 30fps frame-rate provides V2S with millisecond-level resolution of event timings, whereas higher frame-rates will only increase the fidelity of replay timing.

Scenario Replay. Once all the actions have been converted into low-level `sendevent` instructions, they are written to a log file. This log file is then fed into a translator which converts the file into a runnable format that can be directly replayed on a device. This converted file along with a modified version of the RERAN engine [82] is pushed to the target device. We optimized the original RERAN binary to replay event traces more efficiently. Finally, the binary is executed using the converted file to faithfully replay the user actions originally recorded in the initial input video. We provide examples of V2S's generated `sendevent` scripts, alongside our updated version of the RERAN binary in our online appendix [54].

4.2 Background

We briefly discuss DL techniques for image classification and object detection that we adapt for touch/gesture recognition in V2S. Additionally, we briefly discuss how GUI-based events (*i.e.,* touchscreen interactions) from users are collected on the wild throughout record an replay tools to generate replayable traces.

4.2.1 Image Classification

Recently, DL techniques that make use of neural networks consisting of specialized layers have shown great promise in classifying diverse sets of images into specified categories. The increasingly high accuracy of these techniques has been largely driven by three main factors: (i) the curation of large-scale, manually annotated datasets such as ImageNet [180] and MSCOCO [122], (ii) rapid advancement in parallel computing power and GPUs, and (iii)

advancements in neural architectures using *convolutional layers*, which allow for automated learning of abstract hierarchical features [119]. One of the first end-to-end approaches to leverage for highly precise image recognition was AlexNet [117]. The success of AlexNet spurred a series of follow up work on more complex architectures [186, 198, 187, 95] that have reached human levels of accuracy for image classification tasks.

Typically, each *layer* in a CNN performs some form of computational transformation to the data fed into the model. The initial layer usually receives an input image. This layer is typically followed by a *convolutional layer* that extracts features from the pixels of the image, by applying *filters* (*a.k.a.* kernels) of a predefined size, wherein the contents of the filter are transformed via a pair-wise matrix multiplication (*i.e.,* the convolution operation). Each filter is passed throughout the entire image using a fixed *stride* as sliding window to extract *feature maps*. *Convolutional layers* are used in conjunction with "max pooling" layers to further reduce the dimensionality of the data passing through the network. The convolution operation is linear in nature. Since images are generally non-linear data sources, activation functions such as Rectified Linear Units (ReLUs) are typically used to introduce a degree of non-linearity. Finally, a fully-connected layer (or series of these layers) are used in conjunction with a *Softmax classifier* to predict an image class. The training process for CNNs is usually done by updating the weights that connect the layers of the network using gradient descent and back-propagating error gradients.

V2S implements a customized CNN for the specific task of classifying the opacity of an image to help segment GUI-interactions represented by a touch indicator (see Section 4.1.2).

4.2.2 Object Detection

In the task of image classification, a single, usually more general label (*e.g., bird* or *person*) is assigned to an entire image. However, images are typically multi-compositional, containing different objects to be identified. Similar to image classification, DL models for object detection have advanced dramatically in recent years, enabling object tracking

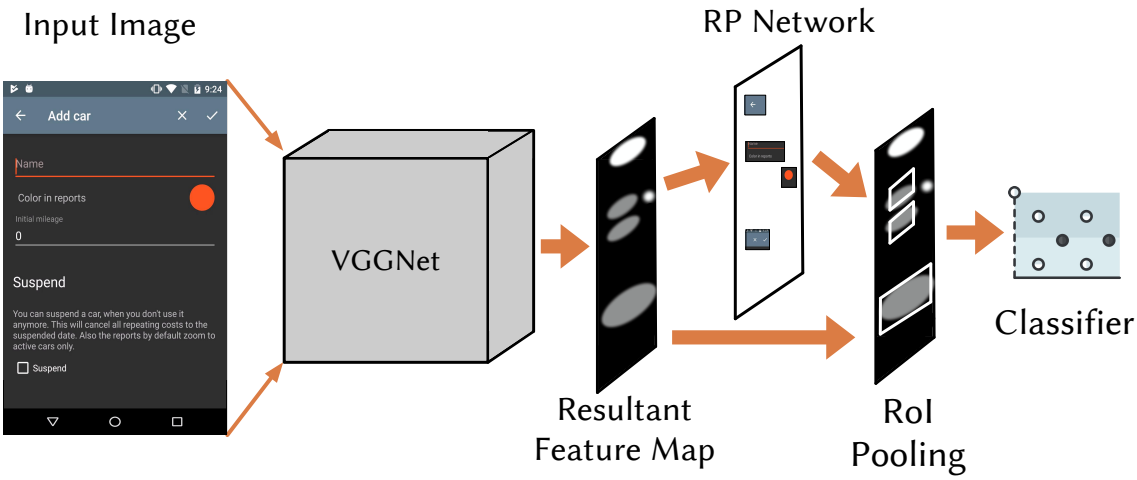


Figure 4.4: Illustration of the FASTER R-CNN Architecture

and counting, as well as face and pose detection among other applications. One of the most influential neural architectures that has enabled such advancements is the R-CNN introduced by Girshick *et al.* [80]. The R-CNN architecture combines algorithms for image region proposals (RPs), which aim to identify image regions where content of interest is likely to reside, with the classification ability of a CNN. An R-CNN generates a set of RP bounding-boxes using a selective search algorithm [190]. Then, all identified image regions are fed through a pre-trained ALEXNET [117] (*i.e.,* the *extractor*) to extract image features into vectors. These vectors are fed into a support vector machine (*i.e.,* the *classifier*) that determines whether or not each image region contains a class of interest. Finally, a greedy non-maximum suppression algorithm (*i.e.,* the *regressor*) is used to select the highest likelihood, non-overlapping regions, as classified objects.

Finally, it runs the bounding box through a linear regression to retrieve the coordinates (*regressor*). Unfortunately, R-CNN is very slow due to the fact that it must run ALEXNET through 2000 images taken from the RPs which presents a bottle neck for the approach.

Two notable version of the R-CNN architecture have been devised since the initial architecture's introduction. The FAST R-CNN [81] partially improved the training and inference speed of its predecessor by combining the three R-CNN steps outlined above into a single model. In this new architecture, only the initial image, as opposed to each

RP, runs through a single CNN to produce a feature map. Fast R-CNN then classifies each RP using *Region of Interest* (RoI) pooling to extract a fixed-length RoI *feature vector*. From this vector it generates two outputs: (i) a class using a fully connected layer and a softmax, and (ii) a linear regression layer to refine the *bounding box* coordinates.

In V2S, we utilize the Faster R-CNN [176] architecture (Fig. 4.4), which improves upon R-CNN architecture through the introduction of a separate NN to predict image region proposals. By integrating the training of the region proposal network into the end-to-end training of the network, both the speed and accuracy of the model increase. In V2S, we adapt the Faster R-CNN model to detect a touch indicator representing a user action in video frames.

4.2.3 GUI-events for Android

Testing frameworks for Android provide an API to send and execute actions to devices. These frameworks execute wrapped high-level events that hide the complexity of producing low-level events generating an issue while trying to faithfully reproduce a trace in some cases. Record and replay tools like RERAN [82] use low-level events to accurately reproduce a trace recorded by users. This tool uses the `getevent` Linux command to record a trace and a modified version of `sendevent` to send multiple events at the same time. Using low-level events allows for the reproduction to be very accurate by considering every single location, tap pressure, and timestamp between the moment a finger touches the screen and when it is released.

4.3 Design of the Experiments

In this section, we describe the procedure we used to evaluate V2S. The goal of our empirical study is to assess the *accuracy*, *robustness*, *performance*, and *industrial utility* of the approach. The *context* of this evaluation consists of: (i) sets of 15,000 and 10,000 images, corresponding to the evaluation of V2S's Faster R-CNN and Opacity CNN

respectively; (ii) a set of 83 Android applications including 68 of the top-rated apps from Google Play, five open source apps with real crashes, five open source apps with known bugs, and five open source apps with controlled crashes; (iii) two popular target Android devices (the Nexus 5 and Nexus 6P)[1]. The main *quality focus* of our study is the extent to which V2S can generate replayable scenarios that mimic original user GUI inputs. To achieve our study goals, we formulated the following five research questions:

- ***RQ***$_1$*: How accurate is* V2S *in identifying the location of the touch indicator?*
- ***RQ***$_2$*: How accurate is* V2S *in identifying the opacity of the touch indicator?*
- ***RQ***$_3$*: How effective is* V2S *in generating a sequence of events that accurately mimics the user behavior from video recordings of different applications?*
- ***RQ***$_4$*: What is* V2S'S *overhead in terms of scenario generation?*
- ***RQ***$_5$*: Do practitioners perceive* V2S *as useful?*

4.3.1 RQ$_1$: Accuracy of Faster R-CNN

To answer RQ$_1$, we first evaluated the ability of V2S'S FASTER R-CNN to accurately identify and localize the *touch indicators* present in screen recording frames with *bounding boxes*. To accomplish this, we followed the procedure to generate training data outlined in Sec. 4.1.2 complete with the 70%–30% split for the training and testing sets, respectively. The implementation of the FASTER R-CNN object detection used by V2S is coupled to the size of the images used for training and inference. Thus, to ensure V2S'S model functions across different devices, we trained two separate FASTER R-CNN models (*i.e.,* one for the Nexus 5 and one for the Nexus 6P), by resizing the images from the REDRAW dataset to the target device image size. As we show in the course of answering other RQs, we found that resizing the images in the already large REDRAW dataset, as opposed to re-collecting natively sized images for each device, resulted in highly accurate detections in practice.

