Mobile App Development: Challenges and Opportunities for Automated Support

by

Mona Erfani Joorabchi

B.Sc., Shahid Beheshti University, Iran, 2007

M.Sc., Simon Fraser University, Canada, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

April 2016

© Mona Erfani Joorabchi, 2016

Abstract

Mobile app development is a relatively new phenomenon that is increasing rapidly due to the ubiquity and popularity of smartphones among end-users. As with any new domain, mobile app development has its own set of new challenges. The work presented in this dissertation has focused on improving the state-of-the-art by understanding the current practices and challenges in mobile app development as well as proposing a new set of techniques and tools based on the identified challenges.

To understand the current practices, real challenges and issues in mobile development, we first conducted an explorative field study, in which we interviewed 12 senior mobile developers from nine different companies, followed by a semi-structured survey, with 188 respondents from the mobile development community. Next, we mined and quantitatively and qualitatively analyzed 32K non-reproducible bug reports in one industrial and five open-source bug repositories. Then, we performed a large-scale comparative study of 80K iOS and Android app-pairs and 1.7M reviews by mining the Google Play and Apple app stores.

Based on the identified challenges, we first proposed a reverse engineering technique that automatically analyzes a given iOS mobile app and generates a state model of the app. Finally, we proposed an automated technique for detecting inconsistencies in the same mobile app implemented for iOS and Android platforms. To measure the effectiveness of the proposed techniques, we evaluated our methods using various industrial and open-source mobile apps. The evaluation results point to the effectiveness of the proposed model generation and mapping techniques in terms of accuracy and inconsistency detection capability.

Preface

All of the work presented henceforth was conducted by the author, Mona Erfani Joorabchi. The contributions and evaluations presented in this dissertation are summarized and published in four conference papers. Additionally, the author and an ECE master student, Mohamed Ali, collaborated equally to a conference submission of Chapter 4 which is currently under review.

The following list presents publications for each chapter.

- Chapter 2:
  - "Real Challenges in Mobile App Development" [86]. M. Erfani Joorabchi, A. Mesbah and P. Kruchten. In Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2013). ACM/IEEE. 15–24.
- Chapter 3:
  - "Works For Me! Characterizing Non-reproducible Bug Reports" [87]. M. Erfani Joorabchi, M. Mirzaaghaei and A. Mesbah. In Proceedings of the 11th ACM Working Conference on Mining Software Repositories (MSR 2014). ACM. 62–71.
- Chapter 4:
  - The author and an ECE master student collaborated equally to an ACM SIGSOFT conference submission of this work and it is currently under review.
- Chapter 5:
  - "Reverse Engineering iOS Mobile Applications" [85]. M. Erfani Joorabchi and A. Mesbah. In Proceedings of the 19th IEEE Working Conference on Reverse Engineering (WCRE 2012). IEEE Computer Society. 177–186.
- Chapter 6:

– "Detecting Inconsistencies in Multi-Platform Mobile Apps" [88]. M. Erfani Joorabchi, M. Ali and A. Mesbah. In Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015). IEEE Computer Society. 450–460.

Regarding ethics approval, the following Human Ethics Certificates were obtained from UBC Behavioural Research Ethics Board:

- Project Title "A Study of Cross-platform Development and Testing Practices of Mobile Applications" with Certificate Numbers H12-03058 and H15-02247.

Table of Contents

List of Tables

List of Figures

Acknowledgments

My deepest gratitude is to my supervisor, Dr. Ali Mesbah. I have been amazingly fortunate to have a supervisor who gave me the freedom to explore on my own, and at the same time the guidance to recover when my steps faltered. You have set an example of excellence as a researcher, advisor, mentor, instructor, and role model.

I would like to thank my thesis committee members who were more than generous with their expertise and precious time through this process; your discussion, ideas, and feedback have been absolutely invaluable. A special thank to Dr. Philippe Kruchten for his co-authorship in our ESEM paper.

I also need to thank the very many great friends from the SALT lab for baring with me and providing a great research and social atmosphere at the school.

I would like to acknowledge and thank Quickmobile for allowing me to conduct my research and providing any assistance requested. Special thanks go to the members of the product development department for their continued support.

My deepest levels of gratitude also go to my amazing twin sister and best friend, Minoo, who made a significant mark in my life with her presence, support, and encouragement.

Last but not least, I would like to thank my amazing and supportive friend, Dr. Nima Kaviani. Nima walked alongside me from the very starting point of my Ph.D. career to the end. Thanks Nima for all the help, support, compassion, and for your trustful manner.

Dedication

To the greatest blessings of my life:

"*my mom and my dad, my sister and my brother...*"

Your support, encouragement, and constant love have sustained me throughout my life and successfully made me the person I am becoming. Thank you!

Chapter 1

Introduction

The ubiquity and popularity of smartphones among end-users has increasingly drawn software developers' attention over the last few years. Mobile apps fall broadly into three categories: *native*, *web-based*, and *hybrid* [157]. Native applications run on a device's operating system and are required to be adapted for different mobile devices and *platforms*, such as Apple's iOS, Google's Android, Windows Phone, and Blackberry. While this approach provides the best performance and access to all native platform features, the downside is, in order to build a multi-platform application, the code has to be rewritten for each platform separately. Web-based apps require a web browser on a mobile device. Web technologies such as `HTML`, `CSS`, and `JavaScript` are used to build web-based apps and multiple platforms can be targeted. However, web technologies are not allowed to access all device features and performance can suffer. Hybrid apps are 'native-wrapped' web apps and primarily built using `HTML5` and `JavaScript`. The native-wrapped container provides access to platform features. Recent surveys [50, 51] reveal that developers are mainly interested in building native apps because they offer the best performance and allow for advanced UI interactions. Throughout this dissertation, we focus on native mobile apps. Henceforth, we use the term 'mobile app' or simply 'app' to denote 'native mobile application'.

While mobile app development for devices and platforms, including Nokia, BlackBerry, Android, and Windows Phone goes back over 10 years, there has been exponential growth in mobile app development since the Apple app store launched in July 2008 [213]. Since then, other mobile platforms have opened online stores and marketplaces as their distribution channels for third-party apps; the Android

Market opened a few months later, followed by BlackBerry App World, Nokia's Ovi Store and Windows Phone Marketplace. The easy distribution via the online app stores have significantly lowered the barrier to market entry [64] and, therefore, for the first time in software development history, small companies and single developers access distribution infrastructures that allow them to provide their mobile apps to millions of users at the tap of a finger [186]. On the other hand, users were granted control over the apps they download and install on their mobile devices, and subsequently rate and review apps publicly on the online app stores. As a result, these app stores provide unique and critical communication channels between developers and users, where users can provide relevant information to guide developers in accomplishing their software development, maintenance, and evolution tasks.

Currently, iOS and Android mobile apps dominate the app market each with over 1.5 million apps in their respective app stores, i.e., Apple's AppStore and Android Market, and there are hundreds of thousands of apps on Windows Marketplace and Blackberry AppWorld.[1] Recent estimations indicate that by 2017 over 80 percent of all handset shipments will be smartphones, capable of running mobile apps.[2]

As with any new domain, mobile app development has its own set of new challenges. Researchers have discussed some of the challenges involved in mobile app development [82, 138, 166, 169, 186], however, most of the early related discussions are anecdotal in nature. Additionally, to obtain better insights of issues and concerns in software development, in general, it is a common practice that researchers investigate other sources of software development data. Thus, several studies have made an effort to understand mobile development issues and concerns through (1) mining mobile bug repositories [63, 112, 155] as they have become an integral component of software development activities; (2) mining and analyzing app stores' content, such as user-reviews [71, 124, 185], mobile app descriptions [105, 143, 200], mobile app bytecode [54, 194, 195]; and (3) mining question and answer (QA) websites that are used by developers [62, 148, 149, 212]. While there are substantial qualitative field studies [106, 133, 203] on different areas of soft-

[1] http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/
[2] http://www.prweb.com/releases/2013/11/prweb11365872.htm

ware engineering and traditional software development, limited field studies have been conducted to investigate the actual challenges and issues associated with mobile development.

Thus, we start by conducting the first qualitative field study to gain an understanding of the current practices and challenges in native mobile app development. To this end, we follow a *Grounded Theory* approach, a research methodology stemming from the social sciences [99], which is gaining increasing popularity in software engineering research [44, 133, 199, 204]. Therefore, instead of starting with predetermined hypotheses, we set our objective to discover the process and challenges of mobile app development across multiple platforms. We first conduct and analyze interviews with 12 senior mobile app developers, from nine different industrial companies, who are experts in platforms such as iOS, Android, Windows Mobile/Phone, and Blackberry. Based on the outcome of these interviews, we design and distribute an online survey, which has been completed by 188 mobile app developers worldwide.

Our results reveal challenges of dealing with multiple mobile platforms during mobile development. Mobile devices and platforms are extensively moving toward *fragmentation*, i.e., 1) each mobile platform is different with regard to the programming languages, API/SDK, supported tools, user interface, user experience, and Human Computer Interaction (HCI) standards; 2) on each platform, various devices exist with different properties such as memory, CPU speed, operating system, graphical resolutions, and screen sizes. Developers currently treat the mobile app for each platform separately and manually check that the functionality is preserved and consistent across multiple platforms. Furthermore, mobile developers need better analysis tools in order to track metrics for their apps during the development phase. Additionally, testing is a significant challenge. Current testing frameworks do not provide the same level of support for different platforms and current testing tools do not support important features for mobile testing such as mobility, location services, sensors, or different gestures and inputs.

In our first study, developers mentioned that one of their challenges is to deal with the crashes that are "*very hard to catch and harder to reproduce*" [86]. Thus, we perform the first empirical analysis of *non-reproducible* bug reports. While we start with mobile non-reproducible bugs, we notice that none of the related work

investigates non-reproducible bug reports in isolation. Thus, we expand the study to other software environments and domains. Furthermore, our first field study indicates that to attract as many users as possible, developers often publish the same app for multiple mobile platforms [86]. While several studies have mined and analyzed app stores [71, 124, 185], all of these studies focus on one app store only and none has studied the *same* app published on *different* app stores. Thus, we conduct the first comparative study on mobile *app-pairs*, i.e., the same app implemented for iOS and Android platforms, in order to analyze and compare their various attributes and root causes of user complaints at multiple levels of granularity.

Additionally, a previous study [192] shows that many developers interact with the graphical user interface (GUI) to comprehend the software by creating a mental model of the application. Reverse engineering of desktop user interfaces was first proposed by Memon *et al.* in a technique called GUI Ripping [159]. For web applications, Mesbah *et al.* [163] proposed a crawling-based technique to reverse engineer the navigational structure and paths of a web application under test. Amalfitano *et al.* [49] extend this approach and propose a GUI crawling technique for a small subset of widgets of Android apps. While other related studies [97, 119] focus on Android apps, none has been done to reverse engineer Objective-C iPhone apps automatically. Thus, in order to help developers gain a high-level understanding of their mobile apps, we propose the first automated technique that through dynamic analysis of a given iPhone app, generates a state model of it. This generated model can assist mobile developers to better comprehend and visualize their mobile apps. It can also be used for maintenance, analysis and testing purposes (i.e., smoke testing, test case generation).

Finally, we use model-based techniques in our last study, in order to address a major challenge, faced by industrial mobile developers. The challenge is to keep the app *consistent* and ensure that the behaviour is the same across multiple platforms [86]. Dealing with multiple platforms is not specific to the mobile domain. The problem also exists for cross-browser compatibility testing [74, 161]. However, in the mobile domain, each mobile platform is different with regard to the OS, programming languages, API/SDKs, and supported tools, making it much more challenging to detect inconsistencies automatically. In the mobile domain, Rosetta [100] infers likely mappings between the JavaME and Android graphics

APIs. While none of the related work addresses inconsistency detection across native mobile apps, we propose the first automated technique which for the same mobile app implemented for iOS and Android platforms infers abstract models from the dynamically captured traces and formally compares the app-pair using different comparison criteria to expose any detected inconsistencies.

1.1 Research Questions

The goal of this thesis is to understand the current practices and challenges in mobile app development as well as proposing a new set of techniques and tools based on the identified challenges to help mobile app developers. In order to address this goal, we designed five research questions. The first three research questions aim to obtain better insights regarding issues and concerns in mobile development through (1) interviewing and surveying developers in the field, (2) mining bug repositories, and (3) analyzing app stores' content. The last two research questions are follow-up studies, which address the identified challenges by proposing techniques and tools.

RQ1. *What are the main challenges developers face in practice when they build mobile apps?*

As a preliminary step in our research journey, we start with this basic and critical question. Thus, we conducted a qualitative field study, following a Grounded Theory approach, in which we interviewed 12 senior mobile developers from nine different companies, followed by a semi-structured survey, with 188 respondents from the mobile development community.

RQ2. *What are the characteristics of non-reproducible bug reports and the challenges developers deal with?*

In our first study, developers mentioned that one of their challenge is "*dealing with the crashes that are very hard to catch and harder to reproduce*" [86]. While, ideally each bug report should help developers to find and fix a software fault, there is a subset of reported bugs that is not (easily) reproducible, on which developers spend considerable amounts of time and effort. Although we start with mobile non-reproducible bugs, we notice that none of the related work investigates non-reproducible bug reports in isolation. Thus, we expand the study to other soft-

ware environment and domains. We perform an empirical analysis of bug reports, in particular, characterizing rate, nature, and root causes of 32K non-reproducible reports. We quantitatively compared them with other resolution types, using a set of metrics and qualitatively analyzed root causes of 1,643 non-reproducible bug reports to infer common categories of the reasons these reports cannot be reproduced.

RQ3. *What are the app-store characteristics of the same mobile app, published in different marketplaces? How are the major concerns or complaints different on each platform?*

Furthermore, our first study indicates that to attract as many users as possible, developers often implement and publish the same app for multiple mobile platforms [86]. While, ideally, a given app should provide the same functionality and high-level behaviour across platforms, this is not always the case in practice and there might be known/unknown differences in functionality of the same app-pairs due to many reasons (legal, marketing, platform, API access). For instance, a user of the Android STARBUCKS app complains: "*I downloaded the app so I could place a mobile order only to find out it's only available through the iPhone app. A paying customer is a customer regardless of what phone they have and limiting their access to the business is beyond me.*" An iOS NFL app review reads: "*on the Galaxy you can watch the game live..., on this (iPad) the app crashes sometimes, you can't watch live games, and it is slow.*" Thus, as part of our goal to gain mobile development insights, we conducted a large-scale comparative study on 80K iOS and Android mobile app-pairs, in order to analyze and compare their various attributes, user reviews, and root causes of user complaints at multiple levels of granularity. Since we noticed that most of the related work focused on one app store only and none has studied the *same* app, published on *different* app stores, we mine the two most popular app stores i.e., the Google Play and Apple app stores and employ a mixed-methods approach using both quantitative and qualitative analysis. We also looked for app-pairs in the top rated 100 free and 100 paid apps listed on Google Play and Apple app stores and identify some of the obstacles that prevent developers from publishing their apps in both stores. Additionally, we built three automated classifiers and classified 1.7M reviews to understand how user complaints and concerns vary across platforms.

RQ4. *How can we help developers to better understand their mobile apps?*

Additionally, the previous study [192] shows that many developers interact with the graphical user interface (GUI) to comprehend the software by creating a mental model of the application. Thus, in order to help developers gain a high-level understanding of their mobile apps, we propose a reverse engineering technique that automatically performs dynamic analysis of a given iPhone app by executing the program and extracting information about the runtime behaviour. Our approach exercises the application's user interface to cover the interaction state space. Our tool, called ICRAWLER (iPhone Crawler), is capable of automatically navigating and generating a state model of a given iPhone app. This generated model can assist mobile developers to better comprehend and visualize their mobile apps. It can also be used for maintenance, analysis and testing purposes (i.e., smoke testing, test case generation).

RQ5. *How can we help developers to automatically detect inconsistencies in their same mobile app across multiple platforms?*

Finally, we address a major challenge, we found in our first qualitative study [86]. The challenge, faced by industrial mobile developers, is to keep the app *consistent* and ensure that the behaviour is the same across multiple platforms. This challenge is due to the many differences across the platforms, from the devices' hardware to operating systems (e.g., iOS/Android), and programming languages used for developing the apps (e.g., Objective-C/Java). We found that developers currently treat the mobile app for each platform separately and manually perform screen-by-screen comparisons, often detecting many cross-platform inconsistencies. This manual process is, tedious, time-consuming, and error-prone. Thus, we first identify the most pervasive cross-platform inconsistencies between iOS and Android mobile app-pairs, through industrial interviews as well as a document shared with us by the interviewees, containing 100 real-world cross-platform mobile app inconsistencies. Then, we propose an automated technique and tool, called CHECKCAMP (Checking Compatibility Across Mobile Platforms), which for the same mobile app implemented for iOS and Android platforms instruments and generates traces of the app on each platform for a set of user scenarios. Then, it infers abstract models from the captured traces that contain code-based and GUI-based information for each pair, and formally compares the app-pair using different com-

parison criteria to expose any discrepancies. Finally, it produces a visualization of the models, depicting any detected inconsistencies.

1.2 Contributions

We have conducted a series of studies on different aspects of mobile app analysis. In response to our research questions as outlined in Section 1.1, the following papers have been published and one is currently under review:

- Chapter 2:
  - "Real Challenges in Mobile App Development" [86]. M. Erfani Joorabchi, A. Mesbah and P. Kruchten. In Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2013). ACM/IEEE. 15–24.
- Chapter 3:
  - "Works For Me! Characterizing Non-reproducible Bug Reports" [87]. M. Erfani Joorabchi, M. Mirzaaghaei and A. Mesbah. In Proceedings of the 11th ACM Working Conference on Mining Software Repositories (MSR 2014). ACM. 62–71.
- Chapter 4: The author and an ECE master student collaborated equally to an ACM SIGSOFT conference submission of this work and it is currently under review. We addressed four research questions in this work and my main contributions are:
  - In RQ1, I was responsible for quantitative analysis of 80K app-pairs' attributes. I compared their attributes such as ratings, stars, categories, prices, versions and calculated their statistics to investigate their differences.
  - In RQ1, I equally contributed to categorize the reasons of price fluctuation.
  - I was responsible for RQ2.

  - In RQ3, I manually inspected and labeled 2.1K problematic reviews for training the generic classifier.
  - In RQ3, I manually inspected and labeled 2.1K problematic reviews for training the sentiment classifier.
  - In RQ3, I calculated the statistical results and figures for the generic and sentiment classes.
  - In RQ3, I equally contributed to defining success rate.
  - In RQ4, I was responsible for topic modelling analysis on 20 app-pairs.
  - In RQ4, I equally contributed to defining classes for the complaints classifier.
  - In RQ4, I manually inspected and labeled 500 problematic reviews for the complaints classifier.
  - I equally contributed in writing the paper.

- Chapter 5:

  - "Reverse Engineering iOS Mobile Applications" [85]. M. Erfani Joorabchi and A. Mesbah. In Proceedings of the 19th IEEE Working Conference on Reverse Engineering (WCRE 2012). IEEE Computer Society. 177–186.

- Chapter 6:

  - "Detecting Inconsistencies in Multi-Platform Mobile Apps" [88]. M. Erfani Joorabchi, M. Ali and A. Mesbah. In Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015). IEEE Computer Society. 450–460.

Chapter 2

Real Challenges in Mobile App Development

Summary[3]

Mobile app development is a relatively new phenomenon that is increasing rapidly due to the ubiquity and popularity of smartphones among end-users. The goal of our study is to gain an understanding of the main challenges developers face in practice when they build apps for different mobile devices. We conducted a qualitative study, following a Grounded Theory approach, in which we interviewed 12 senior mobile developers from nine different companies, followed by a semi-structured survey, with 188 respondents from the mobile development community. The outcome is an overview of the current challenges faced by mobile developers in practice, such as developing apps across multiple platforms, lack of robust monitoring, analysis, and testing tools, and emulators that are slow or miss many features of mobile devices. Our initial study was conducted in 2012; to examine whether the results of our study still hold in 2015, we survey 15 senior developers, including the senior interviewees in our earlier study, and report the findings. Based on our findings of the current practices and challenges, we highlight areas that require more attention from the research and development community.

[3]The main study in this chapter appeared at the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2013) [86].

2.1 Introduction

The ubiquity and popularity of smartphones among end-users has increasingly drawn software developers' attention over the past recent years. Currently, iOS and Android each have over 1.5 million mobile apps on Apple's AppStore and Android Market, and there are hundreds of thousands of apps on Windows Marketplace and Blackberry AppWorld.[4] Recent estimations indicate that by 2017 over 80 percent of all handset shipments will be smartphones, capable of running mobile apps.[5]

As with any new domain, mobile application development has its own set of new challenges, which researchers have recently started discussing [46, 138, 166]. Kochhar *et al.* [138] discussed the test automation culture among app developers by surveying Android and Windows app developers. Miranda *et al.* [166] reported on an exploratory study through semi-structured interviews. Most early related discussions [82, 92, 169, 213], however, are anecdotal in nature. While there are substantial qualitative studies on different areas of software engineering, limited studies have been conducted to investigate the challenges that mobile app developers face in practice.

The goal of our study is to gain an understanding of the current practices and challenges in native mobile app development. To this end, we conducted an explorative study by following a Grounded Theory approach, a research methodology stemming from the social sciences [99], which is gaining increasing popularity in software engineering research [44, 133, 199, 204]. Thus, instead of starting with predetermined hypotheses, we set our objective to discover the process and challenges of mobile app development across multiple platforms. We started by conducting interviews with 12 senior mobile app developers, from nine different industrial companies. The developers are experts in building mobile apps for platforms such as iOS, Android, Windows Mobile/Phone, and Blackberry. Based on the outcome of these interviews, we designed and distributed an online survey, which was properly completed by 188 mobile app developers worldwide.

Our results reveal challenges of dealing with multiple mobile platforms during mobile development. While mobile devices and platforms are extensively moving

[4] http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[5] http://www.displaysearch.com/pdf/131121_smartphones_to_pass_global_mobile_phone_shipments_by_2017.pdf

toward fragmentation, the contemporary development process is missing the adaptation to leverage knowledge from platform to platform. Developers currently treat the mobile app for each platform separately and manually check that the functionality is preserved across multiple platforms. Furthermore, mobile developers need better analysis tools in order to track metrics for their apps during the development phase. Additionally, testing is a significant challenge. Current testing frameworks do not provide the same level of support for different platforms. Additionally, platform-supported tools are required, as the current third party testing solutions have limited support for important features of mobile testing such as mobility, location services, sensors, or different gestures and inputs.

2.2 Study Design

The objective of our study is to gain an understanding of the challenges mobile app developers face in practice.

2.2.1 Methodology

Considering the nature of our research goal, we decided to conduct a qualitative study by following a Grounded Theory approach [79, 99]. Grounded Theory is best suited when the intent is to learn how people manage problematic situations and how people understand and deal with what is happening to them [45]. It is also useful when the research area has not been covered in previous studies [130] and the emphasis is on new theory generation [78], i.e., understanding a phenomenon. Grounded Theory has been gaining more traction in software engineering research recently [44, 70, 106, 132, 133, 139, 191, 203, 204].

2.2.2 Data Collection and Analysis

Our approach for conducting a Grounded Theory research includes a combination of interviews and a semi-structured survey. The interviews targeted *experts* in mobile app development and the survey was open to the general mobile development community.

Our interviews were conducted in an iterative style, and they are at the core of the data collection and analysis process. At the end of each interview, we asked

the interviewees for feedback on our set of questions; what is missing and what is redundant. The analytical process involves collecting, coding and analyzing data after each interview while developing theory simultaneously. From the interview transcripts, we analyze the data line-by-line, break down interviews into distinct units of meaning (sentences or paragraphs), allocate *codes* to the text and label them to generate *concepts* to these units. Our codes, where appropriate, are taken from the text itself. Otherwise, they are created by the authors to capture the emerging concepts. Furthermore, these concepts are then clustered into descriptive *categories*. They are re-evaluated and subsumed into higher-order categories in order to generate an emerging theory. Theoretical sampling evolves into an ever-changing process, as codes are analyzed and categories and concepts continue to develop [77]. We perform *constant comparison* [99] between the analyzed data and the emergent theory until additional data being collected from the interviews adds no new knowledge about the categories. Thus, once the interviewees' answers begin to resemble the previous answers, a state of *saturation* [98] is reached, and that is when we stop the interviewing process.

Based on the theory emerging from the interview phase, we designed a semi-structured survey, as another source of data, to challenge this theory. Before publishing the survey and making it publicly available, we asked four external people – one senior Ph.D. student and three mobile app developers – to review the survey in order to make sure all the questions were appropriate and easily comprehensible. Most of our survey questions are closed-ended, but there are also a few optional open-ended questions for collecting participants' 'insights' and 'experiences'. The responses to these open-ended questions are fed into our coding and analysis step to refine the results, where applicable. This survey, as distributed to participants, is available online.[6] Additionally, the first page of the survey includes the purpose and procedures of the study, potential risks and benefits, privacy and confidentiality, our contact information and consent.

[6] http://www.ece.ubc.ca/~merfani/survey.pdf

Table 2.1: Interview Participants.

ID	Role	Platform Experience	Software Dev Exp (yr)	Mobile Dev Exp (yr)	Company (Mobile Dev Team Size)	Company's Platform Support
P1	iOS Lead	iOS, Android	6-10	6	A (20)	iOS, Android, Windows, Blackberry
P2	Android Lead	Android, iOS	6-10	6	A (20)	iOS, Android, Windows, Blackberry
P3	Blackberry Lead	Blackberry, iOS, Android	6-10	6	A (20)	iOS, Android, Windows, Blackberry
P4	iOS Lead	iOS	6-10	3-4	B (2-5)	iOS, Android
P5	Android Lead	Android	6-10	3	B (2-5)	iOS, Android
P6	iOS Dev	iOS	4-5	3-4	C (20+)	iOS, Android
P7	Windows Mobile Dev	Windows, Android	10+	2	D (1)	Windows
P8	Android Dev	Android	4-5	2-3	E (2-5)	iOS, Android
P9	Android Lead	Android, iOS, Windows	10+	5-6	F (6-10)	iOS, Android, Windows
P10	iOS Dev	iOS, Android	10+	3	G (1)	iOS, Android
P11	Android Lead	Android, Blackberry	10+	6+	H (1)	Android, Blackberry
P12	iOS Dev	iOS, Windows	10+	2-3	I (2-5)	iOS, Windows

2.2.3 Participant Demographics

Interviews. We interviewed 12 experts from nine different companies in Canada. Each interview session took on average around 30 minutes. We recorded audio in the interview sessions and then transcribed them for later analysis. Table 2.1 presents each participant's role in their company, the mobile platforms they have expertise in, the number of years they have work experience in software development and in mobile app development, the size of the mobile development team, and finally all the mobile platforms that each company supports. Regarding the participants' experience in developing mobile apps, five have around six years, four have 3–4 years and three have 2–3 years of experience. Five participants are mainly iOS experts, five are Android experts, one is a Windows expert, and finally one is a Blackberry expert. In addition, the category distribution of their apps includes Tools/Utilities, Business, Social Networks, Maps/Navigation, Games, Education, Travel, Music, Videos, Sports, Entertainment, and (events and conferences, medical professionals and self-improvement) categories.

Survey. Our survey was fully completed by 188 respondents. We released the survey to a wide variety of mobile development groups. We targeted the popular Mobile Development Meetup groups, LinkedIn groups related to the native mobile development and shared the survey through our Twitter accounts. We kept the survey live for two and a half months. No reward or incentive (such as donations) was offered to the participants. In our attempt to distribute our online survey, it was interesting to see people's reactions; they *liked* our post on LinkedIn groups and gave encouraging comments such as "*I hope it will help to make mobile app developers' lives easier*". The following shows our original invitation message:

> Are you a native mobile app developer? Please participate in our research study to help us understand the real challenges: Link to the survey
> Thanks,

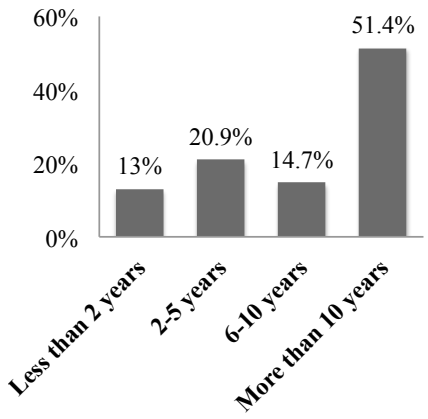


Figure 2.1: How many years of work experience do you have in software development:

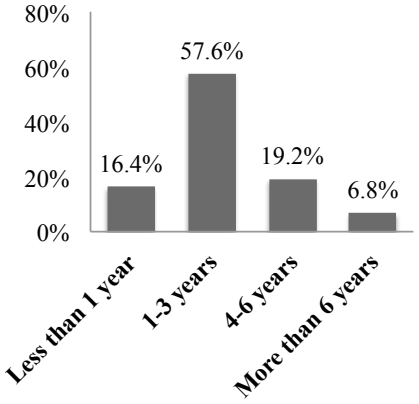


Figure 2.2: How many years of work experience do you have in native mobile application development:

The demographics of the participants in the survey are as follows. 92% were male and 5% female; they come from USA (48%), India (11%), Canada (10%), Israel (5%), The Netherlands (3%), UK (3%), New Zealand (2%), Mexico (2%), and 15 other countries.

The histogram of their work experience in software development is shown in Figure 2.1, where 52% have more than 10 years, 15% between 6–10 years, 20% between 2–5 years, and 13% less than 2 years. Their experience in native mobile development, shown in Figure 2.2, ranges from 6% more than 6 years, 19% between 4–6 years, 59% have between 1–3 years, to 16% less than 1 year.

The platforms they have expertise in are presented in Figure 2.3, which include 72% iOS, 65% Android, 26% Windows, 13% Blackberry, and 6% chose others (e.g., Symbian, J2ME). In terms of the number of mobile apps they have developed, Figure 2.4 depicts 64% have developed less than 10 apps, 22% have developed 10–20 apps and the rest more than 20 apps. As shown in Figure 2.5, they built different types of apps such as Tools/Utilities, Business, Social Networks, Maps, Games, Education and more.