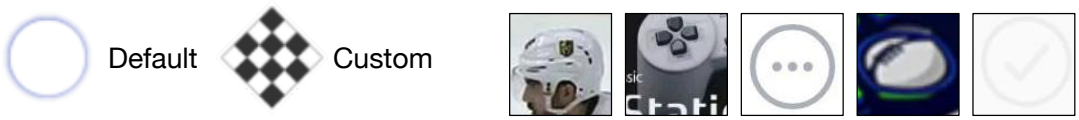
[1]It should be noted that V2S can be used with emulators via minor modifications to the script generation process

We used the *TensorFlow Object Detection API* [33] to train our model. Moreover, for the training process, we modified several of the hyper-parameters after conducting an initial set of experiments. These changes affected the number of classes (*i.e.,* 1), maximum number of detections per class or image (*i.e.,* 10), and the learning rate after 50k (*i.e.,* 3×10^{-5}) and 100k (*i.e.,* 3×10^{-6}) iterations. The training process was run for 150k steps with a batch size of 1, and our implementation of Faster R-CNN utilized a VGGNet [186] instance pre-trained on the MSCOCO dataset [122]. We provide our full set of model parameters in our online appendix [54].

To validate the accuracy of V2S's Faster R-CNN models we utilize *Mean Average Precision* (mAP) which is commonly used to evaluate techniques for the object detection task. This metric is typically computed by averaging the *precision* over all the categories in the data, however, given we have a single class (the touch indicator icon), we report results only for this class. Thus, our mAP is computed as $mAP = TP/(TP + FP)$ where TP corresponds to an identified image region with a correct corresponding label, and FP corresponds to the identified image regions with the incorrect label (which in our case would be an image region that is falsely identified as a touch indicator). Additionally, we evaluate the *Average Recall* of our model in order to determine if our model misses detecting any instances of the touch indicator. This is computed by $AR = TP/k$ where TP is the same definition stated above, and the k corresponds to the total number of possible TP predictions.

During preliminary experiments with Faster R-CNN using the default *touch indicator* (see Fig. 4.5a), we found that, due to the default touch indicator's likeness to other icons and images present in apps, it was prone to very occasional false positive detections (Fig. 4.5b). Thus, we analyzed particular cases in which the default touch indicator failed and replaced it with a more distinct, high-contrast touch indicator. We found that this custom touch indicator marginally improved the accuracy of our models. It should be noted that replacing the touch indicator on a device, requires the device to be rooted. While this is an acceptable option for most developers, it may prove difficult for end-users. How-

ever, even with the default touch indicator, V2S's Faster R-CNN model still achieves extremely high levels of accuracy.



(a) Touch indicators (b) False positive detections

Figure 4.5: Touch indicators and failed detections

4.3.2 RQ_2: Accuracy of Opacity CNN

To answer RQ_2, we evaluated the ability of V2S's Opacity CNN to predict whether the opacity of the touch indicator is solid or semi-transparent. To accomplish this, we followed dataset generation procedure outlined in Sec. 4.1.2, where equal number of full and partial opacity examples are generated. Thus, the generated dataset contains equal numbers of full and partial opacity examples for a total of 10k, which are evenly split into 70%–30% training and testing sets. We used the TensorFlow framework in combination with Keras to implement the Opacity CNN. In contrast to the Faster R-CNN model used previously, we do not need to create a separate model for each device. This is due to the *touch indicator* being resized when fed into the Opacity CNN. Similarly to the Faster R-CNN, we evaluate Opacity CNN using mAP across our two classes.

4.3.3 RQ_3: Accuracy on Different Scenarios

To answer RQ_3, we carried out two studies designed to assess both the *depth*, and *breadth* of V2S's abilities to reproduce user events depicted in screen recordings. The first, *Controlled Study*, measures the depth of V2S's abilities through a user study during which we collected real videos from end users depicting: bugs, real crashes, synthetically injected crashes, and normal usage scenarios for 20 apps. Next in the *Popular Applications Study* we measured the breadth of V2S's abilities by recording scenarios for a larger, more diverse set of 64

most popular apps from the Google Play. We provide the full details of these apps in our online appendix [54].

4.3.3.1 Controlled Study

In this study we, considered four types of recorded usage scenarios depicting: (i) normal usages, (ii) bugs, (iii) real crashes, and (iv) controlled crashes. Normal usage scenarios refer to video recordings exercising different features on popular apps. Bug scenarios refer to video recordings that exhibit a bug on open source apps. Finally, controlled crashes refer to injected crashes into open source apps. This allows us to control the number of steps before the crash is triggered.

For this study, eight participants including 1 undergraduate, 3 masters, and 4 doctoral students were recruited from William & Mary (approved by the Protection of Human Subjects Committee (PHSC) at W&M under protocol PHSC-2019-01-22-13374) to record the videos, with each participant recording eight separate videos, two from each of the categories listed above. Four participants recorded videos on the Nexus 5 and four used the Nexus 6P. This accounts for a total of 64 videos, from 20 apps evenly distributed across all scenarios. Before recording each app, participants were either asked to use the app to become familiar with it, or read a bug/crash report before reproducing the fault. All of the real bugs and crashes were taken from established past studies on mobile testing and bug reporting [157, 155, 59].

4.3.3.2 Popular Applications Study

For the next study, we considered a larger more diverse set of apps from Google Play. More specifically we downloaded the two highest-rated apps from each non-game category (*i.e.,* 32) for a total of 64 applications.

Two of the authors then recorded two scenarios per app accounting for 32 apps each, one using the Nexus 5 and the other using a Nexus 6P. The authors strived to use the apps as naturally as possible, and this evaluation procedure is in line with past evaluations of

Android record-and-replay techniques [174, 82]. The recorded scenarios represented specific use cases of the apps that exercise at least one of the major features, and were independent of one another. During our experiments, we noticed certain instances where our recorded scenarios were not replicable, either due to non-determinism or dynamic content (*e.g.,* random popup appearing). Thus, we discarded these instances and were left with 111 app usage scenarios from 60 apps. It is worth noting that it would be nearly impossible for existing techniques such as RERAN [82] or Barista [77] to reproduce the scenarios due to the nondeterminism of the dynamic content, hence our decision to exclude them.

To measure how accurately V2S replays videos, we use three different metrics. To compute these metrics, we manually derived the ground truth sequence of action types for each recorded video. First, we use Levenshtein distance, which is commonly used to compute distances between words at character level, to compare the original list of action types to the list of classified actions generated by V2S. Thus, we consider each type of action being represented as a character, and scenarios as sequences of characters which represent series of actions. A low Levenshtein distance value indicates fewer required changes to transform V2S's output to the ground truth set of actions. Additionally, we compute the longest common subsequence (LCS) to find the largest sequence of each scenario from V2S's output that aligns with the ground truth scenario from the video recording. For this LCS measure, the higher the percentage, the closer V2S's trace is to a perfect match of the original trace. Moreover, we also computed the precision and recall for V2S to predict each type of action across all scenarios when compared to the ground truth. Finally, in order to validate the fidelity of the replayed scenarios generated by V2S compared to the original video recording, we manually compared each original video to each reproduced scenario from V2S, and determined the number of actions for each video that were faithfully replayed.

4.3.4 RQ_4: Performance

To investigate RQ_4, we evaluated V2S by calculating the average time it takes for a video to pass through each of the three phases of the V2S approach on commodity hardware (*i.e.,* a single NVIDIA GTX 1080Ti). We see this as a worst case scenario for V2S performance, as our approach could perform substantially faster on specialized hardware. Note that since our replay engine is an enhancement of the RERAN engine, we expect our scripts to have similar or better overhead as reported in its respective paper [82].

4.3.5 RQ_5: Perceived Usefulness

Ultimately, our goal is to integrate V2S into real-world development environments. Thus, as part of our evaluation, we investigated V2S's perceived usefulness with three developers who build Android apps (or web apps for mobile) for their respective companies.

The developers (*a.k.a.* participants) were contacted through direct contact of the authors. Participant #1 (P1) was a front-end developer on the image search team of the Google Search app [18], participant #2 (P2) is a developer of the 7-Eleven Android app [1], and participant #3 (P3) is a backend developer for the Proximus shopping basket app [25]. We interviewed the participants using a set of questions organized in two sections. The first section aimed to collect information on participants' background, including their role at the company, the information used to complete their tasks, the quality of this information, the challenges of collecting it, and how they use videos in their every-day activities. The second section aimed to assess V2S's potential usefulness as well as its accuracy in generating replayable scenarios. This section also asked the participants for feedback to improve V2S including likert scale questions. We provide the complete list of interview questions used in our online appendix [54], and discuss selected questions in Sec. 4.4.5.