Figure 2.6 indicates that 25% are freelance mobile developers, 33% work in a company while 33% do both. As shown in Figure 2.7, 42% work with 2–5 native

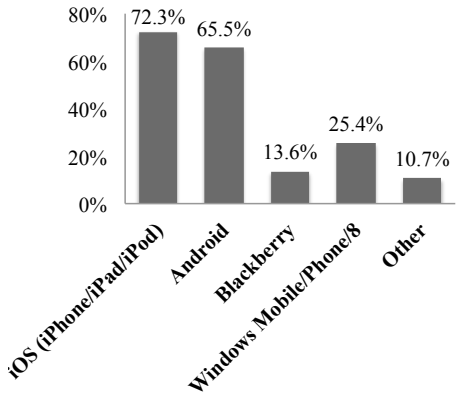


Figure 2.3: What platforms do you develop native mobile applications for (Check all that apply):

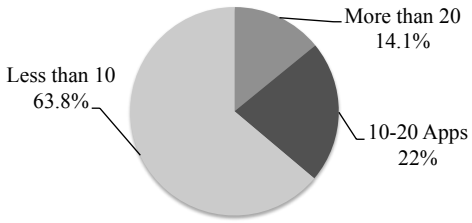


Figure 2.4: How many native mobile applications have you developed so far:

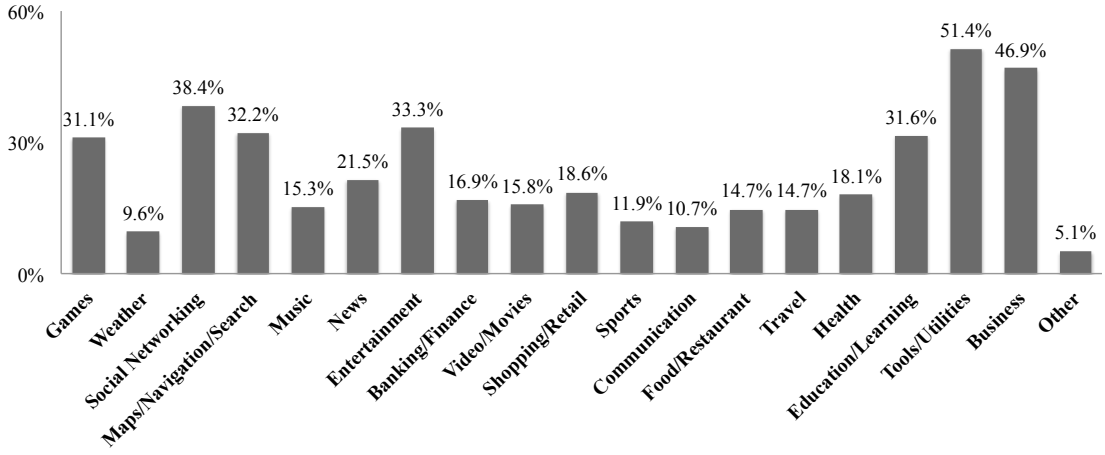


Figure 2.5: What types of native apps have you built (check all that apply)?

mobile app developers (including developers for different platforms), 27% are the only mobile app developers and 15% work with 6–10 other developers.

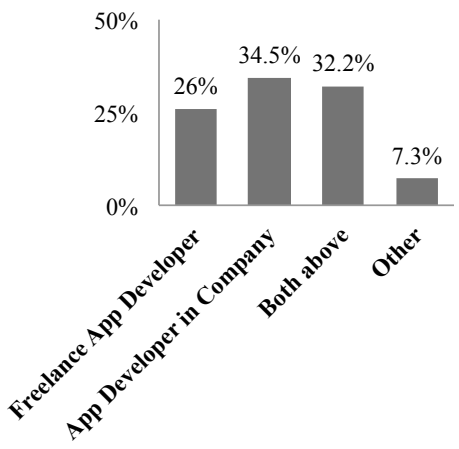


Figure 2.6: Which of the following applies to you:

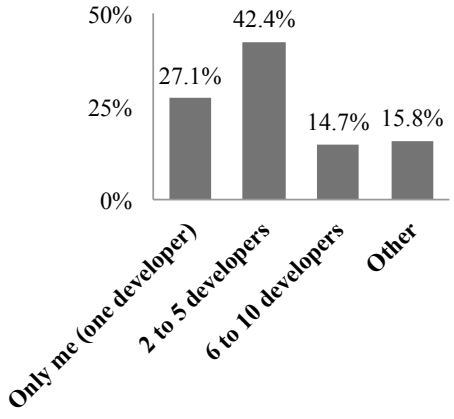


Figure 2.7: If you are working in a company, how big is the native mobile application developer team (including developers for different platforms)?

2.3 Findings

The findings from our study consist of 4 main *categories* and 31 subordinate *concepts*. Figure 2.8 presents an overview of our results. For each concept, appropriate codes and quotes are presented in this section.

In addition to the general challenges faced by mobile developers (Section 2.3.1), two major themes emerged from the study, namely (1) challenges of developing mobile apps across multiple platforms (Section 2.3.2), and (2) current practices (Section 2.3.3) and challenges (Section 2.3.4) of mobile app analysis and testing.

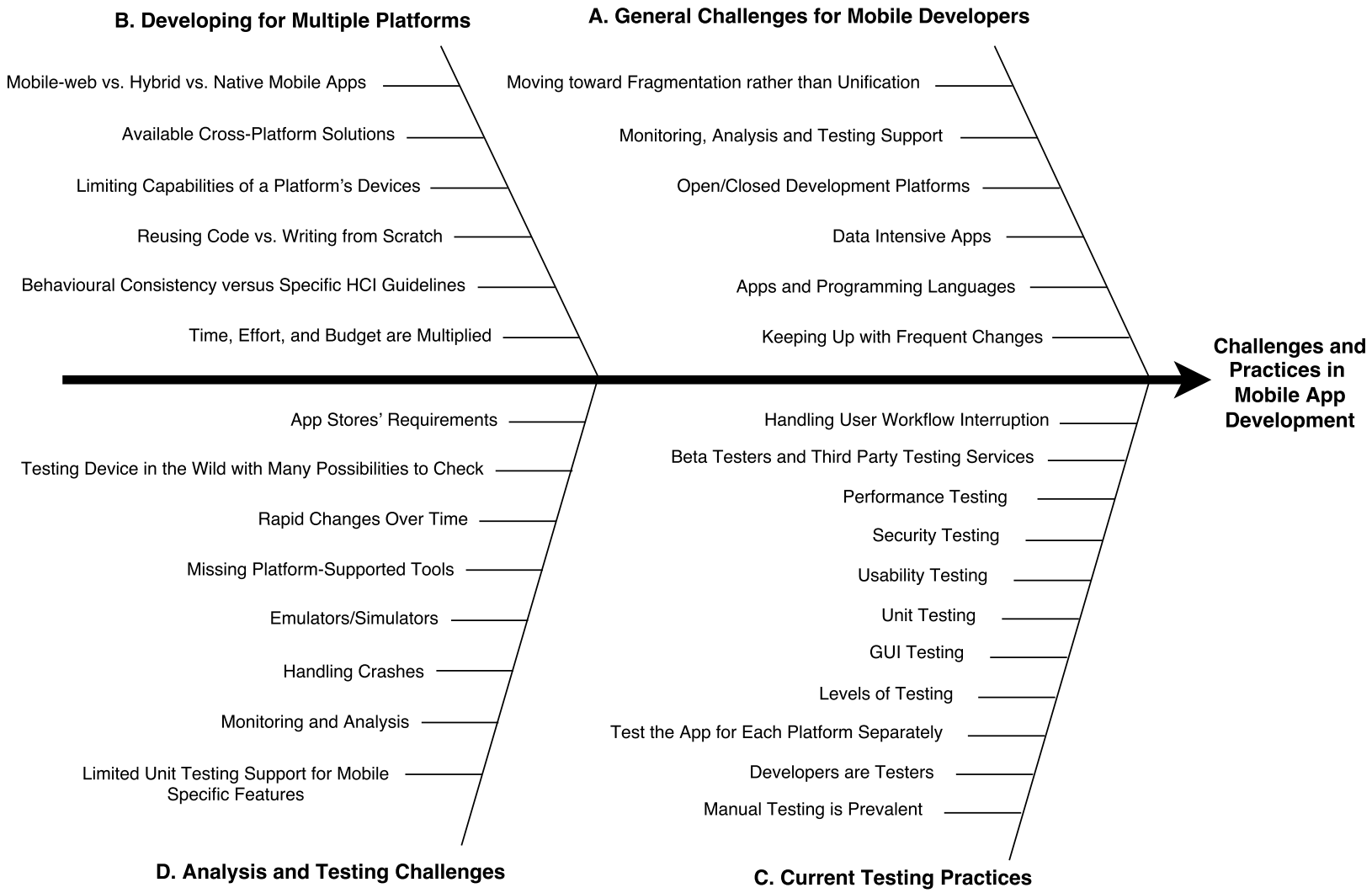


Figure 2.8: An overview of our four main *categories* with 31 subordinate *concepts*.

2.3.1 General Challenges for Mobile Developers

In this subsection, we present the most prominent general challenges faced by mobile app developers, emerging from our study results.

Moving toward Fragmentation rather than Unification

76% of our survey participants see the existence of multiple mobile platforms as a challenge for developing mobile apps, while 23% believe it is an opportunity for technology advances that drive innovation (See Figure 2.9).

More than half of the participants mentioned that mobile platforms are moving toward fragmentation rather than unification:

- *Fragmentation across platforms:* Each mobile platform is different with regard to the user interface, user experience, Human Computer Interaction (HCI) standards, user expectations, user interaction metaphors, programming languages, API/SDK, and supported tools.
- *Fragmentation within the same platform:* On the same platform, various devices exist with different properties such as memory, CPU speed, and graphical resolutions. There is also a fragmentation possible on the operating system level. A famous example is a fragmentation on Android devices with different screen sizes and resolutions. Almost every Android developer in both our interviews and survey mentioned this as a huge challenge they have to deal with on a regular basis.

Furthermore, device fragmentation is not only a challenge for development but also for testing. All of our participants believe that platform versioning and upgrading is a major concern; For example, a respondent said: "*at the OS level, some methods are deprecated or even removed.*" So developers need to test their apps against different OS versions and screen sizes to ensure that their app works. Subject P5 said they mostly maintain "*a candidate list of different devices and sizes*". P11 explained, "*because we monitor our application from the feedback of the users, we tend to focus on testing the devices that are most popular.*" Thus, the current state of mobile platforms adds another dimension to the cost, with a wide variety of devices and OS versions to test against. P11 continued, "*right now we*

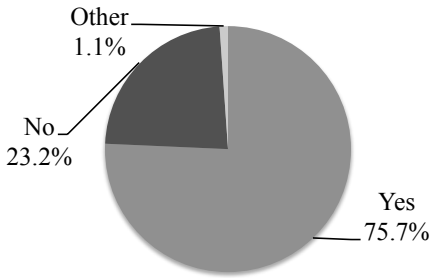


Figure 2.9: Do you see the existence of multiple platforms as a challenge for developing mobile applications and why?

support 5 or 6 different (app) versions only because there are different OS versions, and on each of those OS versions we also have 3–4 different screen sizes to make sure the application works across each of the Android versions." A respondent stated, "*we did a code split around version 2.3 (Android). So we have two different versions of the applications: pre 2.3 version and post 2.3 version. And in terms of our policy, we made that decision since it is too difficult to port some features.*"

Monitoring, Analysis and Testing Support

"*Historically, there has almost been no one doing very much in mobile app testing*", stated P10 and explained that until fairly recently, there has been very little testing, and very few dedicated testing teams. However, that is changing now and they have started to reach out for quality and testing. Automated testing support is currently very limited for native mobile apps. This is seen as one of the main challenges by many of the participants. Current tools and emulators do not support important features for mobile testing such as mobility, location services, sensors, or different gestures and inputs. Our results indicate a strong need of mobile app developers for better analysis and testing support. Many mentioned the need to monitor, measure, and visualize various metrics of their apps through better analysis tools.

Open/Closed Development Platforms

Android is open source whereas iOS and Windows are closed source. Some participants argued that Apple and Microsoft need to open up their platforms. P5 explained: "*We have real challenges with iOS, not with Android. Because you*

don't have API to control, so you have to jump into loops and find a back door because the front door is locked. Whatever Apple allows is not enough sometimes." An example of such lack of control is given: "*to find out whether we are connected to the Bluetooth.*" On the other hand, P9 explained that because Android is open source and each manufacturer modifies the source code to their own desires and releases it, sometimes they do not stick to the standards. A simple example is provided: "*the standard Android uses commas to separate items in a list, but Samsung phones use a semicolon.*" A respondent stated, "*Many Android devices have been badly customized by carriers and original equipment manufacturers.*"

Data Intensive Apps

Dealing with data is tricky for apps that are data intensive. As a respondent explained: "*So much data cannot be stored on the device, and using a network connection to sync up with another data source in the backend is challenging.*" Regarding offline caching in hybrid solutions, P1 said: "*Our apps have a lot of data and offline caching doesn't seem to really work well.*"

Apps and Programming Languages

Two of our participants explained that there have been a number of comparisons (e.g., performance-wise) between programming languages used for native mobile development such as `Java`, `C`, and `Objective-C`. `Java` has huge benefits of being platform independent and popular language with many resources and third party libraries compares to `Objective-C`. However, P3 stated that `Java` is not as efficient on a mobile device and slow. P1 elaborated that "*while Apple had built Cocoa framework over many years, recently* `Objective-C` *is accepted by the iOS development community. Going with* `Java` *would negate a lot of their advantages that they have in Cocoa.*"

Keeping Up with Frequent Changes

One type of challenge mentioned by many developers is learning more languages and APIs for the various platforms and remaining up to date with highly frequent changes within each software development kit (SDK). "*Most mobile developers*

will need to support more than one platform at some point", a respondent stated. "*Each platform is totally different (marketplaces, languages, tools, design guidelines), so you need experts for every one of them. Basically, it is like trying to write simultaneously a book in Japanese and Russian; you need a native Japanese and a native Russian, or quality will be ugly*", explained another respondent. As a result, learning another platform's language, tools, techniques, best practices, and HCI rules is challenging.

While many developers complained about learning more languages and APIs for the various platforms and the lack of an integrated development environment that supports different mobile platforms, P1 explained: "*Right now we develop in two main platforms: iPhone and Android. That is not really that hard, the native SDKs are pretty mature and they are easy to learn. Additionally, it is not required to have hundreds of thousands of lines of code to do something. You have 50 thousand lines of code and you have a complex app.*"

2.3.2 Developing for Multiple Platforms

67% of our interview participants and 63% of our survey respondents have experienced developing the *same* app for more than one mobile platform.

Mobile-web vs. Hybrid vs. Native Mobile Apps

Subjects P1 and P8 support developing hybrid apps. The remaining 10 interviewees are in favour of building pure native apps and believe that the current hybrid model tends to look and behave much more like web pages than mobile applications. P11 argued that "*the native approach offers the greatest features*" and P4 stated, "*user experience on native apps is far superior [compared] to a web app.*" In a number of cases, the participants had completely moved away from the hybrid to the native approach. A recurring example given is Facebook's switch from an `HTML5`-based mobile app to a native one.

On the other hand, P1 argued that "*it really depends on the complexity and type of the application*", for example, "*information sharing apps can easily adopt the hybrid model to push news content and updates across multiple platforms.*"

In the survey, 82% responded having native development experience, 11% have

tried hybrid solutions, and 7% have developed mobile web apps. Most respondents are in favour of the native approach. Regarding mobile web, they stated that: "*Mobile web doesn't feel or look like any of the platforms.*" Or: "*You will never be able to control some hardware through the web.*" Or: "*Mobile web is a powerful tool appropriate for some uses, but is not extensive enough to replace native.*"

Regarding the hybrid approach, others said that: "*HTML5 has much potential and will likely address many of the current problems in the future as it saves development time and cost*"; or: "*since many big players are investing a lot in HTML5, it may take a big chunk of the front-end side when it becomes stable.*"

Regarding native approach, they stated that: "*There will always be a demand for the specificity of a native app.*" Or: "*In some fields, web or hybrid will prevail; but there are many cases where we need a native app.*" Most of the participants argued that when development cost is not an issue, companies tend to develop native apps. Of course it also depends on the application type; where better user experience or device specific features are needed, native seems to be a clear choice.

Lastly, when we asked our participants that whether native app development will be replaced by hybrid solutions or mobile web development due to its challenges, all the interviewees and 70% of survey participants disagreed, and 10% indicated that there will always be a combination of native and hybrid approaches: "*They'll coexist as they do today in PC world.*"

Available Cross-Platform Solutions

Regarding common practices for building cross-platform apps, our participants explained that a variety of technologies exists. Hybrid approaches have the concept of recompilation in native with the power of cross-platform execution. PHONEGAP, APPCELERATOR TITANIUM, XAMARIN, CORONA[7] and many other tools exist, which follow different approaches and some have their own SDK. As expected, there are different attitudes towards them since there are no silver bullets yet defined. For instance, P1 explained that "*with* PHONEGAP *one basically writes* `HTML` *and* `JavaScript` *code and it is translated to the native libraries, but they have performance and user experience drawbacks.*" A respondent said,

[7] http://coronalabs.com/

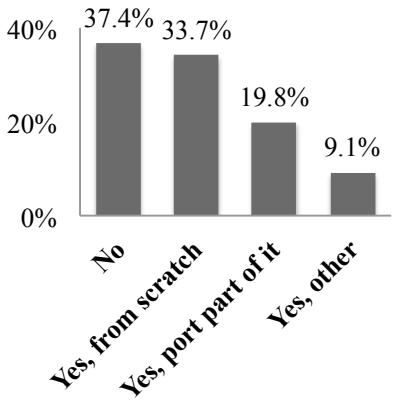


Figure 2.10: Have you developed the *same* native mobile app across different platforms?

"*I would like to see mobile web applications or hybrid frameworks (e.g.* PHONEGAP/APPCELERATOR TITANIUM*) reach a level of responsive user experience that truly mimics the experience users will find native to the platform.*"

Limiting Capabilities of a Platform's Devices

Not all devices and operating systems of a platform have the same capabilities. For instance, Android has different versions and browsers in some of those versions have poor support for `HTML5`. Most of the participants in favour of the hybrid approach believe that once the adaptation is complete (e.g., with mature web browsers in the platforms), there would be more interest from the community for hybrid development.

Reusing Code vs. Writing from Scratch

67% of our interview participants have tried both methods of writing a native mobile app from scratch for a different platform and reusing some portions of the same code across platforms. The majority stated that it is impossible or challenging to port functionality across platforms and that when a code is reused in another platform, the quality of the results is not satisfactory.

Figure 2.10 shows that out of the 63% survey respondents, who have experienced developing mobile apps across different platforms, 34% have written the

same app for each platform from scratch, and 20% have experienced porting some of the existing code. A respondent said, "*every platform has different requirements for development and porting doesn't always produce quality*"; or: "*At this moment, I believe that it is best to create the apps from scratch targeting the individual OS.*" P11 argued that "*we ported a very little amount of the code back and forth between Android and Blackberry, but we typically write the code from scratch. While they both use Java, they don't work the same way. Even when basic low levels of Java are the same, you have to rewrite the code.*"

In addition to the differences at the programming language level (e.g., `Java` versus `Objective-C`), P9 elaborated why migrating code does not work: "*A simple example is the way they [platforms] process push messages. In Android, a push message wakes up parts of the app and it requests for CPU time. In iOS, the server would pass the data to Apple push server. The server then sends it to the device and no CPU time to process the data is required.*" These differences across platforms force developers to rewrite the same app for different platforms, with no or little code reuse. This is seen as one of the main disadvantages of native app development.

Behavioural Consistency versus Specific HCI Guidelines

Ideally, a given mobile app should provide the same functionality and behaviour regardless of the target platform it is running on. However, due to the internal differences in various mobile devices and operating systems, "*a generic design for all platforms does not exist*"; For instance, P12 stated that "*an Android design cannot work all the way for the iPhone.*" This is mainly due to the fact that HCI guidelines are quite different across platforms, since no standards exist for the mobile world, as they do for the Web for instance. Thus, developers are constantly faced with two competing requirements:

- *Familiarity for platform users:* Each platform follows a set of specific HCI guidelines to provide a consistent look-and-feel across applications on the same device. This makes it easier for end users to navigate and interact with various applications.
- *Behavioural consistency across platforms:* On the other hand, developers

would like their application to behave similarly across platforms, e.g., user interaction with a certain feature on Blackberry should be the same as on iPhone and Android.

Thus, creating a reusable basic design that will translate easily to all platforms while preserving the behavioural consistency is challenging. As P9 stated: "*The app should be re-designed per platform/OS to make sure it flows well*"; A respondent put it: "*We do screen by screen design review for each new platform*"; or: "*Different platforms have different strengths and possibilities. It is foolish to try to make the apps exactly the same between platforms*"; and: "*It requires multi-platform considerations at the designing stage and clever decisions should be made where platform-specific design is necessary.*" Other respondents explained, "*You are writing a lot of native code, across all the different platforms. You have to write it 3-4 times. One of the problems is keeping them consistent. There are UI and UX differences between different platforms. Android is different from iPhone app and you also want to take advantage of native, but you also want to make sure that the same app is consistent across all the different platforms.*"

Time, Effort, and Budget are Multiplied

Due to the lack of support for automated migration across platforms, developers have to redesign and reimplement most of the application. Although the application definition, the logic work, and backend connectivity would be similar regardless of platform, the production phase would require adaptations to create native applications for each platform. This is because the flow of the application development is different; iOS uses `Objective-C` for development, Android uses `Java`, etc. Therefore, creating quality products across platforms is not only challenging but also time-consuming and costly, i.e."*developing mobile apps across platforms natively is like having a set of different developers per each platform*", stated P11. As a result, "*re-coding against wildly different API sets*" increases the cost and *time-to-market* within phases of design, development, testing, and maintenance, which is definitely a large issue for start-up and smaller companies.

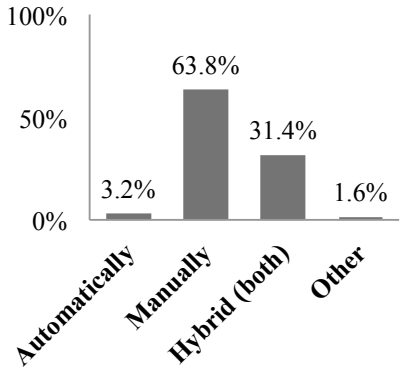


Figure 2.11: How are your native mobile apps tested?

2.3.3 Current Testing Practices

As outlined in Subsection 2.3.1, many developers see analysis and testing of mobile apps as an important activity to provide dependable solutions for end-users. Our study results shed light on the current practices of mobile application analysis and testing.

Manual Testing is Prevalent

As shown in Figure 2.11, 64% of our survey participants test their mobile apps manually, 31% apply a hybrid approach, i.e., a combination of manual and automated testing and only 3% engage in fully automated testing. P3 explained: "*Right now, manually is the best option. It's kind of like testing a new game, testing on consoles and devices. It is that kind of testing I believe just maybe smaller, but you have to worry about more platforms and versions.*" A respondent stated: "*Organizations, large and small, believe only in manual testing on a small subset of devices*"; and another one said: "*It's a mess. Even large organizations are hard to convince to do automated testing*", or "*I have not used automation testing. I've heard good and bad about it, but most of my apps are small enough to be manually tested.*"

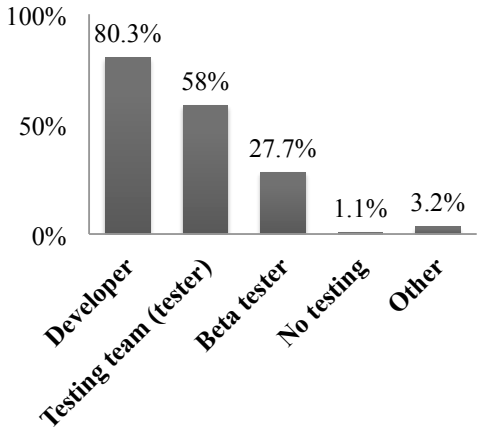


Figure 2.12: Who is responsible for testing your native mobile apps?

Developers are Testers

There are different combinations of testing processes and approaches currently taken by the industry. They can be categorized based on a company's size, clients, development culture, testing policy, application type, and the mobile platforms supported. These testing approaches are performed by various people such as developers, testing teams, beta testers, clients, as well as third-party testing services. As indicated in Table 2.1, our interviewees' companies vary from small size with 1–2 developers to larger mobile development companies or teams with over 20 developers. As expected, larger companies can afford dedicated testing teams or employ beta field testing while in smaller companies testing is mainly done by developers or clients (end-users) and in more informal and ad-hoc way. Additionally, large teams use more devices for manual testing, and have tried to automate part of the testing procedure such as data creation, building and deploying the app into devices.

Figure 2.12 depicts the results of our survey with regard to roles responsible for testing. 80% of the respondents indicated that the developers are the testers, 53% have dedicated testing teams or testers, and 28% rely on beta testers.

The majority of the participants, with or without testing teams, stated that after developing a new feature, the developers do their own testing first and make sure it is functional and correct. This is mostly manual testing on simulators and if

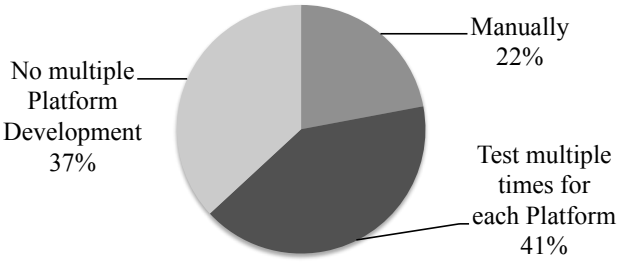


Figure 2.13: How do you test your application's correctness across multiple platforms?

available on physical devices.

Test the App for Each Platform Separately

Our interviews reveal that app developers treat each platform completely separately when it comes to testing. Currently, there is no coherent method for testing a given mobile app across different platforms; being able to handle the differences at the UI level is seen as a major challenge. Testers write "*scripts that are specific for each platform*", and they "*are familiar with the functionality of the app, but are testing each platform separately and individually*". We also noticed that there are usually separate teams in the same company, each dedicated to a specific platform with their own set of tools and techniques; P6, an iOS developer, said: "*I am not sure about Android, as the teams in our company are so separate and I don't even know what is going on with the other side.*" Responses provided by 63% of our survey participants, who develop the *same* native mobile app for more than one platform, confirmed the interview results, stating: "*The test cases apply to each platform, but they must be implemented uniquely on each platform*", or: "*I have to do it twice or more depending on how many platforms I have to build it on*", or: "*Treat them as separate projects, as they essentially are, if native. Do testing independently!*"

Figure 2.13 shows that, out of 63% of our survey participants that develop the *same* native mobile app for more than one platform, 22% test their apps manually, and 41% indicated that they "*test the app for each platform separately*".

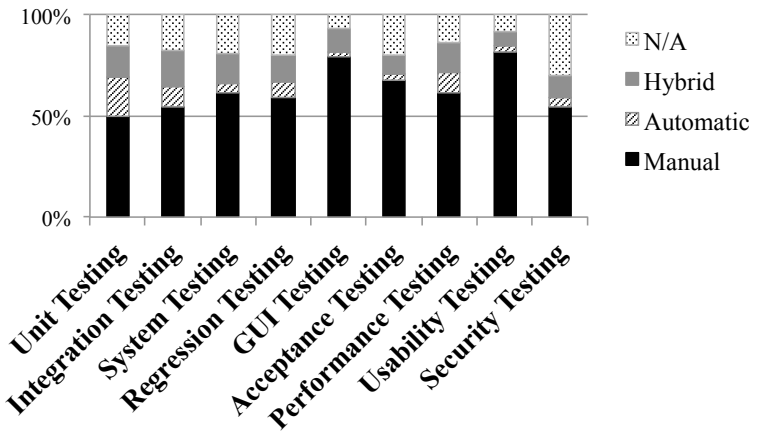


Figure 2.14: What levels of testing do you apply and how?

Levels of Testing

Levels of testing refer to the stages of testing such as unit, integration, system, regression, GUI, etc. Our study aimed to determine the existence, value and process of the testing stages in mobile app development. Figure 2.14 illustrates different levels of testing applied on mobile apps. There is very little automation for different levels of testing, e.g., around 3% for each of GUI, acceptance, and usability testing. P2 noted: "*It is not really well structured or formal what we do. We do some pieces of all of them [ad-hoc] but the whole testing is a manual process.*"

GUI Testing

More than half of the participants admitted that GUI testing is challenging to automate. P2 said: "*Automated UI testing is labor intensive, and can cause inertia when you want to modify the UI. We have a manual tester, core unit testing, then employ beta field testing with good monitoring.*"

P7 stated: "*Our company has Microsoft products. With Microsoft studio interface, you can emulate a lot of sensors for testing GUI whereas in Eclipse for Android, you need to click a lot of buttons. You can emulate the position in your phone, but Android doesn't do this.*"

P3 elaborated: "*Blackberry is actually really hard to create test scripts for GUI*

testing. Because it is not like other platforms, which are touch-based and layout-based. With Blackberry, you have to know what field manager is and it is hard to actually get this information by clicking on buttons. You have to go through the whole array of elements."

Some tools were highlighted such as ROBOTIUM[8] and MONKEYRUNNER[9] for Android. A few iOS developers said they have tried MONKEYTALK[10] (formerly called FONEMONKEY) and KIF[11] for GUI testing; P1 stated: "*I find KIF to be a lot more mature than automation testing provided by Apple, esp. if you want to automate using a build server. Even with KIF you have to write a lot of* `Objective-C` *code to work properly. But it is still hard to be used for our custom and dynamic applications.*"

Unit Testing

Our study shows that the use of unit testing in the mobile development community is relatively low. Both interview and survey results (See Figure 2.14) reveal that unit testing for native mobile apps was not commonplace in 2012, however, that is changing recently (see Section 2.4). Figure 2.14 shows levels of testing for all participants. The comparison of automation testing across the main platforms shown in Figure 2.15 – Figure 2.18.

While some respondents argued that "*the relatively small size of mobile apps makes unit testing overkill and deciding whether it's worth writing unit tests or save the time and test manually is always difficult*"; or: "*Complete unit testing to get full coverage is overkill. We only unit test critical code*"; or: "*Small projects with small budgets - the overhead of creating rigorous test plans and test cases would have a serious impact on the budget.*" On the other hand, others said that "*the rapidly changing user expectations and technology mean unit testing is crucial.*" Our interviewees believe that having a test suite for the core generic features of the app is the best approach in the long term. P12 said: "*Unit tests are still the best. They are easy to run, and provide immediate feedback when you break something.*"

[8] http://code.google.com/p/robotium/

[9] http://developer.android.com/tools/help/monkeyrunner_concepts.html

[10] http://www.gorillalogic.com/testing-tools/monkeytalk

[11] https://github.com/square/KIF

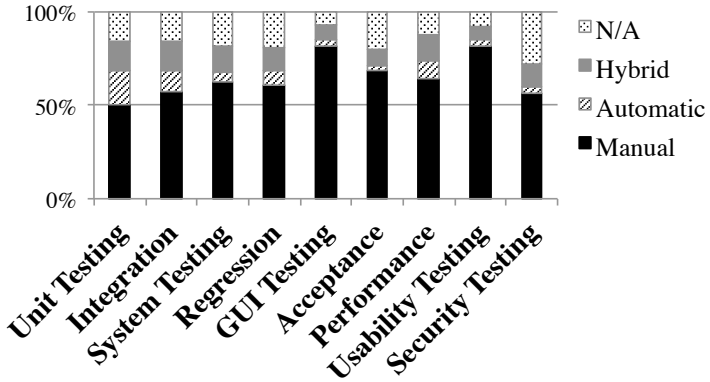


Figure 2.15: iOS levels of testing.