The participants answered questions from the second section by comparing two videos showing the same usage scenario for their respective app: one video displaying the scenario manually executed on the app, and the other one displaying the scenario executed

Table 4.1: Touch Indicator Detection Accuracy

Model	Device	mAP	mAP@.75	AR
Faster R-CNN-Original	Nexus 5	97.36%	99.01%	98.57%
Faster R-CNN-Original	Nexus 6P	96.94%	99.01%	98.19%
Faster R-CNN-Modified	Nexus 5	97.98%	99.01%	99.33%
Faster R-CNN-Modified	Nexus 6P	97.49%	99.01%	99.07%

automatically via V2S's generated script. Specifically, we defined, recorded, and manually executed a usage scenario on each app. Then, we ran V2S on the resulting video recordings. To define the scenarios, we identified a feature on each app involving any of the action types (*i.e.,* taps, long taps, and gestures). Then, we generated a video showing the original scenario (*i.e.,* video recording) and next to it the replayed scenario generated when executing V2S's script. Both recordings highlight the actions performed on the app. We presented the video to participants as well as V2S's script with the high-level actions automatically identified from the original video.

4.4 Empirical Results

4.4.1 RQ_1: Accuracy of Faster R-CNN

Table 4.1 depicts the precision and recall for V2S's Faster R-CNN network for touch indicator detection on different devices and datasets. The first column identifies the usage of either the default touch indicator or the modified version. The second column describes the target device for each trained model. The third column provides the mAP regardless of the Intersection Over Union (IoU) [176] between the area of the prediction and the ground truth. The forth column presents the AP giving the proportion of TP out of the possible positives. All models achieve ≈97% mAP, indicating that V2S's object detection network is highly accurate. The mAP only improves when we consider bounding box IoUs that match the ground truth bounding boxes by at least 75%, which illustrates that when the model is able to predict a reasonably accurate bounding box, it nearly always properly detects the touch indicator (≈99%). As illustrated by the last column in Table 4.1, the model also achieves extremely high recall, detecting at least ≈98% of the inserted touch

Table 4.2: Confusion Matrix for Opacity CNN. Low Opacity Original (L-Op.-Orig.), High Opacity Original (H-Op.-Orig.), Low Opacity Custom (L-Op.-Cust.), High Opacity Custom (H-Op.-Cust.)

	Total	L-Op.-Orig.	H-Op.-Orig.	L-Op.-Cust.	H-Op.-Cust.
Low Op	5000	97.8%	2.2%	99.7%	0.3%
High Op	5000	1.4%	98.6%	0.8%	99.2%

indicators.

> **Answer to RQ$_1$**: V2S benefits from the strong performance of its object detection technique to detect touch indicators. All FASTER R-CNN models achieved at least $\approx$ 97% precision and at least $\approx$ 98% recall across devices.

4.4.2 RQ$_2$: Accuracy of the OPACITY CNN

To illustrate the OPACITY NETWORK's accuracy in classifying the two opacity levels of touch indicators, we present the confusion matrix in Table 4.2. The results are presented for both the default and modified touch indicator. The overall top-1 precision for the original touch indicator is 98.2% whereas for the custom touch indicator is 99.4%. These percentages are computed by aggregating the correct identifications for both classes (i.e., Low/High-Opacity) *together* for the original and custom touch indicators. Hence, it is clear V2S'S OPACITY CNN is highly effective at distinguishing between differing opacity levels.

> **Answer to RQ$_2$**: V2S benefits from the CNNs accuracy in classifying levels of opacity. OPACITY CNN achieved an average precision above 98% for both touch indicators.

4.4.3 RQ$_3$: Scenario Replay Accuracy

Levenshtein Distance. Fig. 4.6a and 4.6b depict the number of changes required to transform the output event trace into the ground truth for the apps used in the *controlled study* and the *popular apps* study, respectively. For the controlled study apps, on average

Table 4.3: Detailed Results for RQ_3 popular applications study. Green cells indicate fully reproduced videos, orange cells >50% reproduced, and Red Cells <50% reproduced. Blue cells show non-reproduced videos due to non-determinism/dynamic content.

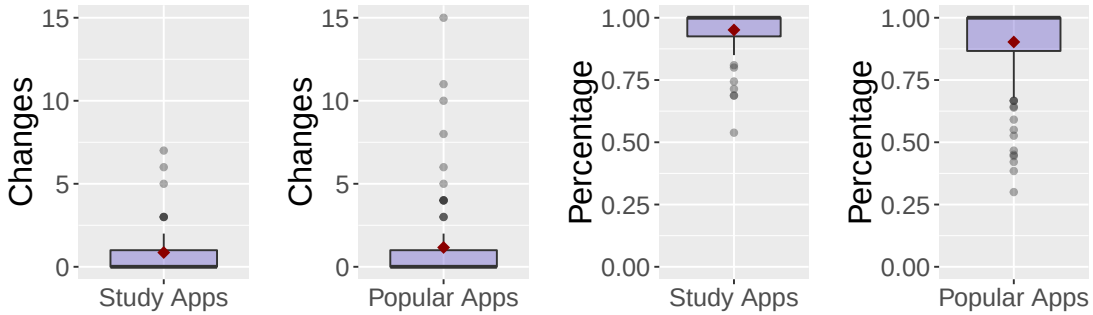
AppName	Actions		AppName	Actions		App Name	Actions		App Name	Actions	
Ibis Paint X	11/11	36/36	Firefox	22/22	N/A	Tasty	20/20	14/36	SoundCloud	12/12	13/13
Pixel Art Pro	20/20	9/9	MarcoPolo	12/12	30/30	Postmates	26/26	12/12	Shazam	12/15	20/20
Car-Part.com	20/20	16/16	Dig	13/13	N/A	Calm	9/9	11/16	Twitter	14/14	19/19
CDL Practice	8/8	13/13	Clover	15/15	19/19	Lose It!	36/36	N/A	News Break	1/18	9/9
Sephora	4/9	13/13	PlantSnap	39/39	18/24	U Remote	14/14	18/18	FamAlbum	19/19	8/26
SceneLook	14/14	16/16	Translator	20/20	28/28	LEGO	52/52	24/24	Baby-Track	14/14	12/12
KJ Bible	16/16	19/19	Tubi	2/19	30/30	Dev Libs	35/35	22/22	Walli	8/8	N/A
Bible App	12/12	15/15	Scan Radio	24/27	N/A	Horoscope	24/24	19/19	ZEDGE	9/9	18/18
Indeed Jobs	15/15	19/19	Tktmaster	30/30	14/14	Waze	17/17	19/19	G-Photo	18/18	18/18
UPS Mobile	16/16	19/24	Greet Cards	23/23	N/A	Transit	26/26	18/18	PicsArt	18/18	39/39
Webtoon	17/17	15/21	QuickBooks	47/47	28/28	WebMD	7/34	7/26	G-Docs	3/26	N/A
MangaToon	16/16	28/28	Yahoo Fin	23/23	N/A	K-Health	10/10	15/24	M. Outlook	27/27	21/26
LetGo	17/17	15/15	CBSSports	25/25	16/16	G-Podcast	9/9	15/15	DU Record	15/15	9/9
TikTok	14/14	11/11	MLBatBat	11/13	N/A	Airbnb	9/9	14/14	AccuWeath	13/13	21/21
LinkedIn	18/18	13/13	G-Translate	14/17	15/15	G-Earth	8/13	1/30	W. Radar	14/14	13/13

it requires 0.85 changes per user trace to transform V2S's output into ground truth event trace, whereas for the popular apps it requires slightly more with 1.17 changes. Overall, V2S requires minimal changes per event trace, being very similar to the ground truth.

Longest Common Subsequence. Fig. 4.6c and 4.6d presents the percentage of events for each trace that match those in the original recording trace for the *controlled study* and *popular apps* study respectively. On average V2S is able to correctly match 95.1% of sequential events on the ground truth for the controlled study apps and 90.2% for popular apps. These results suggest that V2S is able to generate sequences of actions that closely match the original trace in terms of action types.

Precision and Recall. Fig. 4.7 and 4.8 show the precision and recall results for the *controlled study* and *popular apps study*, respectively. These plots were constructed by creating an order agnostic "bag of actions" for each predicted action type, for each scenario in our datasets. Then the precision and recall are calculated by comparing the actions to a ground truth "bag of actions" to compute precision and recall metrics. Finally, an overall average precision and recall are calculated across all action types. The results indicate that on average, the precision of the event traces is 95.3% for the controlled study apps

and 95% for popular apps. This is also supported for each type of event showing also a high level of precision across types except for the precision on `Long Taps` for the popular apps. This is mainly due to the small number (*i.e.,* 9 `Long Taps`) of this event type across all the popular app traces. Also, Fig. 4.7 and 4.8 illustrate that the recall across action types is high with an average of 99.3% on controlled study apps and 97.8% on the popular apps for all types of events. In general, we conclude that V2S can accurately predict the correct number of event types across traces.