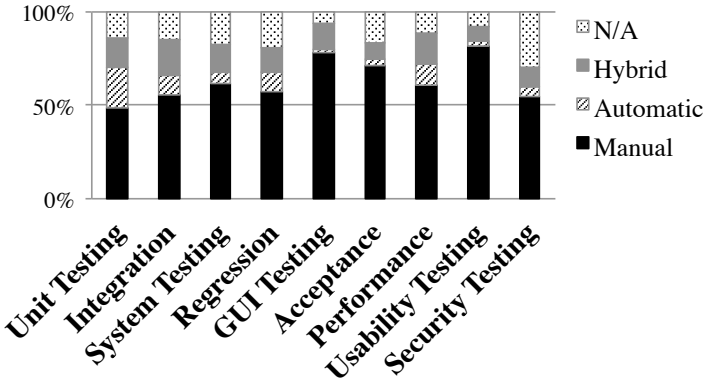


Figure 2.16: Android levels of testing.

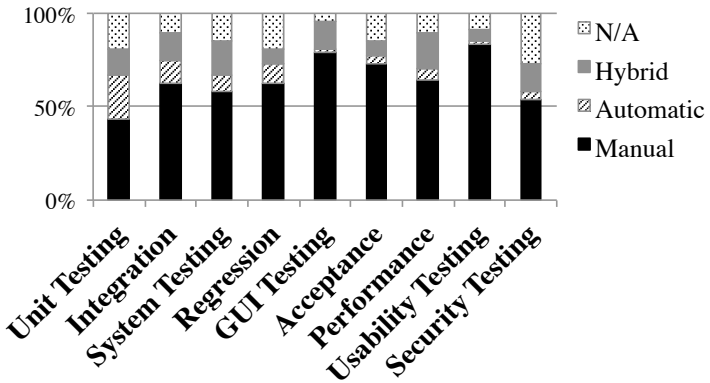


Figure 2.17: Windows levels of testing.

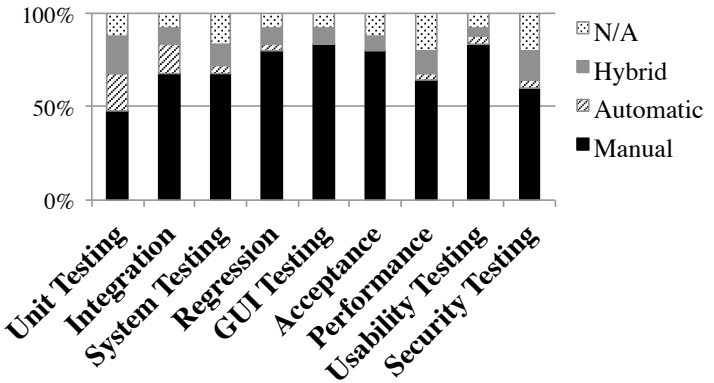


Figure 2.18: Blackberry levels of testing.

Unit testing seems to be more popular among Android and Windows developers, using `JUnit` and `NUnit`, respectively.

Two iOS participants have tried writing unit tests for iPhone using XCODE INSTRUMENTS[12] as well as SENTESTINGKITFRAMEWORK, a built-in Xcode tool. P1 explained: "*iOS apps are not really built to be fully unit tested. You have to structure your code properly in order to actually write good unit tests. It is hard to test our apps because a lot of view manipulation logic and business logic are mixed in the controllers and it is hard to write unit tests for controllers. This is one of the MVC's [Model View Controller] shortcomings that could discourage developers from writing unit tests. Testing models are easier with unit-test and a better way to test UI is to write integration/acceptance tests using e.g.,* KIF *or* CALABASH.[13]" P12 argued: "*iOS doesn't make it easy to have test automation*" and a respondent said: "*Apple's Developer testing tools don't play well.*"

Usability Testing

While there is not a common agreement on usability standards for all platforms and "*usability experts have not agreed on [mobile app] standards like [they have for] the web*", our participants acknowledged that "*usability is the most important factor for testing on these devices, because that is what people care about.*" P10 stated, "*In the past we have controlled the dialog and the image and the distribution on the web but now the store providers, like Apple and Microsoft have. They control whether we can submit or not. We have to follow their rules, and people have a huge social platform to rank us on the store and it is much more front centre than it was before.*" End-users nowadays have the ability to collectively rank apps on the mobile app stores. If users like an app, they download and start using it. If not, they delete it and move on immediately. If they really like it, they rank it high; Or if they really dislike it, they go on social media and complain. Thus, a low-quality release can have devastating consequences for mobile developers. As a result, there is a huge emphasis on usability testing. As P8 explained, "*definitely one of our big challenges is usability testing, which is manual. I do heuristic eval-*

[12] https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html

[13] http://calaba.sh/

uations personally and then evaluate with real users." P11 elaborated: "*Our usability testing encompasses a large portion of not only features but also UI. Within the application, we got a community of testers that are willing to test our newest and greatest software parts and put some feedback on it.*" Or a respondent said: "*You need to spend a good amount of time to fix the UI, usability and performance issues for each platform.*" Additionally, the next aspect is emotion, explained by a participant: "*the personal attachment of people to their devices cannot be simulated in a test environment. Users take these devices into bed, bathroom, doctor's office. So in very intimate parts of their lives, they are using these apps and they have an emotion attached to that.*"

Security Testing

As observed by another study for Android apps [63], security bug reports are of higher quality but get fixed slower than non-security bugs. While there has been a number of studies related to security, privacy leaks and malware behaviour of mobile apps [48, 67, 84, 189, 207], in practice, as it is shown in Figure 2.14, security testing has the least priority (i.e., N/A) among mobile app development community. P10 stated that "*I don't do security, but everything else I do. There is very little tool supports to help with this.*" However, P1 explained that it depends on the extensive of testing, for example if the app is for a major enterprise client that security is a top priority, they need more testing than normal apps. He continued: "*We have got security audit from various security companies that actually have done security testing on our apps. But we don't do anything internally.*"

Performance Testing

The more critical the app is, the more performance testing is conducted. As shown in Figure 2.14, performance testing has been performed mostly manually but some of our participants have used different types of tools. P5 stated that "*Because of the nature of our apps which are on goggles, we have to be very critical about performance and battery consumption. So we do a lot of related testing and measure the currency of battery usage.*" P7 added: "*A lot of apps, such as games, require performance, if you develop some middleware to drag down the user experiences,*

nobody will use it because in games user experience is the most important feature."

Beta Testers and Third Party Testing Services

Beta testing, such as TESTFLIGHT[14], seems to be quite popular in mobile app development; although P5 emphasized that "*the beta testers are in the order of dozens not thousands.*" TestFlight automates parts of the process, from deploying the app to collecting feedback. Further, there are many cases in which the clients are responsible for testing, i.e., recruiting beta testers or acceptance testing. P6 explained that they have internal and external client tracking systems: "*Basically we have two bug tracking systems, internal and client tracking system (external). The clients create bugs in that tracking system and our testing team try to reproduce bugs to see if it is a valid and reproducible bug. If so they duplicate it in our internal tracking system. Then developers will look at it again.*"

Additionally, some developers rely on third party testing services such as PERFECTOMOBILE[15] and DEVICEANYWHERE.[16] However, "*it is usually too volatile and the tools in many cases support very simple apps. Honestly not really worth the effort*", said one of our interviewees. Other participants' attitudes toward testing services are varied; P12 argued: "*Services should be affordable, and not just report bugs but also provide some documents that indicate how people test the application, and give a high-level overview of all the paths and possibilities that are tested.*" Another respondent said: "*Most online testing services charge a very hefty premium even for apps that are distributed for free*"; and: "*It is nice to test an app by a third party, someone who is not the developer. At the same time, just random testing doesn't do the trick. You need to have a more methodical approach, but the problem with methodical approaches is that they turn the price up.*" P11 said: "*We don't want to lock in on one specific vendor and tend to use open-source tools, such as JUnit.*" Another problem mentioned is that "*if we want to change something the way we want to, we don't have access to the source code. So we can't change the services of the framework.*"

[14] https://developer.apple.com/testflight/update/

[15] http://www.perfectomobile.com/

[16] http://www.keynote.com/solutions/testing/mobile-testing

Handling User Workflow Interruption

Related to the usability and multi-screens, some of our participants stated that many users go through a workflow using multiple devices. For instance, P10 explained that a user may search for a flight on her smart phone and find a good deal, but to book the flight requires much typing. So she goes on her laptop or tablet and books the flight from there, then she goes back to the smartphone to save the e-ticket. Thus, long workflows tend to make users swap between devices, and apps should be able to handle such interruptions.

2.3.4 Analysis and Testing Challenges

In this subsection, we present the challenges experienced, by our interview participants and survey respondents, for analyzing and testing native mobile apps.

Limited Unit Testing Support for Mobile Specific Features

Although `JUnit` is used by more than half of the Android participants, many also point out that "*JUnit is designed for stationary applications and it has no interface with mobile specifics such as sensors (GPS, accelerometer, gyroscope), rotation, navigation*". As a result, "*there is no simple way to inject GPS positions, to rotate the device and verify it that way*". P11 explained: "*we are creating a 'map application', which requires users typically being outdoors, moving around and navigating, which is not supported by current testing tools.*" Writing mobile specific test scenarios requires a lot of codes and is time-consuming and challenging. A number of participants indicated that having "*a JUnit type of framework with mobile specific APIs and assertions*" would be very helpful.

Monitoring and Analysis

Both our interview and survey data indicate a strong need of mobile app developers for better analysis and monitoring support. Many mentioned the need to monitor, measure, and visualize various metrics of their apps such as memory management (to spot memory leaks), battery usage (to optimize battery life), CPU usage, pulling/pushing data, and network performance (over various networks, e.g., 2G, 3G, 4G and wireless connections) through better analysis tools. "*A visualization*

tool such as those hospital monitoring devices with heart rate, blood pressure, etc., would help to gain a better understanding of an app's health and performance", explained P8.

Handling Crashes

One major problem mentioned in mobile app testing is about crashes, which are often intermittent, non-deterministic, and irrecoverable. It is challenging for developers to capture enough information about these crashes to analyze and reproduce them [220] so that they can be fixed. Many developers in our study found it helpful to have a set of tools that would enable capturing state data as a crash occurs and creating a bug report automatically. P5 stated: "*Dealing with the crashes that are very hard to catch and harder to reproduce is an issue. It would be good that when the crashes happen, system logs and crash logs can be immediately captured and sent to developers over the phone.*"

Emulators/Simulators

Emulators are known to mimic the software and hardware environments found on actual devices whereas simulators only mimic the software environment. Many mobile developers believe that better support is needed to mimic real environments (e.g., network latency, sensors) for testing. Another issue mentioned is that *rooted* simulators and emulators are needed in order to access features outside of the application, such as settings, play store, Bluetooth and GPS, which could be part of a test case. Also, the performance of emulators is a key factor mentioned by many of our participants. Compared to iOS Simulator, "*Android emulator is very slow. I use my device for testing instead*", said P8.

Missing Platform-Supported Tools

Almost all of the participants mentioned that current tools are weak and unreliable with limited support for important features for mobile testing such as mobility, location services, sensors and different inputs. They have experienced many automation failures or many cases where testing tools actually slowed the development process down substantially.

Some of our participants stated that platform-supported tools are needed, e.g., "*unit testing should be built-in*". A respondent said: "*the platforms have to support it (testing). 3rd party solutions will never be good enough.*", and another one said they need "*strong integrated development environment support*". Some noted that the process will be similar to that for web applications, "*it took years to create powerful tools for analyzing and testing web apps, and we are still not there completely.*"

Rapid Changes Over Time

Our interview reveals that requirements for mobile app projects change rapidly and very often over time. This is the reason our participants argued that they have difficulties to keep the testing code up to date. A respondent said: "*changing requirements means changing UI/logic, so GUI and integration tests must be constantly rewritten.*" P1 stated: "*there are some testing tools out there, but we don't use any of them because we can't keep the tests updated for our dynamic apps.*" P10 stated that due to rapid changes, they have "*time constraints for creating test scripts and performing proper testing*".

Testing Device in the Wild with Many Possibilities to Check

An issue mentioned by some of the participants is the fact that combination of parameters in the wild is challenging, but it is best to test the app where the users are actually using it. P10 stated: "*Weather condition has an effect on wireless activity and the visual representation of app.*" Our participants explained that there are so many different possibilities to test, and places that could go potentially wrong on mobile apps. Thus, "*it is difficult to identify all the usage scenarios and possible use cases while there is a lot of hidden states; for example, enabling/disabling the location services, and weak and strong network for network connectivity*". P12 finds: "*The relation between apps should be well managed, you might be interrupting other apps, or they might be interrupting yours.*" P12 provides an example: "*manage the states when an audio recording app goes into background.*" Furthermore, a participant argued that based on missing or misleading usage specifications, they should avoid under-coverage (missing test cases) and over-coverage

(waste of time and human resources for testing situations that won't happen in the real world). Another related issue to take care includes upgrading as P12 explained: "*For example, when upgrading from iOS 5 to iOS 6, the permissions are different. So your app in the wild just stop working. Unfortunately, there is nothing that can help you figure those out.*"

App Stores' Requirements

Developers have to follow mobile app stores' (e.g., Apple's AppStore, Android Google Play, Windows Marketplace and Blackberry AppWorld) requirements to distribute their apps to end users. These requirements change often, and developers need a way to test their apps' conformance. "*I would like to have something more robust for me to mimic what the publisher (store) will be doing so that I can catch the error earlier in the development process,*" said a respondent. Additionally, "*pushing the app to individual devices is more complex than necessary*", for instance in iPhone.

2.4 What Has (not) Changed since 2012? A Follow-up Study

Three years after our initial qualitative study, we felt it was necessary to find out what has changed and what has remained the same in the app development spectrum of challenges and practices.

2.4.1 Survey Design

This follow-up study was conducted during the months of February and March of 2015, by surveying mobile app experts. First, we created a document outlining the main findings of our results from the initial study with each finding has space for an optional comment. We also included an open-ended question asking whether the existing mobile app development challenges have changed or whether new challenges have emerged since 2012.[17] The goal of the extended study is not to perform a whole new study. We aimed to target, particularly, our original inter-

[17] http://goo.gl/forms/kp98G1ldpY

viewees (the earlier 12 experts) again for a follow-up and see what has changed. Thus, we sent an email providing a link to this survey directly to 25 experts in our network, including all the interviewees in our earlier study as well as new known experts in our network. Similar to the original study, we also shared this link to the popular Mobile Development Meetup and LinkedIn groups related to native mobile development. However given the nature of this survey, all of our respondents are from the emails we directly contacted.

2.4.2 Our Participants

We received 15 responses, from which four were from our original pool of interviewees. The 15 respondents were native iOS or Android developers in Canada, and they are from 12 different companies. Additionally, they have an average of five years of app development experience.

2.4.3 Analysis and Summary of Survey Findings

The survey results reveal interesting findings; they indicate that overall, the list of the challenges from our initial study is still valid and that there are some new or changed challenges, which we discuss next.

Moving toward Fragmentation rather than Unification

External fragmentation, i.e., fragmentation across platforms, seems to have decreased since 2012; as a respondent explained: "*the fragmentation [across platforms] is definitely less visible than before. Before, there had to be implementations for Symbian, Blackberry, Android, iOS, and Windows Phone, but this has changed; Android and iOS are the front runners. So by choosing [these] two, a good percentage of users can already be covered.*"

The challenge with internal fragmentation, i.e., fragmentation within the same platform, used to mainly affect Android developers because of the many hardware variations in devices. However, Apple is also moving into that direction. As a result, currently there is also internal fragmentation with Apple as it has released new devices with different screen sizes (such as iPhone 5, 6 and 6+) resulting in more variations. A respondent stated: "*They [Apple] have released updated APIs*

that make supporting the new screen sizes much easier, but it requires using the latest OS version (8.0), which is difficult if you have a legacy codebase."