(a) LD-Study (b) LD-Popular (c) LCS-Study (d) LCS-Popular

Figure 4.6: Effectiveness Metrics

Success Rate. Finally, we also evaluated success rate of each replayed action for all scenarios across both RQ_3 studies. The 175 videos were analyzed manually and each action was marked as successful if the replayable scenario faithfully exercised the app features according to the original video. This means that in certain cases videos will not *exactly* match the original video recording (*e.g.,* due to a single keyboard keystroke error that still led to the same feature result). Thus, after validating all 64 videos for the controlled study, V2S fully reproduces 93.75% of the scenarios, and 94.48% of the consecutive actions. V2S fully reproduced 96.67% of the scenarios for bugs and crashes and 91.18% of apps usages. Detailed results for the *popular apps study* are shown in Table 4.3, where each app, scenario (with total number of actions), and successfully replayed actions are displayed. *Green* cells indicate a fully reproduced video, *Orange* cells indicate more than 50% of events reproduced, and Red cells indicate less than 50% of reproduced events. *Blue* cells show

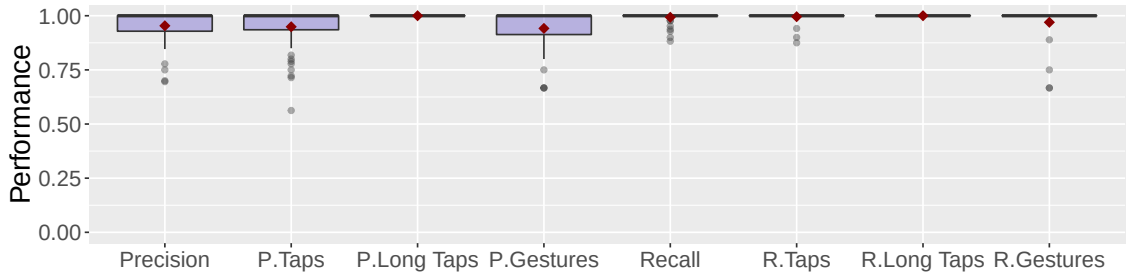


Figure 4.7: Precision and Recall - Controlled Study

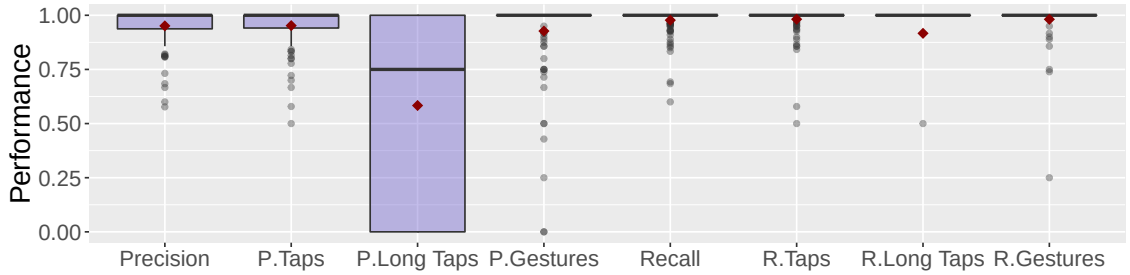


Figure 4.8: Precision and Recall - Popular Apps

non-reproduced videos due to non-determinism/dynamic content. For the 111 scenarios recorded for the popular apps, V2S fully reproduced 81.98% scenarios, and 89% of the consecutive actions. Overall, this signals strong replay-ability performance across a highly diverse set of applications. Instances where V2S failed to reproduce scenarios are largely due to minor inaccuracies in `Gesture` events due to our video resolution of 30fps. We discuss potential solutions to this limitation in Sec. 4.5.

> **Answer to RQ$_3$**: V2S is capable of generating event traces that require on average $\approx$ 1 change to match original user scenarios. Moreover, at least 90.2% of events match the ground truth, when considering the sequence of event types. Overall, precision and recall are $\approx$95% and $\approx$98% respectively for event types produced by V2S. Finally, in 96.67% and 91.18% of the cases, V2S successfully reproduces bugs/crashes- and app-usage-related videos, respectively.

4.4.4 RQ_4: Approach Performance

To measure the performance of V2S, we measured the average time in seconds/frame (s/f) for a single video frame to be processed across all recorded videos for three components: (i) the frame extraction process (0.045 s/f), (ii) the touch indicator detection process (1.09 s/f), and (iii) the opacity classification process (0.032 s/f). The script generation time is negligible compared to these other processing times, and is only performed once per video. This means that an average video around 3 mins in length would take V2S ≈105 minutes to fully process and generate the script. However, this process is *fully automated*, can run in the background, and can be accelerated by more advanced hardware. We expect the overhead of our replayed scripts to be similar or better than RERAN since V2S replay engine is essentially an improved version of RERAN's.

> **Answer to RQ_4**: V2S is capable of fully processing an average 3-min screen recording in ≈105 mins.

4.4.5 RQ_5: Perceived Usefulness

The three industry participants agreed that further tool support is needed for helping QA members and other stakeholders with generating video recordings. For example, P3 mentions that while videos are "more useful than images" (in some cases), they "may be difficult to record" because of "time constraints". All participants also (strongly) agreed that the scenarios produced by V2S (in the generated scripts) are accurate with respect to the scenarios that were manually executed.

Regarding V2S's usefulness, P1 remarked that the QA team could use V2S to help them create videos more optimally. P2 supported this claim as he mentions that V2S could help "the QA team write/provide commands or steps, then the tool would read and execute these while recording a video of the scenario and problem. This solution could be integrated in the continuous integration pipeline". In addition, P3 mentions that V2S could be used during app demos: V2S could "automatically execute a script that shows

the app functionality and record a video. In this way, the demo would focus on business explanation rather than on manually providing input to the app or execute a user scenario".

P2 also indicated that V2S could be used to analyze user behavior within the app, which can help improve certain app screens and navigation. He mentions that V2S "could collect the type of interactions, # of taps, *etc.* to detect, for example, if certain screens are frequently used or if users often go back after they go to a particular screen". He mentions that this data "could be useful for marketing purposes". P3 finds V2S potentially useful for helping reproduce hard-to-replicate bugs.

The participants provided valuable and specific feedback for improving V2S. They suggested to enrich the videos produced when executing V2S's script with a bounding box of the GUI components or screen areas being interacted with at each step. They also mention that the video could show (popup) comments that explain what is going on in the app (*e.g.,* a comment such as "after 10 seconds, button X throws an error"), which can help replicate bugs. They would like to see additional information in the script, such as GUI metadata that provides more in-depth and easy-to-read information about each step. For example, the script could use the names or IDs of the GUI components being interacted with and produce steps such as "the user tapped on the send button" instead of "the user tapped at (10.111,34.56)". P3 mentioned that "it would be nice to change the script programmatically by using the GUI components' metadata instead of coordinates, so the script is easier to maintain". They suggest to include an interpreter of commands/steps, written in a simple and easy-to-write language, which would be translated into low-level commands.

> Answer to $\mathbf{RQ}_5$: Developers find V2S accurate in replicating app usage scenarios from input videos, and potentially useful for supporting several development tasks, including automatically replicating bugs, analyzing usage app behavior, helping users/QA members generate high-quality videos, and automating scenario executions.

4.5 Limitations & Threats to Validity

Limitations. Our approach has various limitations that serve as motivation for future work. One current limitation of the Faster R-CNN implementation our approach utilizes is that it is tied to the screen size of a single device, and thus a separate model must be trained for each screen size to which V2S is applied. However, as described in Sec. 4.1.2, the training data process is fully automated and models can be trained once and used for any device with a given screen size. This limitation could be mitigated by increasing dataset size including all type of screen sizes with a trade-off on the training time. To facilitate the use of our model by the research community, we have released our trained models for the two popular screen sizes of the Nexus 5 and Nexus 6P in our online appendix [54].

Another limitation, which we will address in future work, is that our replayable traces are currently tied to the dimensions of a particular screen, and are not easily human readable. However, combining V2S with existing techniques for device-agnostic test case translation [77], and GUI analysis techniques for generating natural language reports [157] could mitigate these limitations.

Finally, as discussed in Sec. 4.4.3, one limitation that affects the ability of V2S to faithfully replay swipes is the video frame-rate. During the evaluation, our devices were limited to 30fps, which made it difficult to completely resolve a small subset of gesture actions that were performed very quickly. However, this limitation could be addressed by improved Android hardware or software capable of recording video at or above 60fps, which, in our experience, should be enough to resolve nearly all rapid user gestures.

Internal Validity. In our experiments evaluating V2S, threats to internal validity may arise from our manual validation of the correctness of replayed videos. To mitigate any potential subjectivity or errors, we had at least two authors manually verify the correctness of the replayed scenarios. Furthermore, we have released all of our experimental data and code [54], to facilitate the reproducibility of our experiments.

Construct Validity. The main threat to construct validity arises from the potential

bias in our manual creation of videos for the popular apps study carried out to answer RQ_3. It is possible that the author's knowledge of V2S influenced the manner in which we recorded videos. To help mitigate this threat, we took care to record videos as naturally as possible (*e.g.,* normal speed, included natural quick gestures). Furthermore, we carried out an orthogonal controlled study in the course of answering RQ_3, where users unfamiliar with V2S naturally recorded videos on a physical device, representing an unbiased set of videos. Another potential confounding factor concerns the quality of the dataset of screens used to train, test, and evaluate V2S's Faster R-CNN and Opacity CNN. To mitigate this threat, we utilize the ReDraw dataset [156] of screens which have undergone several filtering and quality control mechanisms to ensure a diverse set of real GUIs. One more potential threat concerns our methodology for assessing the utility of V2S. Our developer interviews only assess the *perceived* usefulness of our technique, determining whether developers actually receive benefit from V2S is left for future work.

External Validity. Threats to the generalizability of our conclusions are mainly related to: (i) the number and diversity apps used in our evaluation; (ii) the representativeness of usage scenarios depicted in our experimental videos; and (iii) the generalizability of the responses given by the interviewed developers. To help mitigate the first threat, we performed a large-scale study with 64 of the top applications on Google Play mined from 32 categories. While performing additional experiments with more applications is ideal, our experimental set of applications represents a reasonably large number of apps with different functionalities, which illustrate the relatively applicability of V2S. To mitigate the second threat, we collected scenarios illustrating bugs, natural apps usages, real crashes, and controlled crashes from eight participants. Finally, we do not claim that the feedback we received from developers generalizes broadly across industrial teams. However, the positive feedback and suggestions for future work we received in our interviews illustrate the potential practical usefulness of V2S.

4.6 Related Work

In this section we survey research that investigates: (i) approaches that use video screen recording/capture, (ii) record and replay approaches that collect low level event traces, and (iii) crowdsourcing input data collection approaches that leverages user data to generate test scenarios.

4.6.1 Analysis of Video and Screen Captures

Lin *et al.* [121] proposed an approach called Screenmilker to automatically extract screenshots of sensitive information (*e.g.,* user entering a password) by using the Android Debug Bridge (adb). This technique focuses on the extraction of keyboard inputs from"real-time" screenshots. Screenmilker is primarily focused upon extracting sensitive information, whereas V2S analyzes every single frame of a video to generate a high fidelity replay script from a sequence of video frames.

Krieter *et al.* [116] use video analysis to extract high-level descriptions of events from user video recordings on Android apps. Their approach generates log files that describe what events are happening at the app level. Compared to our work, this technique is not able to produce a script that would automatically replay the actions on a device, but instead simply describe high-level app events (*e.g., "WhatsApp chat list closed"*). On the other hand, our work focuses on video analysis to help with bug reproduction and generation of test scenarios, rather than describing usage scenarios at a high level.

Bao *et al.* [51] and Frisson *et al.* [79] focus on the extraction of user interactions to facilitate behavioral analysis of developers during programming tasks using CV techniques. In our work, rather than focusing on understanding user actions on mobile apps, we instead focus on generating high-fidelity replay scripts that closely match users' interactions from a screen recorded video.

Other researchers have proposed approaches that focus on the generation of source code for Android applications from screenshots or mock-ups. These approaches rely on

techniques that vary solely from *CV-based* [160] to *DL-based* [53, 156, 60].

The most related work to V2S is the AppFlow approach introduced by Hu *et al.* [98]. AppFlow leverages machine learning techniques to analyze Android screens and categorize types of test cases that could be performed on them (*i.e.,* a sign in screen whose test case would be a user attempting to sign in). However, this technique is focused on the generation of semantically meaningful test cases in conjunction with automated dynamic analysis. In contrast, V2S is focused upon the automated replication of any type of user interaction on an Android device, whether this depicts a usage scenario or bug. Thus, V2S could be applied to automatically reproduce crowdsourced mobile app videos, whereas AppFlow is primarily concerned with the generation of tests rather than the reproduction of existing scenarios.

4.6.2 Record and Replay

Many tools assist in recording and replaying tests for mobile platforms [82, 3, 12, 150]. However, many of these tools require the recording of low-level events using `adb`, which usually requires rooting of a device, or loading a custom operating system (OS) to capture user actions/events that are otherwise not available through standard tools such as `adb`. While our approach uses RERAN [82] to replay system-level events, we rely on video frames to transform touch overlays to low-level events. This facilitates bug reporting for users by minimizing the requirement of specialized programs to record and replay user scenarios.

Hu *et al.* [99] developed VALERA for replaying device actions, sensor and network inputs (*e.g.,* GPS, accelerometer, etc.), event schedules, and inter-app communication. This approach requires a rooted target device and the installation of a modified Android runtime environment. These requirements may lead to practical limitations, such as device configuration overhead and the potential security concerns of rooted devices. Such requirements are often undesirable for developers [118]. Conversely, our approach is able to work on any unmodified Android version, requiring just a screen recording.

Nurmuradov *et al.* [162] introduced a record and replay tool for Android applications

that captures user interactions by displaying the device screen in a web browser. This technique uses event data captured during the recording process to generate a *heatmap* that facilitate developers' understanding on how users are interacting with an application. This approach is limited in that users must interact with a virtual Android device through a web application, which could result in unnatural usage patterns. This technique is more focused towards session-based usability testing, whereas V2S is focused upon replaying "in-field" app interactions from users or crowdsourced testers collected from devices via screen recordings.

Other work has focused on capturing high-level interactions in order to replay events [91, 3, 143, 85]. For instance Mosaic [91], uses an intermediate representation of user interactions to make replays device agnostic. This intermediate representation is high level which uses a virtual screen to map coordinates into actions that can later be replayed. Our work is different due to the fact that for our replay engine we rely on RERAN, which uses low level events. This has the issue of not being directly device agnostic. However, replays are more accurate since exact events are being replayed instead of a high level representations. Additional tools including HiroMacro [19] and Barista [77] are Android applications that allow for a user to record and replay interactions. They require the installation or inclusion of underlying frameworks such as `replaykit` [12], or AirTest [3]. Android Bot Maker [29] is an Android application that allows for the automation of individual device actions, however, it does not allow for recording high-level interactions, instead one must enter manually the type of action and raw (x, y) coordinates. In contrast to these techniques, one of V2S's primary aims is to create an Android record and replay solution which an inherently low barrier to usage. For instance, there are no frameworks to install, or instrumentation to add, the only input is an easily collectable screen recording. This makes V2S suitable for use in crowd- or beta-testing scenarios, and improves the likelihood of its adoption among developers for automated testing, given its ease of use relative to developer's perceptions of other tools [130].

4.6.3 Crowdsourcing Input Data Collection

As crowdsourcing information from mobile app users has become more common with the advent of a number of testing services [8, 34, 22], researchers have turned to utilizing usage data recorded from crowd-testers to enhance techniques related to automated test case generation for mobile apps. Linares-Vásquez *et al.* first introduced the notion of recording crowd-sourced data to enhance input generation strategies for mobile testing through the introduction of the MonkeyLab framework [135]. This technique adapted N-gram language models trained on recorded user data to predict event sequences for automated mobile test case generation. Mao *et al.* developed the Polariz approach which is able to infer "motif" event sequences collected from the crowd that are applicable across different apps [142]. The authors found that the activity-based coverage of the Sapienz automated testing tool can be improved through a combination of *crowd-based* and *search-based* techniques. The two approaches discussed above require either instrumented or rooted devices (MonkeyLab), or interaction with apps via a web-browser (Polariz). Thus, V2S is complementary to these techniques as it provides a frictionless mechanism by which developers can collect user data in the form of videos.

4.7 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group at William & Mary and a researcher from the University of Texas at Dallas:

Bernal-Cárdenas, C., Cooper, N., Moran, K., Chaparro, O., Marcus, A., and Poshyvanyk, D., "Translating Video Recordings of Mobile App Usages into Replayable Scenarios", in Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE'20), Seoul, South Korea, May 23rd- 29th, 2020, pp. 309-321

Chapter 5

On-Device Bug Reporting for Android Applications

In order to aid in the difficult process of reproducing and fixing bugs related to mobile apps, in this paper we present a tool that enables developers to effectively carry out the process of **O**n-**D**evice **B**ug **R**eporting. Our prototype **ODBR** app is capable of running on a standalone physical or virtual Android device and recording precise user touch interactions coupled to specific GUI-components as well as sensor data streams for a target app. Then, this information is sent to a Java web application that displays a detailed, actionable bug report including screenshots, and series of user events that can be replayed on a target device allowing developers and testers to debug the app.

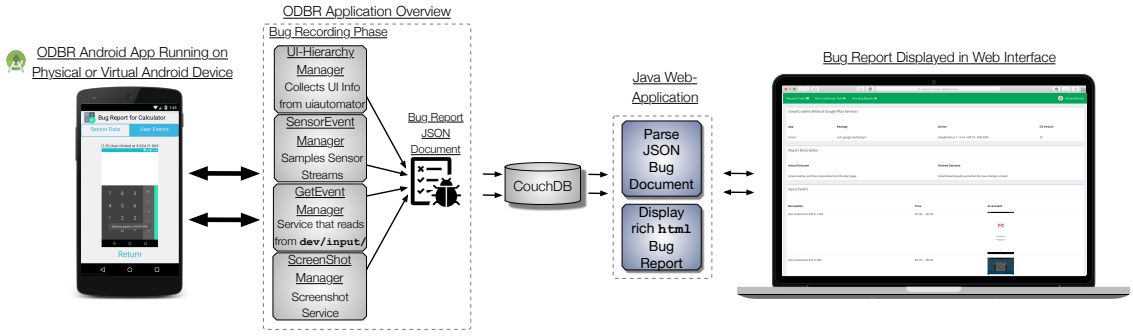


Figure 5.1: On-Device Bug Reporting Tool Architecture

5.1 Background & Related Work

Given that effective bug and error reporting is a problem faced by nearly all Android developers, there are several existing commercial and research-oriented tools geared toward improving the reporting process. Several commercial bug reporting and analytics platforms are available to support Android apps including Airbrake [2], TestFairy [35], Appsee [10], and Bug Clipper[11]. Typically, these solutions consist of an external library that a developer can include when writing their app. This library then enables the collection of information, such as device logs, performance information, screenshots and video recordings. To the best of the authors' knowledge, none of these services is capable of collecting fine-grained user input and GUI information capable of being replayed on a target device as **ODBR** does. This precise collection methodology could capture critical information that a developer needs to successfully reproduce and fix an incoming bug report.