Open/Closed Development Platforms

Our respondents indicated that mobile development tools have matured significantly and are more stable in the last few years. Apple and Google have released enhanced IDEs (e.g., XCode 6, Android Studio) that makes development more efficient. Furthermore, app development has become more popular and it is easier to find skilled developers as the pool of developers has grown.

In terms of Android being open-source, the fact that any manufacturer has the ability to customize their version of Android and modify the source is still a major issue on Android. Our respondents explained that manufacturers introduce variations on the underlying implementations of the OS that has the potential of breaking the contract of APIs. When this happens, apps that work on stock or near-stock versions of the OS perform as expected, but the custom versions have unexpected anomalies on the UI and also hardware components, such as the camera. This results in software fragmentation that is most difficult to deal with for app developers.

On the other hand, as explained by a respondent, "*as an Android developer, I have been blessed with two entities; the first being Google engineers and the second being the open-source community. Both these entities have contributed immensely to lower the barriers of mobile app development with tools, libraries and documentation.*" Another android developer agreed that "*More open-source libraries are now available that are easy to integrate into your mobile app providing robust functionality and interesting features. This helps prevent reinventing the wheel as there are more resources for developers to lean on.*" However, it was also mentioned that while developers would prefer open-source software, most of the apps out there are still closed-source.

Web vs. Hybrid vs. Native Mobile Apps

In terms of hybrid solutions and cross-platform tools that promise *write-once, run everywhere*, the respondents stated that they are still not mature enough to be used in production code. Currently, there seems to be more interest in adopting the hy-

brid approach. A respondent stated, "*the trend I have seen recently is that people start with a hybrid implementation with minimal functionality, and as they start gaining traction, they change to a native implementation for performance and better user experience.*" Another respondent stated, "*there is still quite a large technical debate, especially for start-ups, about whether to create native apps (specifically iOS and Android) versus just creating a responsive mobile web experience. This is mostly determined by [the availability of] funds and resources to create, test, monitor, handle customer support issues, and maintain additional software applications.*"

Although our respondents agreed that mobile browsers are becoming more mature to support mobile web apps, they also raised concerns; for instance one respondent added: "*user expectations about the platform have also risen and some UI effects (for example the extensive use of transparency in iOS 7+) are difficult to do non-natively. So non-native apps are trying to hit a moving target, and there is no reason to think that the target will stop moving.*"

Also when it comes to more complex apps, hybrid apps cannot fully support everything that is needed; "*we have stayed away from hybrid also because it limits the app performance and list of available APIs to use*", a respondent stated. Another major problem in the hybrid approach, mentioned by our respondents, is debugging: "*it is difficult to debug code when developing a hybrid model as the issue could be in the native or web layer. Also attaching a debugger to the web-layer is slow and makes it extremely difficult to use.*"

Monitoring and Analysis

All of the respondents still mentioned that monitoring and analysis challenges are further increased today. While mobile app crash reporting tools such as CRASHLYTICS[18], Google Analytics[19] for mobile apps, and NEWRELIC[20] have evolved over time and a few profiling/monitoring tools embedded in the IDEs, our respondents believed that "*analytics, logging, crash-reporting, etc. are general requirements for any app development and should be available as a framework uniform*

[18] https://try.crashlytics.com/
[19] http://www.google.com/analytics/mobile/
[20] http://newrelic.com/

across all platforms and controllable via configurations. At this moment, each of these is being addressed via certain service providers in a variety of ways."

Levels of Testing

There is more awareness among developers related to the benefits of testing and more testing tools are emerging to help app developers. Most of our respondents agreed that automated testing tools have improved in the last three years. So has the prevalence of unit and GUI testing in practice. An android developer stated, "*for Unit testing and functional testing, Google and the open-source community have come together once again to provide a toolset that allows developers to really provide unit testing and proper UI testing. Tools like* ROBOLECTRIC[21] *and* ESPRESSO[22] *for Android have brought unit testing and functional testing to a much higher level. But, it is still not easy to write the unit tests, mainly because of all the deep native APIs that are difficult to mock.*" Another respondent said, "*with platforms like* GENYMOTION[23] *allowing to run the same application across multiple devices very quickly and efficiently and then integration test platforms like Jenkins, it has become easier to run the app across many different platforms.*"

Keeping Up with Frequent Changes

It is still challenging to keep up with the new releases, especially with Apple breaking core functionality in new releases. A respondent stated, "*it is getting more difficult to develop for iOS now and guidelines are scattered when it comes to App Store submissions. Also, Apple has made this worse by discontinuing tools, buying up others and breaking them, such as* TESTFLIGHT."

In terms of keeping up to date for devices, an Android developer explained that "*before Google and the open-source community (OSC) really picked up their pace in Android 4.0, backwards compatibility was extremely difficult. Many devices never got upgraded to newer versions. However, Google and the OSC found ways to decouple some of their core APIs into libraries that brought backwards compatibility all the way back to Android 2.0.*"

[21] http://robolectric.org/
[22] https://code.google.com/p/android-test-kit/wiki/Espresso
[23] https://www.genymotion.com/

Data Intensive Apps

Regarding dealing with data-intensive apps, a respondent explained: "*syncing with a back-end is challenging but there are third-party tools such as* PARSE[24] *that solve this problem for some use-cases.*"

Apart from the aforementioned updates, our respondents agreed that the rest of our initial findings remains the same: "*The rest is pretty much the same as it used to be and I think your paper has captured it reasonably well*", a respondent stated.

2.5 Threats to Validity

Similar to quantitative research, qualitative studies could suffer from threats to validity, which is challenging to assess as outlined by Onwuegbuzie *et al.* [179].

For instance, in codification, the researcher bias can be troublesome, skewing results on data analysis [132]. We tried to mitigate this threat through triangulation; The codification process was conducted by two researchers, one of whom had not participated in the interviews, to ensure minimal interference of personal opinions or individual preferences. Additionally, we conducted a survey to challenge the results emerging from the interviews.

Both the interview and survey questionnaire were designed by a group of three researchers, with feedback from four external people – one senior Ph.D. student and three industrial mobile app developers – in order to ensure that all the questions were appropriate and easily comprehensible.

Another concern was a degree of generalizability. We tried to draw representative mobile developer samples from nine different companies. Thus, the distribution of participants includes different companies, development team sizes, platforms, application domains, and programming languages – representing a wide range of potential participants. Of course, the participants in the survey also have a wide range of background and expertise. All this gives us some confidence that the results have a degree of generalizability.

One risk within Grounded Theory is that the resulting findings might not fit with the data or the participants [99]. To mitigate this risk, we challenged the findings from the interviews with an online survey, filled out by 188 practitioners

[24] https://parse.com/products/core

worldwide. The results of the survey confirmed that the main concepts and codes, generated by the Grounded Theory approach, are in line with what the majority of the mobile development community believes.

Lastly, in order to make sure that the right participants would take part in the survey, we shared the survey link with some of the popular Mobile Development Meetup and LinkedIn groups related to native mobile app development. Furthermore, we did not offer any financial incentives nor any special bonuses or prizes to increase response rate.

2.6 Discussion

We discuss the challenges that are worth further investigation by the research and development community.

2.6.1 Mapping Study

We complement our quantitative and qualitative analysis with a mapping study in order to show how the research and industry community is investigating some of these challenges. Table 2.2 presents a mapping study between the research community and our challenges, listed into *Analysis and Testing Studies* and *Multiple Platforms Studies*. Among analysis and testing studies are model-based approaches, record-and-replay approaches, context-sensitive events, mobile security and privacy leaks, and performance profiling. Among multiple platforms studies are device fragmentation, cross-compilation approaches, and mappings and consistency checking. Additionally, there exists a body of other challenges that are recognized by researchers, listed under *Other Studies* in Table 2.2. Among them are studies related to mobile energy efficiency, app bytecode, the impact of unstable or buggy APIs, the impact of mobile ads, management of informative user reviews, management of bug reports, and technology selection frameworks. We discussed most of them in this section and the related work section.

Table 2.2: A mapping study.

Analysis and Testing Studies:

- Model-based Approach [49, 73, 85, 119, 128, 153, 154, 217]
- Record-and-Replay Approach [102]
- Context-Sensitive Approach [43, 66, 120, 146, 216]
- Mobile Security and Privacy Leaks [47, 54, 67, 84, 196, 218, 221]
- Performance Profiling [142, 151, 177]

Multiple Platforms Studies:

- Android Fragmentation [112, 135]
- Cross-Compilation Approach* [101, 117, 171, 187]
- Mappings and Consistency Checking [88, 100]

Other Studies:

- Mobile Energy Efficiency [57, 60, 176, 188]
- Mobile App Bytecode [194, 205]
- Impact of Unstable/Buggy APIs* [58, 147, 158, 195]
- Impact of Mobile Ads [107, 167, 173, 174]
- Informative User Reviews Management* [71, 105, 110, 123, 152, 181, 184]
- Bug Reports Management* [63, 112, 155, 168]
- Technology Selection Frameworks [157]

* Discussed in Section 2.7.

2.6.2 Same App across Multiple Platforms

Development

A prominent challenge emerging from our study is the fact that developers have to build the same native app for multiple mobile platforms. Although developing for multiple platforms is a recurring problem that is not unique to the mobile world, the lack of proper development and analysis support in the mobile environment

exacerbates the challenges. Opting for standardized cross-platform solutions, such as HTML5, seems to be the way to move forward. However, HTML5 needs to be pushed towards maturation and adoption by major mobile manufacturers, which in turn can mitigate many of the cross-platform development problems. Another possible direction to pursue is exploring ways to declaratively construct [114] native mobile applications, by abstracting the implementation details into a model, which could be used to generate platform-specific instances. That being said, dealing with the critically of user experience in such apps and the dramatic differences among the platforms not just in terms of APIs but also in the types of interactions, are among the challenges.

Consistency Checking

Since each mobile platform requires its own unique environment in terms of programming languages, tools, and development teams, another related challenge is checking the correctness and consistency of the app developed across different platforms. As revealed by our findings, developers currently conduct manual screen-by-screen comparisons of the apps across platforms to check for consistent behaviour. However, this manual process is tedious and error-prone. One way to tackle this problem is by constructing tools and techniques that can automatically infer interaction models from the app on different platforms (See Table 2.2). In Chapter 5, we reverse engineer a model of iOS applications [85]. Similarly, others [119, 217] are looking into Android apps. The models of the app, generated from different platforms, can be formally compared for equivalence on a pairwise basis [162] to expose any detected discrepancies. In Chapter 6, we propose an automated technique for detecting inconsistencies in the same native app implemented in iOS and Android platforms [88]. Other studies could use image-processing techniques in the mapping phase. Additionally, they could focus on capturing information regarding the API calls made to utilize the device's native functionality such as GPS, SMS, Address Book, E-mail, Calendar, Camera, and Gallery, as well as device's network communication i.e., client-server communication of platform-specific versions of a mobile app (similar to the cross-platform feature matching of web applications [74]). Such automated techniques would drastically minimize the

difficulty and effort in consistency checking since many mobile developers manually "*do screen-by-screen design review for each new platform*".

Testing

Regarding the testing challenges, follow-up studies could focus on generating test cases for mobile apps. A centralized automatic testing system that generates a (different) test case for each target platform could be a huge benefit. While platform-specific features can be customized, core features could share the same tests. Thus, further research should focus on streamlining application development and testing efforts regardless of the mobile platform.

2.6.3 Testing Mobile-Specific Features

The existing testing frameworks have limitations for testing mobile-specific features and scenarios such as sensors (GPS, Accelerometer, gyroscope), rotation, navigation, and mobility (changing network connectivity). As a consequence developers either need to write much test fixture code to assert mobile-specific scenarios or opt for manual testing. Thus, creating "*a* `JUnit` *type of framework with mobile-specific APIs and assertions*" would be really beneficial. While there are open-source and commercial tools available in the market that help emulate contextual events e.g., Genymotion[25] or Lockito[26], our interviews mentioned that built-in and platform-supported tools are needed as third-party solutions are hard to be good enough.

Additionally, on the academic side as listed in Table 2.2, related studies [43, 66, 120, 146] proposed testing frameworks that consider not only GUI events but also contextual events. Liang *et al.* [146] present Caiipa, a cloud service for testing apps over an expanded mobile context space in a scalable way. It incorporates key techniques to make app testing more tractable, including a context test space prioritizer to quickly discover failure scenarios for each app. Chandra *et al.* [66] have developed techniques for scalable automated mobile app testing within two prototype services – VanarSena [190] and Caiipa [146]. In their paper, they describe a

[25] https://www.genymotion.com/

[26] https://play.google.com/store/apps/details?id=fr.dvilleneuve.lockito

vision for SMASH, a cloud-based mobile app testing service that combines both previous systems to tackle the complexities presently faced by testers of mobile apps.

2.6.4 Other Challenging Areas

There are also serious needs for (1) rooted emulators that can mimic the hardware and software environments realistically; (2) better analysis tools, in order to measure and monitor different metrics of the app under development; (3) techniques that would help debugging apps by capturing better state data when unexpected crashes occur; (4) testing APIs from app stores, in order to catch the inconsistencies of code with a store's guidelines and internal APIs. In particular for Apple app store, it would be beneficial if a set of testing APIs (e.g., as services) could check the code against, before submitting to the stores.

2.7 Related Work

Researchers have discussed some of the challenges involved in mobile app development [46, 82, 92, 138, 166, 169, 206, 213], however, most of these discussions are anecdotal in nature. Other recent studies have made an effort to obtain better insights regarding issues and concerns in mobile development through (1) mining question and answer (QA) websites [62, 148, 149, 212] that are used by developers; (2) mining and analyzing app stores' content, such as user-reviews [68, 71, 95, 113, 134–136, 140, 147, 152, 156, 181, 184], mobile app attributes and descriptions [105, 143, 197], mobile app bytecode [54, 194, 195, 218], and (3) mining mobile bug repositories [63, 112, 155, 168]. We categorize related work into the aforementioned classes as well as cross-platform app development studies and grounded theory studies in software engineering. We also provide a review of the current papers and their relationship with the challenges in mobile development that are widely recognized by the researchers, such as proliferation of malware via fake markets or apps highjacking, crowdsourced requirements, the management of non-informative reviews/bug reports from the crowd, and impact of unstable/buggy APIs.

Mobile App Development and Testing Challenges

Recently, there have been numerous studies [46, 82, 92, 138, 150, 166, 169, 213] related to the development and testing of mobile apps. Kochhar *et al.* [138] discussed the test automation culture among app developers. They surveyed 83 Android and 127 Windows app developers and found that time constraints, compatibility issues, lack of exposure, and cumbersome tools are the main challenges. Miranda *et al.* [166] reported on an exploratory study through semi-structured interviews with nine mobile developers. They found that developers perceive the Android platform as more accessible and compatible with their existing knowledge, however, its fragmentation is the major problem. Additionally, some developers choose iOS because sales are more profitable on that platform. Muccini *et al.* [169] briefly discussed challenges and research directions on testing mobile apps by analyzing the state of the art. Performance, security, reliability, and energy are strongly affected by the variability of the environment where the mobile device moves towards. Dehlinger *et al.* [82] briefly described four challenges they see for mobile app software engineering and possible research directions. These challenges are namely, creating user interfaces accessible to differently-abled users, developing for mobile application product-lines, supporting context-aware applications, and specifying requirements uncertainty. Franke *et al.* [92] have shown that life-cycles of mobile platforms (iOS, Android, Java ME) have issues with the official lifecycle models. They presented a way to reverse engineer any mobile app lifecycle. They found for each platform either errors in the official models, inconsistencies in the documentation or a lack of information in both. Wasserman [213] briefly discussed a number of mobile-related research topics including development processes, tools, user interface design, application portability, quality, and security.