Two recent automated input generation tools for mobile devices, CrashScope [157, 152] and Sapienz [141] are capable of producing detailed *crash reports*. CrashScope uses a combination of static and dynamic analysis to perform targeted testing of application features according several touch, text, and contextual feature input generation strategies and is capable of producing an `html` crash report with images, and a replayable script. Sapienz formulates the process of automated input generation as a multi-objective search problem, and is capable of producing videos and replayable scripts. While these automated tools are able to provide relatively robust *crash-reports* for apps they test, they are plagued by the oracle problem, in that while they can recognize and report when an app has crashed, they may miss user facing bugs that are more subtle. Additionally, FUSION [155, 152] is an off-device bug reporting system for Android applications that leverages static and dynamic analysis performed before the reporting process to help guide users through reporting detailed reproduction steps for an application. **ODBR** is complimentary to FUSION in that it provides another avenue for developers and testers to create detailed bug reports with minimal effort. Finally, previous Record & Replay approaches for Android apps have

been proposed including RERAN [82], VALERA [99], and MobiPlay [174]. While each of these solutions offers the ability to record user actions and replay them later, none of the tools offers the ability to record user actions on a standalone device without a connection to a host machine or server. **ODBR** leverages the highly efficient and accurate event stream recording introduced by RERAN and VALERA and combines this with detailed GUI-hierarchy information collected via the `uiautomator` tool to enable the capture of precise information to aid in the debugging process.

5.2 The ODBR Bug Reporting Tool

The ODBR Workflow: The overall architecture of our **ODBR** tool is illustrated in Figure 5.1. The entry point for a tester or developer using this tool is the *ODBR Android Application*, which is available as an open source project accessible from our tool's webpage[1] along with further information about the project and demo videos. To use our tool, a reporter simply needs to install the **ODBR** app on their target virtual or physical device[2] select the app for which they wish to report a bug from the list of installed applications, hit the record button and perform the actions on the device that manifest a bug. After the **ODBR** app (which is running in the background) detects that no touch-based inputs have been entered after a certain period of time, it asks the user if they have finished the report or if they wish to continue. Once the user has finished, they will have a chance to enter additional information about the bug, including a natural language description of the expected and actual behavior, and a title for the report. Additionally, the user can view the screenshots and sensor data traces from the device and even replay the touch events to ensure they were properly captured. This replay feature relies on a custom written Java implementation of the `sendevent` utility[3]. Once the report has been created, a `BugReport` is translated to a `JSON` document where it is sent over the web to a `CouchDB` server instance.

[1] `https://www.android-dev-tools.com/odbr`

[2] Our tool's app currently requires a rooted target device, a requirement for accessing the `/dev/input` event stream

[3] `https://goo.gl/EEUSfu`

A Java web-application then reads the bug report information from the `JSON` file and converts the information into a fully expressive html report. From the java web app, a developer can view the reproduction steps of the report, complete with screenshots and action descriptions, as well as download a replayable script that will reproduce the actions via `adb` or `sendevent` commands.

ODBR App components: The ODBR Application has 4 major components that aid in the collection of information during the bug reporting process. These include: (i) the ***GetEvent Manager***, (ii) the ***UI-Hierarchy Manager***, (iii) the ***Sensor Event Manager*** and (iv) the ***Screenshot Manager***. The ***GetEvent Manager*** is responsible for precisely and efficiently reading in the user event stream. To accomplish this, during the reporting process the app creates threads that read from the underlying Linux input event streams at `/dev/input/...` These input events provide highly detailed information about the user's interaction with the phone's physical devices, including (where applicable) the touch screen, power button, keyboard, etc. This information is identical to what is gathered using the Android `getevent` utility as used by RERAN [82]. Next, these low-level input event streams are parsed into higher level user interactions (e.g. *swipe from (a,b) to (c,d)*). The low-level input events are retained to support precise analysis and replayability, while the higher level interactions are used to summarize the report in natural language. Whenever the ***GetEvent Manager*** detects the user is taking a new action, it notifies the applicable managers to take a screenshot and dump of the UI-hierarchy, associating these with the new interaction. The ***UI-Hierarchy Manager*** interfaces with the Android `uiautomator` framework to capture dumps of the Android view hierarchy in `ui-dump.xml` files for each new user action. Because these dump files contain information about the screen location of each UI-component, we can use the event information obtained from the previous component to precisely infer the UI-component that the user interacted with at each step in the bug reporting process. This component also extracts attributes of the various components on the screen including information such as the type (e.g., button, spinner) and whether the component is clickable. The

Sensor Event Manager is responsible for efficiently sampling the sensor input streams (e.g., accelerometer, GPS) during the bug reporting process. This component accomplishes this by registering `SensorEventListener` instances for each sensor which sample the sensor values at appropriate rates. Finally, the ***Screenshot Manager*** is responsible for capturing an image of the screen for each new user interaction using the `screencap` utility included in Android.

5.3 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with other members of the SEMERU group and undergraduate students at William & Mary:

Moran, K., Bonett, R., **Bernal-Cárdenas, C.**, Otten, B., Park, D., and Poshyvanyk, D., "On-Device Bug Reporting for Android Applications", in Proceedings of 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'17), Formal Research Tool Demonstrations Track, Buenos Aires, Argentina, May 20-28, 2017, pp. 215-216

Chapter 6

Conclusion and Future Work

In this dissertation, we have presented a study and several approaches to support Android software developers and testers. Here, we summarize all conclusions and future work for each chapter.

6.1 Mutation Testing in Android

The survey in Chapter 2 helps us to take other directions to improve quality of test cases in Android app. Although these apps rely on the Java language as a programming platform, they have specific elements that make the testing process different than other Java apps. In particular, the type and distribution of faults exhibited by Android apps may be very peculiar, requiring, in the context of mutation analysis, specific operators.

In Chapter 3, we presented the first taxonomy of faults in Android apps, based on a manual analysis of 2,023 software artifacts from six different sources. The taxonomy is composed of 14 categories containing 262 types. Then, based on the taxonomy, we have defined a set of 38 Android-specific mutation operators, implemented in an infrastructure called MDroid+, to automatically seed mutations in Android apps. To validate the taxonomy and MDroid+, we conducted a comparative study with Java mutation tools. The study results show that MDroid+ operators are more representative of Android faults than other catalogs of mutation operators, including both Java and Android-specific

operators previously proposed. Also MDroid+ is able to outperform state-of-the-art tools in terms of stillborn and trivial mutants.

The obtained results make our taxonomy and MDroid+ ready to be used and possibly extended by other researchers and practitioners. To this aim, MDroid+ and the wrappers for using Major and Pit with Android apps are available as open source projects [127].

Future work could consider extending MDroid+ by implementing more operators, and extending the framework for mutation analysis of Android apps by considering other type of devices. For instance, mutation testing could be applied in the context of Internet of Things (IoT) devices that can potentially benefit users in daily activities during the day. It could facilitate tasks and provide better security for the final user with a well-tested environment of IoT devices. Also, future work could contemplate to experiment with MDroid+ in the context of test case prioritization.

6.2 Bug Reporting Reproducibility

In Chapter 4, we presented V2S, an approach for automatically translating video recordings of Android app usages into replayable scenarios. A comprehensive evaluation including 175 videos from over 80 apps indicates that V2S: (i) accurately identifies touch indicators and it is able to differentiate between opacity levels, (ii) is capable of reproducing a high percentage of complete scenarios related to crashes and other bugs, with promising results for general user scenarios as well, and (iii) is potentially useful to support real developers during a variety of tasks. V2S can potentially save hundreds of hours per month to developers by letting them to focus their attention on bug triaging and fixing.

Future work can make V2S applicable to different software maintenance tasks, such as: (i) producing scripts with coordinate-agnostic actions, (ii) generating natural language user scenarios, (iii) improving user experience via behavior analysis, (iv) facilitating additional maintenance tasks via GUI-based information, and (v) increasing number of test cases with small perturbations to the main flow of the test scenario.

6.3 Facilitating Bug Reporting

In Chapter 5 of this dissertation, we presented an approach for facilitating recording bug reports. It records user events and GUI information in an Android app that sends the information over a web application capable of displaying and managing bug reports. This facilitates users to send feedback to developers presenting relevant information such screenshots and a set of actions. It provides a fully expressive `html` report with a step by step granularity detail.

As future work the bug reporting application could be further built out to incorporate system level performance information, such as CPU or memory usage. Additionally, the Java web application could be further developed to support additional features commonly found in modern day issue trackers such as issue labels, discussions, and references to code and design artifacts.

Bibliography

[1] 7-eleven https://play.google.com/store/apps/details?id=com.sei.android.