Mining QA Websites

Beyer *et al.* [62] presented a manual categorization of 450 Android related posts of *StackOverflow* concerning their question and problem types using the input of three Android app developers. The study highlights that developers have problems with the usage of API components, such as user interface and core elements. Errors

are mentioned in questions related to Network, Database, and Fragments. Linares-Vasquez *et al.* [148] used topic modelling techniques to extract hot topics from *StackOverflow* mobile-development related questions. Their findings suggest that most of the questions include general topics such as IDE-related and compatibility issues, while the specific topics, such as crash reports and database connection, are presented in a reduced set of questions. In another study, Linares-Vasquez *et al.* [149] investigated how changes occurring to Android APIs trigger questions and activity in *StackOverow*. They found that Android developers have more questions when the behaviour of APIs is modified e.g., deleting public methods from APIs is a trigger for questions. Wang *et al.* [212] analyzed API-related posts regarding iOS and Android development from *StackOverflow* to understand API usage challenges based on forum-based input from a multitude of developers. Bajaj *et al.* [56] mined *StackOverflow* for questions and answers related to mobile web apps. They found that web-related discussions are becoming more prevalent in mobile development, and developers face implementation issues with new HTML5 features such as Canvas.

Mining App Stores

Khalid [134] manually analyzed and tagged reviews of iOS apps to identify the different issues that users of iOS apps complain about. They [136] studied 6,390 low star-rating user-reviews for 20 free iOS apps and uncovered 12 types of complaints. Their findings suggest that functional errors, feature requests and app crashes are the most frequent complaints while privacy and ethical issues, and hidden app costs are the complaints with the most negative impact on app ratings. Gorla *et al.* [105] clustered Android apps by their description topics to identify potentially malicious outliers in terms of API usage. Their CHABADA prototype identified several anomalies in a set of 22K Android apps. Avdiienko *et al.* [54] compared benign and malicious Android apps by mining their data flow from sensitive sources. They found that the data for sensitive sources ends up in typical sinks that differ between benign and malicious apps. Khalid *et al.* [135] helped game app developers deal with Android fragmentation by picking the devices that have the most impact on their app ratings, and aiding developers in prioritizing their testing efforts. Mining

the user reviews of 99 free game apps, they found that although apps receive user reviews from 38-132 unique devices, 80% of the reviews originate from a small subset of devices. Pagano *et al.* [181] carried out an exploratory study on over one million reviews from iOS apps to determine their potential for requirements engineering processes. They found that most of the feedback is provided shortly after new releases, and the content has an impact on download numbers. They also found that reviews' topics include user experience, bug reports, and feature requests. Linares-Vasquez *et al.* [147] investigated how the fault and change-proneness of APIs used by free Android apps relates to their success estimated as the average rating provided by the users. They [58] also surveyed 45 Android developers to indicate that apps having high user ratings use APIs that are less fault- and change-prone than the APIs used by low rated apps. As also revealed by our study, McDonnell *et al.* [158] found that rapid platform/library/API evolution is among the challenges mobile developers and testers are faced with.

Mining Bug Repositories

Han *et al.* [112] analyzed fragmentation within Android by extracting topics from bug reports of HTC and Motorola, using topic modelling techniques. They found that hardware-based fragmentation affecting the bugs reported in the Android bug repository as even for shared common topics there was a divergence in topic keywords between vendors. Martie *et al.* [155] presented an approach to examine the topics of concern for the Android open-source projects using issue trackers. They used LDA to examine Android bug XML logs and analyzed topic trends and distribution over time and releases.

Cross-platform App Development

There have been a number of comparison studies [81, 121, 180, 182] of several "write once run anywhere" tools (e.g., PHONEGAP, APPCELERATOR TITANIUM, XAMARIN, etc.). Other studies [83, 101, 129, 157] have analyzed different web-based or hybrid mobile app development frameworks, while others [187] have discussed cross-compilation approach. For instance, Palmieri *et al.* [182] report a comparison between four different cross-platform tools (RHODES, PHONEGAP,

DRAGONRAD and MOSYNC) to develop applications on different mobile OSs. Huy *et al.* [121] studied and analyzed four types of mobile applications, namely, native, mobile widgets, mobile web, and `HTML5`. Masi *et al.* [157] proposed a framework to support developers with their technology selection process for the development of a mobile application, which fits the given context and requirements. Gokhale *et al.* [101] discussed an approach for developing and delivering existing web and desktop applications as mobile apps. Their proposal is a variant of Hybrid development model that utilizes code translators to translate existing web or desktop applications for the target mobile platforms. Puder *et al.* [187] described a cross-compilation approach, where Android apps are cross-compiled to `C` for iOS and to `C#` for Windows Phone 7, from byte code level to API mapping.

Grounded Theory Studies in Software Engineering

Many researchers have used a grounded theory approach in qualitative software engineering studies [44, 45, 70, 77, 78, 89, 90, 106, 130, 132, 132, 133, 139, 175, 191, 203, 204, 214] in order to understand software development practices and challenges of industrial practitioners [44]. For instance, Khadka *et al.* [133] described an exploratory study where 26 industrial practitioners were interviewed on what makes a software system a legacy system, what the main drivers are that lead to the modernization of such systems, and what challenges are faced during the modernization process. The findings were validated through a survey with 198 respondents. Greiler *et al.* [106] conducted a grounded theory study to understand the challenges involved in Eclipse plug-in testing. The outcome of their interviews with 25 senior practitioners and a structured survey of 150 professionals provides an overview of the current testing practices, a set of barriers to adopting test practices, and the compensation strategies adopted because of limited testing by the Eclipse community. Based on their findings, they proposed a set of recommendations and areas for future research on plug-in based systems testing. Through a grounded theory approach, Sulayman *et al.* [204] performed interviews with 21 participants representing 11 different companies, and analyze the data qualitatively. They propose an initial framework of key software process improvement success factors for small and medium Web companies. Kasurinen *et al.* [131] discussed

the limitations, difficulties, and improvement needs in software test automation for different types of organizations. They surveyed employees from 31 software development organizations and qualitatively analyzed 12 companies as individual cases. They found that 74% of surveyed organizations do not use test automation consistently. Karhu *et al.* [130] explored the factors that affect the use of software testing automation through a case study within 5 different organizations. They collected data from interviews with managers, testers, and developers and used grounded theory. They found that the generic and independent (of third-party systems) tested products emphasize on the wide use of testing automation. Coleman *et al.* [77, 78] adopt the grounded theory methodology to report on the results of their study of how software processes are applied in the Irish software industry. The outcome is a theory that explains when and why software process improvement is undertaken by software developers.

Our study aims at understanding the actual challenges mobile developers face by interviewing and surveying developers in the field. To the best of our knowledge, our work is the first to report a qualitative field study targeting mobile app development practices and challenges.

2.8 Conclusions

Our study has given us a better, more objective understanding of the real challenges faced by the mobile app developers today, beyond anecdotal stories.

Our results reveal that having to deal with multiple mobile platforms is one of the most challenging aspects of mobile development. In particular, more recently the challenge with internal fragmentation within the same platform is significant. Since mobile devices and platforms are moving toward fragmentation, the development process cannot leverage information and knowledge from a platform to another platform. When the 'same' app is developed for multiple platforms, developers currently treat the mobile app for each platform separately and manually check that the functionality is preserved across multiple platforms and devices. Also creating a reusable user-interface design for the app is a trade-off between consistency and adhering to each platform's standards. Our study also shows that mobile developers need mainly platform-supported analysis tools to measure and

monitor their apps. Also, testing is a huge challenge currently. Most developers test their mobile apps manually. There are more awareness recently for Unit testing within the mobile community, however current testing frameworks do not provide the same level of support for different platforms. Additionally, most developers feel that current testing tools are weak and have limited support for important features of mobile testing such as mobility (e.g., changing network connectivity), location services, sensors, or different gestures and inputs. Finally, emulators seem to lack several real features of mobile devices, which makes analysis and testing, even more, challenging.

Chapter 3

Works For Me! Characterizing Non-reproducible Bug Reports

Summary[27]

Bug repository systems have become an integral component of software development activities. Ideally, each bug report should help developers to find and fix a software fault. However, there is a subset of reported bugs that is not (easily) reproducible, on which developers spend considerable amounts of time and effort. We present an empirical analysis of non-reproducible bug reports to characterize their rate, nature, and root causes. We mine one industrial and five open-source bug repositories, resulting in 32K non-reproducible bug reports. We (1) compare properties of non-reproducible reports with their counterparts such as active time and number of authors, (2) investigate their life-cycle patterns, and (3) examine 120 *Fixed* non-reproducible reports (i.e., non-reproducible reports that were marked as *Fixed* later in their life-cycle). In addition, we qualitatively classify a set of randomly selected non-reproducible bug reports (1,643) into six common categories. Our results show that, on average, non-reproducible bug reports pertain to 17% of all bug reports, remain active three months longer than their counterparts, can be mainly (45%) classified as "Interbug Dependencies", and 66% of *Fixed* non-reproducible reports were indeed reproduced and fixed.

[27]This chapter appeared at the 11th ACM Working Conference on Mining Software Repositories (MSR 2014) [87].

3.1 Introduction

When a failure is detected in a software system, a bug report is typically filed through a bug tracking system. The developers then try to validate, locate, and repair the reported bug as quickly as possible. In order to validate the existence of the bug, the first step developers take is often using the information in the bug report to *reproduce* the failure. However, reproducing reported bugs is not always straightforward. In fact, some reported bugs are difficult or impossible to reproduce. When all attempts at reproducing a reported bug are futile, the bug is marked as *non-reproducible (NR)* [7, 24].

Non-reproducible bugs are usually frustrating for developers to deal with [40]. First, developers usually spend a considerable amount of time trying to reproduce them, without any success. Second, due to the very nature of these bug reports, there is typically no coherent set of policies to follow when developers encounter such bug reports. Third, because they cannot be reproduced, developers are reluctant to take responsibility and close them.

Mistakenly marking an important bug as non-reproducible and ignoring it, can have serious consequences. An example is the recent security vulnerability found in Facebook [75], which allowed anyone to post to other users' walls. Before exposing the vulnerability, the person who had detected the vulnerability had filed a bug report. However, the bug was ignored by Facebook engineers: "*Unfortunately your report [...] did not have enough technical information for us to take action on it. We cannot respond to reports which do not contain enough detail to allow us to reproduce an issue.*"

Researchers have analyzed bug repositories from various perspectives including bug report quality [61], prediction [108], reassignment [109], bug fixing and code reviewing [53, 219], reopening [222], and misclassification [116]. None of these studies, however, has analyzed non-reproducible bugs in isolation. In fact, most studies have ignored non-reproducible bugs by focusing merely on the *Fixed* resolution.

In this work, we provide an empirical study on non-reproducible bug reports, characterizing their prevalence, nature, and root causes. We mine six bug repositories and employ a mixed-methods approach using both quantitative and qualitative

analysis. To the best of our knowledge, we are the first to study and characterize non-reproducible bug reports.

Overall, our work makes the following main contributions:

- We mine the bug repositories of one proprietary and five open source applications, comprising 188,319 bug reports in total; we extract 32,124 non-reproducible bugs and quantitatively compare them with other resolution types, using a set of metrics;
- We qualitatively analyze root causes of 1,643 non-reproducible bug reports to infer common categories of the reasons these reports cannot be reproduced. We systematically classify 1,643 non-reproducible bug reports into the inferred categories;
- We extract patterns of status and resolution changes pertaining to all the mined non-reproducible bug reports. Further, we manually investigate 120 of these non-reproducible reports that were marked as *Fixed* later in their life-cycle.

Our results show that, on average:

1. NR bug reports pertain to 17% of all bug reports;
2. compared with bug reports with other resolutions, NR bug reports remain active around *three months* longer, and are similar in terms of the extent to which they are discussed and/or the number of involved parties;
3. NR bug reports can be classified into 6 main cause categories, namely "Interbug Dependencies"' (45%), "Environmental Differences" (24%), "Insufficient Information" (14%), "Conflicting Expectations" (12%), and "Nondeterministic Behaviour" (3%);
4. 68% of all NR bug reports are resolved directly from the initial status (New / Open). The remaining 32% exhibit many resolution transition scenarios.
5. NR bug reports are seldom marked as *Fixed* (3%) later on; from those that are finally fixed, 66% are actually reproduced and fixed through code patches (i.e., changes in the source code).

3.2 Non-Reproducible Bugs

Most bug tracking systems are equipped with a default list of bug *statuses* and *resolutions*, which can be customized if needed. Generally, each bug report has a *status*, which specifies its current position in the bug report life cycle [7]. For instance, reports start at *New* and progress to *Resolved*. From *Resolved*, they are either *Reopened* or *Closed*, i.e., the issue is complete. At the *Resolved* status, there are different *resolutions* that a bug report can obtain, such as *Fixed*, *Duplicate*, *Won't Fix*, *Invalid*, or *Non-Reproducible* [7, 24].

There are various definitions available for non-reproducible bugs online. We adopt and slightly adapt the definition used in Bugzilla [24]:

Definition 1 *A* Non-Reproducible (NR) bug *is one that cannot be reproduced based on the information provided in the bug report. All attempts at reproducing the issue have been futile, and reading the system's code provides no clues as to why the described behaviour would occur.*

Other resolution terminologies commonly used for non-reproducible bugs include *Cannot Reproduce* [28], *Works on My Machine* [40] and *Works For Me* [41].

Our interest in studying NR bugs was triggered by realizing that developers spend considerable amounts of time and effort on these reports. For instance, issue #106396 in the ECLIPSE project has 62 comments from 28 people, discussing how to reproduce the reported bug [23]. This motivated us to conduct a systematic characterization study of non-reproducible bug reports to better understand their nature, frequency, and causes.

3.3 Methodology

Our analysis is based on a mixed-methods research approach [80], where we collect and analyze both quantitative as well as qualitative data. All our empirical data is available for download [9]. We address the following research questions in our study:

RQ1. How prevalent are NR bug reports? Are NR bug reports treated differently than other bug reports?

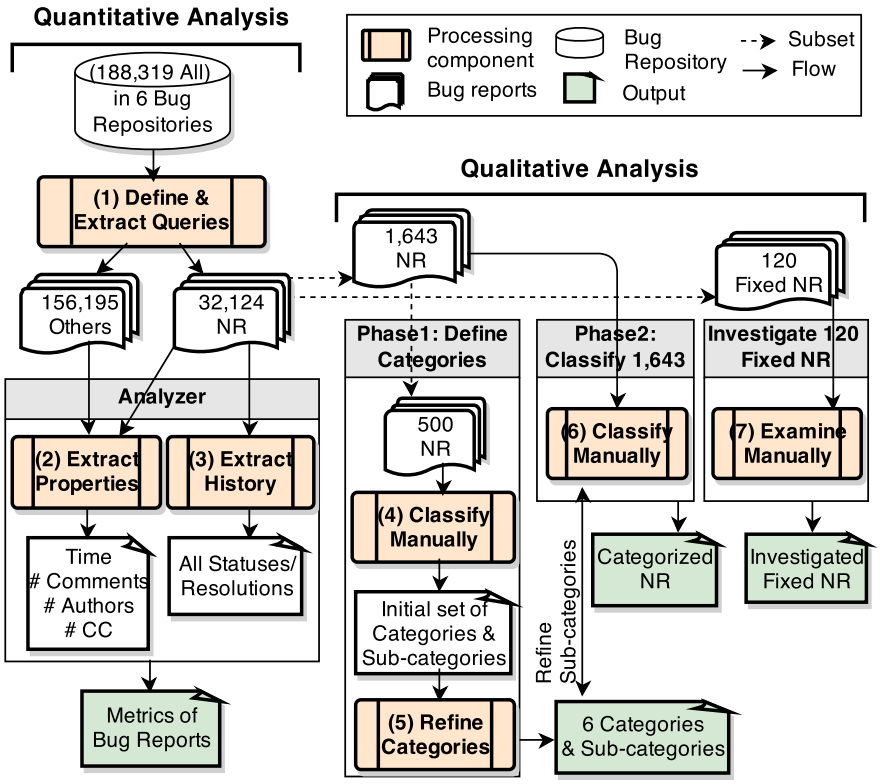


Figure 3.1: Overview of our methodology.