[2] Airbrake https://airbrake.io.

[3] Airtest project http://airtest.netease.com/.

[4] Android fragmentation statistics http://opensignal.com/reports/2014/android-fragmentation/.

[5] Android show touches option https://medium.theuxblog.com/enabling-show-touches-in-android-screen-recordings-for-user-research-cc968563fcb9.

[6] Android ui/application exerciser monkey http://developer.android.com/tools/help/monkey.html.

[7] Android uiautomator http://developer.android.com/tools/help/uiautomator/index.html.

[8] Applause https://www.applause.com.

[9] Apple app store https://www.apple.com/ios/app-store/.

[10] Appsee https://www.appsee.com.

[11] Bugclipper http://bugclipper.com.

[12] Command line tools for recording, replaying and mirroring touchscreen events for android https://github.com/appetizerio/replaykit.

[13] Current number of smartphones in use https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[14] Ffmpeg tool https://www.ffmpeg.org/.

[15] Google firebase test lab robo test https://firebase.google.com/docs/test-lab/robo-ux-test.

[16] Google play screen recording apps https://play.google.com/store/search?q=screen%20recording&c=apps.

[17] Google play store https://play.google.com/store?hl=en.

[18] Google search https://play.google.com/store/apps/details?id=com.google.android.googlequicksearchbox.

[19] HiroMacro auto-touch macro - apps on google play https://play.google.com/store/apps/details?id=com.prohiro.macro&hl=en.

[20] Junit. http://junit.org.

[21] Mr. tappy mobile usability testing device https://www.mrtappy.com.

[22] Mycrowd crowdsourced testing service https://mycrowd.com.

[23] Perfecto. http://www.perfectomobile.com.

[24] Pit. http://pitest.org/.

[25] Proximus https://www.proximus.be.

[26] Qualtrics. http://www.qualtrics.com.

[27] Roboelectric. http://robolectric.org.

[28] Robotium. https://code.google.com/p/robotium/.

[29] [ROOT] bot maker for android - apps on google play https://play.google.com/store/apps/details?id=com.frapeti.androidbotmaker&hl=en.

[30] Rspec. http://rspec.info.

[31] Sauce labs. https://saucelabs.com/features/#features-automated-mobile.

[32] Stackoverflow android screen record https://stackoverflow.com/questions/29546743/what-is-the-frame-rate-of-screen-record/44523688.

[33] Tensorflow object detection api https://github.com/tensorflow/models/tree/master/research/object_detection.

[34] Testbirds crowdsourced testing service https://www.testbirds.com.

[35] Testfairy https://testfairy.com.

[36] Watchsend https://watchsend.com.

[37] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO'04*, pages 1338–1349, 2004.

[38] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *ISSTA'15*, pages 83–93, 2015.

[39] P. Aho, M. Suarez, T. Kanstren, and A.M. Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *ICSTW'14*, pages 343–348, 2014.

[40] M. Ali, M. E. Joorabchi, and A. Mesbah. Same app, different app stores: A comparative study. In *MOBILESoft'17*, pages 79–90, 2017.

[41] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. Memon. Exploiting the saturation effect in automatic random testing of android applications. In *MOBILESoft'15*, pages 33–43, 2015.

[42] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. Memon. Mobiguitar - a tool for automated model-based testing of mobile apps. *IEEE Software*, pages 53–59, 2014.

[43] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *ASE'12*, pages 258–261, 2012.

[44] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *ESE/FSE'12*, 2012.

[45] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE'05*, pages 402–411, 2005.

[46] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[47] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *ISSTA'14*, pages 259–269, 2014.

[48] Apple. App store. `https://itunes.apple.com/us/genre/ios/id36?mt=8`, 2017.

[49] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Trans. Softw. Eng.*, 38(2):258–277, 2012.

[50] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA'13*, pages 641–660, 2013.

[51] L. Bao, J. Li, Z. Xing, X. Wang, and B. Zhou. scvripper: video scraping tool for modeling developers' behavior using interaction data. In *ICSE'15*, pages 673–676, 2015.

[52] G. Bavota, M. Linares-Vásquez, C.E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Trans. on Softw. Eng.*, 41(4):384–407, April 2015.

[53] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *EICS'18*, SIGCHI'18, page 3, 2018.

[54] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk. V2s online appendix `https://sites.google.com/view/video2sceneario/home`, 2019.

[55] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *ICSE'20*, 2020.

[56] C. Bernal-Cárdenas, K. Moran, M. Tufano, Z. Liu, L. Nan, Z. Shi, and D. Poshyvanyk. Guigle: A gui search engine for android apps. In *ICSE'19-C'*, pages 71–74, 2019.

[57] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *FSE'08*, pages 308–318.

[58] S. Beyer and M. Pinzger. A manual categorization of android app development issues on stack overflow. In *ICSME'14*, pages 531–535, Sept 2014.

[59] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marus, M. Di Penta, D. Poshyvanyk, and V. Ng. Assessing the quality of the steps to reproduce in bug reports. In *ESEC/FSE'19*, pages 86–96, 2019.

[60] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *ICSE'18*, pages 665–676, 2018.

[61] N. Chen, S. C.H. Hoi, S. Li, and X. Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In *WSDM '15*, pages 305–314, 2015.

[62] N. Chen, J. Lin, S. Hoi, X. Xiao, and B. Zhang. AR-Miner: Mining informative reviews for developers from mobile app marketplace. In *ICSE'14*, pages 767–778, 2014.

[63] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *OOPSLA'13*, pages 623–640, 2013.

[64] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android:are we there yet? In *ASE'15*, pages 429–440, 2015.

[65] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Softw. Test. Verif. Reliab.*, 21(1):3–28, March 2011.

[66] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. C. Gall. Analyzing reviews and code of mobile apps for better release planning. In *SANER'17*, SANER'17, pages 91–102, 2017.

[67] R. Coelho, L. Almeida, G. Gousios, and A. Van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *MSR'15*, pages 134–145, 2015.

[68] Henry Coles. Mutation testing systems for java compared. `http://pitest.org/java_mutation_testing_systems/`, 2017.

[69] JulietM. Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1990.

[70] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *ISSTA'96*, pages 158–171, 1996.

[71] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[72] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *ICSTW'15*, pages 1–10, 2015.

[73] A. Derezińska and K. Hałas. Analysis of mutation operators for the python language. In *DepCoS-RELCOMEX'14*, pages 155–164, 2014.

[74] D. Di Nardo, F. Pastore, and L. C. Briand. Generating complex and faulty test data through model-based mutation analysis. In *ICST'15*, pages 1–10, 2015.

[75] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes. In *FSE'16*, pages 499–510, 2016.

[76] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, 1984.

[77] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. Barista: A technique for recording, encoding, and running platform independent android tests. In *ICST'17*, pages 149–160, 2017.

[78] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM'03*, 2003.

[79] C. Frisson, S. Malacria, G. Bailly, and T. Dutoit. Inspectorwidget: A system to analyze users behaviors in their applications. In *CHI'16*, pages 1548–1554, 2016.

[80] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR'14*, pages 580–587, 2014.

[81] R. B. Girshick. Fast R-CNN. *CoRR*, 2015.

[82] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE'13*, pages 72–81, 2013.

[83] Google. Lint. `http://developer.android.com/tools/help/lint.html`.

[84] Google. monkeyrunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

[85] Google. Testing ui for a single app. `http://developer.android.com/training/testing/ui-testing/espresso-testing.html`.

[86] Google. Ui/application exerciser monkey. `http://developer.android.com/tools/help/monkey.html`.

[87] Google. Android x86 project. `http://www.android-x86.org`, 2017.

[88] Google. Google play. `https://play.google.com/store`, 2017.

[89] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, , and A. Groce. Measuring effectiveness of mutant sets. In *ICSTW'16*, pages 132–141, 2016.

[90] R. Gopinath, M. Amin Alipour, I. Ahmed, C. Jensen, and A. Groce. On the limits of mutation reduction strategies. In *ICSE'16*, pages 511–522, 2016.

[91] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *ISPASS'15*, pages 215–224, 2015.

[92] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[93] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *WCRE'12*, pages 83–92, 2012.

[94] S. Hao, B. Liu, S. Nath, W.G.J. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys'14*, pages 204–217, 2014.

[95] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR'16*, pages 770–778, 2016.

[96] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 6:65–70, 1979.

[97] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *EuroSys'14*, 2014.

[98] G. Hu, L. Zhu, and J. Yang. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *ESEC/FSE'18*, pages 269–282, 2018.

[99] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *OOPSLA'15*, pages 349–366, 2015.

[100] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *MSR'13*, pages 41–44, 2013.

[101] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE'14*, pages 435–445, 2014.

[102] C. S. Jensen, M. R. Prasad, and A. Moller. Automated testing with targeted event sequence generation. In *ISSTA'13*, pages 67–77, 2013.

[103] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. on Softw. Eng.*, 37(5):649–678, 2011.

[104] N. Jones. Seven best practices for optimizing mobile testing efforts. Technical Report G00248240, Gartner, 2013.

[105] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile apps. In *ESEM'13*, pages 15–24, 2013.

[106] M.E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *ESE'13*, pages 15–24, 2013.

[107] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for java. In *ISSTA'14*, pages 433–436, 2014.

[108] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA'14*, pages 315–326, 2014.

[109] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE'14*, pages 654–665, 2014.

[110] R. Just, G. M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *ICST'12*, pages 720–725, 2012.

[111] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In *ASE'11*, pages 612–615, 2011.

[112] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, pages 70–77, 2015.

[113] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Softw. Test., Verif. Reliab.*, (3):207–225, 2001.

[114] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *ICST'15*, pages 1–10, 2015.

[115] E. Kowalczyk and A. Memon. Extending manual gui testing beyond defects by building mental models of software behavior. In *ASEW'15*, pages 35–41, 2015.

[116] P. Krieter and A. Breiter. Analyzing mobile application usage: generating log files from mobile screen recordings. In *MobileHCI'18*, pages 1–10, 2018.

[117] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS'12*, pages 1097–1105. 2012.

[118] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie. Record and replay for android: are we there yet in industrial cases? In *ESEC/FSE'17*, pages 854–859, 2017.

[119] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[120] C. Mike Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *MobiCom '14*, pages 519–530, 2014.

[121] C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *NDSS'14*, 2014.

[122] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common objects in context. In *ECCV'14*, pages 740–755, 2014.

[123] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *ESEC/FSE 2013*, pages 477–487, 2013.

[124] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *MSR'14*, pages 2–11, 2014.

[125] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *ESEC/FSE'15*, pages 143–154, 2015.

[126] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Multi-objective optimization of energy consumption of guis in android apps. *ACM Trans. Softw. Eng. Methodol.*, 27(3):1–47, 2018.

[127] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Online appendix/replication package for "enabling mutation testing for android apps". `http://android-mutation.com/fse-appendix`.

[128] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Enabling mutation testing for android apps. In *ESEC/FSE'17*, pages 233–244, 2017.

[129] M. Linares-Vásquez, C. Bernal-Cárdenas, G. Bavota, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Gemma: multi-objective optimization of energy consumption of guis in android apps. In *ICSE'17-C*, pages 11–14. IEEE, 2017.

[130] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. How do Developers Test Android Applications? In *ICSME'17*, pages 613–622, 2017.

[131] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *MSR'13*, pages 93–96, 2013.

[132] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *MSR'14*, pages 242–251, 2014.

[133] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *ICSME'17*, pages 399–410, 2017.

[134] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *ICSME'15*, pages 352–361, 2015.

[135] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *MSR'15*, pages 111–122, 2015.

[136] Y. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for java. In *ISSRE'02*, pages 352–366, 2002.

[137] Y. Ma, J. Offutt, and Y. Kwon. Mujava: An automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.

[138] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for android apps. In *ESEC/FSE'13*, pages 224–234, 2013.

[139] L. Madeyski and N. Radyk. Judy - a mutation testing tool for Java. *IET Software*, 4(1):32–42, Feb 2010.

[140] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *FSE'14*, pages 599–609, 2014.

[141] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *ISSTA'16*, pages 94–105, 2016.

[142] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *ASE'17*, pages 16–26, 2017.

[143] Diego Torres Milano. Android ViewServer client. contribute to dtmilano/AndroidViewClient development by creating an account on GitHub. original-date: 2012-02-02T06:04:40Z.

[144] M. B. Miles, A. M. Huberman, and J. Saldaña. *Qualitative Data Analysis: A Methods Sourcebook*. SAGE Publications, Inc, 3rd edition, Apr 2013.

[145] M. Miranda, R. Ferreira, C. R. B. de Souza, F. Figueira Filho, and L. Singer. An exploratory study of the adoption of mobile development platforms by software engineers. In *MOBILESoft'14*, pages 50–53, 2014.

[146] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *ISSRE'15*, pages 461–471, 2015.

[147] K. H. Moller and D. J. Paulish. An empirical investigation of software fault distribution. In *METRICS'93*, pages 82–90, 1993.

[148] Ivan Moore. Jester - the junit test tester. `http://goo.gl/cQZOL1`, 2017.

[149] K. Moran, C. Bernal-Cárdenas, M. Linares-Vásquez, and D. Poshyvanyk. Overcoming Language Dichotomies: Toward Effective Program Comprehension for Mobile App Development. In *ICPC'18*, pages 7–18, 2018.

[150] K. Moran, R. Bonett, C. Bernal-Cardenas, B. Otten, D. Park, and D. Poshyvanyk. On-Device Bug Reporting for Android Applications. In *MOBILESoft'17*, pages 215–216, 2017.

[151] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk. Automated reporting of gui design violations in mobile apps. In *ICSE'18*, pages 165–175, 2018.

[152] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Fusion: A tool for facilitating and augmenting android bug reporting. In *ICSE'16*, 2016.

[153] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *ICSE-C'17*, pages 15–18, 2017.

[154] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson. Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In *ICSE'20*, pages 873–885, 2020.

[155] K. Moran, M. Linares Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In *ESEC/FSE'15*, pages 673–686, 2015.

[156] K. P. Moran, C. Bernal-Cardenas, M. Curcio, R. Bonett, and D. Poshyvanyk. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Trans. on Softw. Eng.*, 2018.

[157] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *ICST'16*, 2016.

[158] M. Nagappan and E. Shihab. Future trends in software engineering research for mobile apps. In *SANER'16*, 2016.

[159] B. Nguyen and A. Memon. An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces. *IEEE Trans. on Softw. Eng.*, 99:216–234, 2014.

[160] T. A. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with remaui. In *ASE'15*, pages 248–259, 2015.

[161] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *MSR'13*, pages 237–246, 2013.

[162] D. Nurmuradov and R. Bryce. Caret-HM: recording and replaying android user sessions with heat map generation using UI state clustering. In *ISSTA'17*, pages 400–403, 2017.

[163] A. J. Offutt and S. D. Lee. How strong is weak mutation? In *TAV'91*, pages 200–213, 1991.

[164] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.*, 7:165–192, 1997.

[165] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *ICSTW'15*, pages 1–10, 2015.

[166] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, 2002.

[167] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *IREC'13*, pages 125–134, 2013.

[168] D. N. Palacio, D. McCrystal, K. Moran, C. Bernal-Cárdenas, D. Poshyvanyk, and C. Shenefiel. Learning to identify security-related issues using convolutional neural networks. In *ICSME'19*, pages 140–144, 2019.

[169] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software*, pages 143–162, 2018.

[170] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *ICSME'15*, pages 291–300, 2015.

[171] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *ICSE'17*, pages 106–117, 2017.

[172] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *ICSME'15*, pages 281–290, 2015.

[173] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu. An experimental evaluation of web mutation operators. In *ICSTW'16*, pages 102–111, 2016.

[174] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *ICSE'16*, pages 571–582, 2016.

[175] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *MobiSys'14*, pages 190–203, 2014.

[176] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS'15*, pages 91–99, 2015.

[177] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: towards real-time object detection with region proposal networks. *IEEE Trans. on Pattern Analy. & Mach. Int.*, (6):1137–1149, 2017.

[178] B. Robinson and P. Brooks. An initial study of customer-reported GUI defects. In *ICSTW'09*, pages 267–274, 2009.

[179] C. Rosen and E Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21:1192–1223, 2016.

[180] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115:211–252, 2015.

[181] R. Sasnauskas and J. Regehr. Intent fuzzer: Crafting intents of death. In *WODA+PERTEA'14*, pages 1–5, 2014.

[182] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *ESE/FSE'09*, pages 297–298, 2009.

[183] R. Schusteritsch, C. Y. Wei, and M. LaRosa. Towards the perfect infrastructure for usability testing on mobile devices. In *CHI'07*, pages 1839–1844, 2007.

[184] D.DJ. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall/CRC, second edition edition, 2000.

[185] D. Shin and D. Bae. A theoretical framework for understanding mutation-based testing methods. In *ICST'16*, 2016.

[186] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR'14*, 2014.

[187] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR'15*, 2015.

[188] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In *ICSE'14*, 2014.

[189] Reel Two. Jumble. `http://jumble.sourceforge.net`.

[190] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104:154–171, 2013.

[191] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta. Release planning of mobile apps based on user reviews. In *ICSE'16*, 2016.

[192] L. Wei, Y. Liu, and S. C. Cheung. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *ASE'16*, pages 226–237, 2016.

[193] M. White, M. Linares-Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In *ICPC'15*, 2015.

[194] Xamarin Inc. Xamarin test cloud. `https://xamarin.com/test-cloud`.

[195] W. Yang, M.R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *FASE'13*, pages 250–265, 2013.

[196] Yuan-W. mudroid project at github. `https://goo.gl/sQo6EL`, 2017.

[197] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *ICST'14*, pages 183–192, 2014.

[198] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *ECCV'14*, pages 818–833, 2014.

[199] H. Zhang. On the distribution of software faults. *IEEE Trans. Softw. Eng.*, 34:301–302, 2008.

[200] H. Zhang and A. Rountev. Analysis and testing of notifications in android wear applications. In *ICSE'17*, 2017.

[201] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. Softw. Eng. Methodol.*, 23(4):32:1–32:28, 2014.

[202] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *ESEC/FSE'15*, pages 214–224, 2015.

[203] C. Zhou and P. G. Frankl. Mutation testing for java database applications. In *ICST'09*, pages 396–405, 2009.