RQ2. Why can NR bug reports not be reproduced? What are the most common cause categories?

RQ3. Which resolution transition patterns are common in NR bug reports?

RQ4. What portion of NR bug reports is fixed eventually? Were they mislabelled initially? What cause categories do they belong to?

Figure 3.1 depicts our overall approach. We use this figure to illustrate our methodology throughout this section.

3.3.1 Bug Repository Selection

To answer our research questions, we need bug tracking systems that provide advanced search/filter mechanisms and access to historical bug report life-cycles.

Since BUGZILLA and JIRA both support these features (e.g., `Changed to/from` operators), we choose projects that use these two systems.

Table 3.1 shows the bug repositories we have selected for this study. To ensure representativeness, we select five popular, actively maintained software projects from three separate domains, namely *desktop* (FIREFOX and ECLIPSE IDE), *web* (MEDIAWIKI and MOODLE), and *mobile* (FIREFOX ANDROID). In addition, we include one commercial closed source application (INDUSTRIAL). The proprietary bug tracking system is from a Vancouver-based mobile app development company. The bug reports are filed by their testing team and end-users, and are related to different mobile platforms such as Android, Blackberry, iOS, and Windows Phone, as well as their content management platform and backend software.

3.3.2 Mining Non-Reproducible Bug Reports

In this study, we include all bug reports that are resolved as non-reproducible at least once in their life-cycles. In our search queries, we include all resolution terminologies commonly used for non-reproducible bug reports, as outlined in Section 3.2. We extract these NR bug reports in three main steps (Box 1 in Figure 3.1):

Step 1. We start by filtering out all *Invalid*, *Duplicate*, and *Rejected* reports. Where applicable, we also exclude *Enhancement*, *Feedback*, and *Unconfirmed* reports. The set of bug reports retrieved afterward is the total set that we consider in this study ('#All Bugs' in Table 3.1).

Step 2. We use the filter/search features available in the bug repository systems and apply the `Changed to/from` operator on the resolution field to narrow down the list of bug reports further to the non-reproducible resolution ('#NR Bugs' in Table 3.1).

Step 3. We extract and save the data in XML format, containing detailed information for each retrieved bug report.

Table 3.1: Studied bug repositories and their rate of NR bugs.

ID	Domain	Repository	Product/Component	#All Bugs*	#NR Bugs**	NR(%)	FixedNR(%)***
FF	Desktop	Bugzilla [4]	Firefox	65,408	18,516	**28%**	1%
E	Desktop	Bugzilla [2]	Eclipse/Platform	65,475	8,189	**13%**	4%
W	Web	Bugzilla [3]	MediaWiki	9,335	1,125	**12%**	9%
M	Web	Jira [10]	Moodle	22,175	2,503	**11%**	5%
FFA	Mobile	Bugzilla [4]	FirefoxAndroid	7,902	1,148	**15%**	3%
PTY	Mobile	Jira	Proprietary	18,024	643	**4%**	17%
Overall				188,319	32,124	**17%**	3%

*All_Query: Resolution: All except (*Duplicate*, *Invalid*, *Rejected*) and Severity: All except (*Enhancement*, *Feedback*) and Status: All except *Unconfirmed*
**NR_Query: All_Query and Resolution: `Changed to/from` *Non-Reproducible*
***FixedNR_Query: Resolution: *Fixed* and Severity: All except (*Enhancement*, *Feedback*) and Status: All except *Unconfirmed* and Resolution CHANGED FROM *Non-Reproducible* and Resolution: CHANGED TO *Fixed*

This mining step was conducted during August, 2013. We did not constrain the start date for any of the repositories. The detailed search queries used in our study are available online [9]. Overall, our queries extracted **32,124** NR bug reports from a total of 188,319 bug reports.

3.3.3 Quantitative Analysis

In order to perform our quantitative analysis, we measure the following metrics from each extracted bug report:

Active Time pertains to the period between a bug report's creation and the last update in the report.

Number of Unique Authors measures the number of people directly involved with the report, based on their user ID.

Number of Comments provides information about the extent to which a bug is discussed; this is an indication of how much attention a bug report attracts.

Number of CCs/Watchers measures the number of people that would receive update notifications for the report. It provides insights as how many people are interested in a particular bug report.

Historical Status and Resolution Changes collects data on how the status and resolution of a bug report changes throughout time.

To address RQ1, we measure the first four metrics for all the bug reports to compare the properties of *NR* bug reports (32,124) with the others (156,195). We built an analyzer tool, called NR-Bug-Analyzer [9], to calculate these metrics. It takes as input the extracted XML files and measures the first four metrics (Box 2 in Figure 3.1). Since each repository system has a different set of fields, we performed a mapping to link common fields in BUGZILLA and JIRA, as presented in Table 3.2.

To address RQ3, the last metric (historical changes) is extracted for all NR bug reports and used to mine common transition patterns. The data retrieved from bug repositories does not contain any information on how the statuses and resolutions

Table 3.2: Mapping of BUGZILLA and JIRA fields.

#	BUGZILLA	JIRA	Description
1	bug_id	key	The bug ID.
2	comment_id	id (in *comment* field)	A unique ID for a comment.
3	who	author (in *comment* field)	Name and id of the user who added a bug, a comment, or any other type of text.
4	creation_ts	created	The date/time of bug creation.
5	delta_ts	resolved (updated)	The timestamp of the last update. If *resolved* field is not available, *updated* field is used.
6	bug_status	status	The bug's latest status.
7	resolution	resolution	The bug's latest resolution.
8	cc	watches	Receive notifications.

change over time for each bug report. Thus our tool parses the HTML source of each NR bug report to extract historical data of status and resolution changes (Box 3 in Figure 3.1). BUGZILLA provides a *History Table* with historical changes to different fields of an issue, including the status and resolution fields, attachments, and comments. We extract the history of each bug report by concatenating the issue ID with the base URL of the HISTORY TABLE.[28] JIRA provides a similar mechanism called *Change History*. Our bug report analyzer tool along with all the collected (open source) empirical data are available for download [9].

3.3.4 Qualitative Analysis

In order to address RQ2, we perform a qualitative analysis that requires manual inspection. To conduct this analysis in a timely manner, we constrain the number of NR bug reports to be analyzed through random sampling. The manual classification is conducted in two steps, namely, common category inference and classification.

Common Category Inference. In the first phase, we aim to infer a set of common categories for the causes of NR bugs, i.e., understanding why they are resolved as NR. We randomly selected 250 NR reports from the open source repositories and 250 NR reports from INDUSTRIAL.

In order to infer common cause categories, each bug report was thoroughly analyzed based on the bug's description, tester/developer discussions/comments, and

[28]For example, the base URL for the *History Table* in FIREFOX BUGZILLA is https://bugzilla.mozilla.org/show_activity.cgi?id=bug_id.

historical data. We defined a set of classification definitions and rules and generated the initial set of categories and sub-categories (Box 4 in Figure 3.1). Then, the generated (sub)categories were cross-validated through discussions, merged, and refined (Box 5 in Figure 3.1). Based on an analysis of the reasons the bug reports could not be reproduced, in total, we extracted six high-level cause categories, each with a set of sub-categories, which were fed into our classification step. The categories and our classification rules are presented in Table 3.3. In the given examples in Table 3.3 and throughout the work, R refers to reporter and D refers to anyone else other than reporter.

Classification. In the second phase, we randomly selected 200 NR bug reports from each of the open source repositories. In addition, to have a comparable number of NR bug reports from the commercial application, we included all the 643 NR bug reports from INDUSTRIAL in this step. We then systematically classified these 1,643 NR bug reports, using the rules and (sub)categories inferred in the previous phase. Where needed, the sub-categories were refined in the process (Box 6 in Figure 3.1). Similar to the category inference step, each bug report was manually classified by analyzing its descriptions, discussions/comments, and historical activities. At the end of this step, each of the 1,643 NR bug reports was distributed into one of the 6 categories of Table 3.3.

Inspecting *Fixed* NR Bug Reports. To address RQ4, we performed a query on the set of NR bug reports to extract the subset that is finally changed to a Fixed resolution.

We randomly selected 20 fixed NR bug reports from the 6 repositories and manually inspected them (120) to understand why they were marked as *Fixed* (Box 7 in Figure 3.1), to understand whether the reports were initially mislabelled [116] or became reproducible/fixable, e.g., through additional information provided by the reporter. In addition, this would provide more insights in types of NR bug reports that are expected to be fixed, and the additional information that is commonly asked for, which helps reproduce NR bugs.

3.4 Results

In this section, we present the results of our study for each research question.

Table 3.3: NR Categories and Rules.

1) Interbug Dependencies: NR report cannot be reproduced because it has been implicitly fixed:

a) as a result or a side effect of other bug fixes
b) although it is not clear what patch fixed this bug
c) and the bug is a possible duplicate of or closely related to other *fixed* bugs.

Example #759127 in FIREFOX*:* R: "*It is now working with Firefox 15.0.1. I believe it was fixed by the patches to #780543 and #788600 [...].*"

2) Environmental Differences: NR report cannot be reproduced due to different environmental settings such as:

a) cashed data (e.g., cookies), user settings/preferences, builds/profiles, old versions
b) third party software, plugins, add-ons, local firewalls, extensions
c) databases, Virtual Machines (VM), Software Development Kits (SDK), IDE settings
d) hardware(mobile/computer) specifics such as memory, browser, Operating System (OS), compiler
e) network, server configuration, server being down/slow.

Example #261055 in FIREFOX*:* D: "*This is probably an extension problem. Uninstall your extensions and see if you can still reproduce these problems.*" R: "*that did it, I just uninstalled all themes and extensions, and afterwards reinstalled everything from the getextensions website. And now everything works again [...].*"

3) Insufficient Information: NR report cannot be reproduced due to lack of enough details in the report; developers request more detailed information:

a) regarding test case(s)
b) pertaining to precise steps taken by the reporter leading to the bug
c) regarding different conditions that result in the reported bug.

Example in INDUSTRIAL*:* D: "*Cannot reproduce this problem. [...] go to the main screen of the blackberry device, hold ALT and press L+O+G, it will show the logs. That information can help us to some degree.*"

4) Conflicting Expectations: NR report cannot be reproduced when there exist conflicting expectations of the application's functionality between end-users/developers/testers:

a) misunderstanding of a particular functionality or system behaviour when it works as designed (i.e., lack of documentation)
b) misunderstanding of (non)supported features, out of scope, dropping support or obsolete functionality in newer versions
c) change in requirements
d) misunderstandings turning into QA conversations

Example #29825 in ECLIPSE*:* D: "*PDE Schema works as designed [...] Since we cannot tell when you want to use tags and when you want to use reserved chars as-is, you need to escape them yourself EXCEPT, again, when between the '¡pre¿' and '¡/pre¿' tags that we recognize as a special case [...].*"

5) Non-deterministic Behaviour: NR report cannot be reproduced deterministically.

Example #MDLSITE-2255 in MOODLE*:* R: "*This happened for me again, and then went away again (started working). It seems there is an intermittent problem.*"

6) Other: NR report cannot be reproduced due to various other reasons, such as mistakes of reporters:

Example #MDL-35391 in MOODLE*:* R: "*I'm so sorry... This is not a bug. It occurred because I have been using Moodle 2.3 since beta and overwriting old source in the same directory. Could admin please delete this ticket? Sorry again.*" D: "*Thanks for the explanation, closing.*"

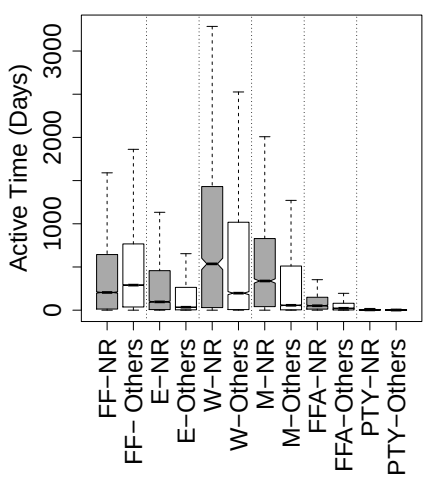


Figure 3.2: Active Time.

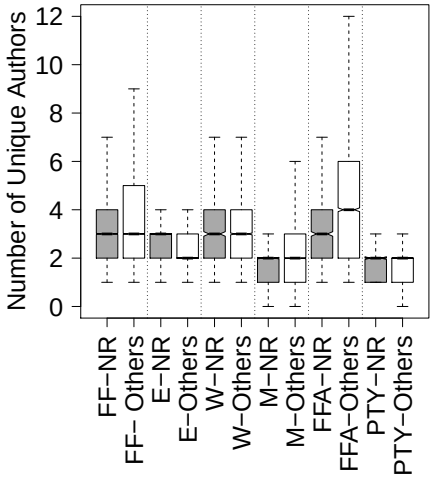


Figure 3.3: No. of Authors.

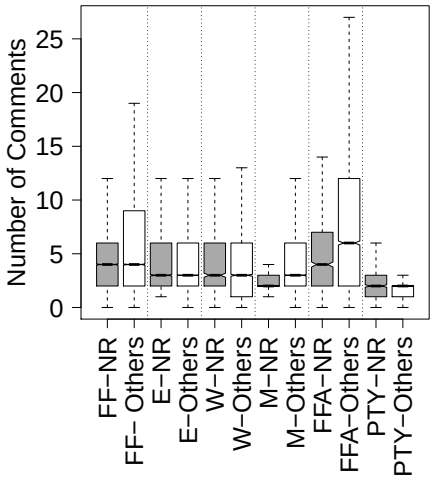


Figure 3.4: No. of Comments.

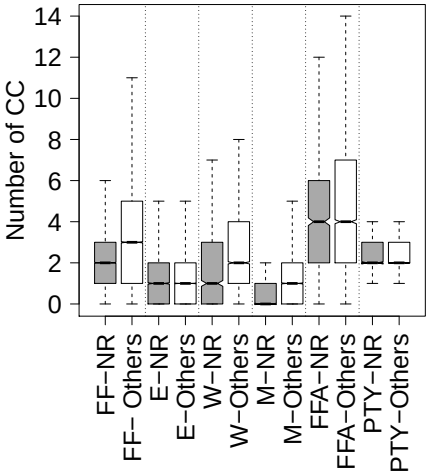


Figure 3.5: No. of Watchers.

3.4.1 Frequency and Comparisons (RQ1)

Table 3.1 presents the percentage of NR bug reports for each repository. The results of our study show that, on average, 17% of all bug reports are resolved as non-reproducible at least once in their life-cycles.

Figures 3.2–3.5 depict the results of comparing NR bug reports with other resolution types. For each bug repository, the NR bug reports are shown with grey

Table 3.4: Descriptive statistics between NR and Others, for each defined metric: Active Time (AT), # Unique Authors (UA), # Comments (C), # Watchers (W), from all repositories.

Metric	Type	Mean	Median	SD	Max	p-value
AT	NR	396	154	553	4534	0.00
	Others	313	40	531	4326	
UA	NR	3.16	3	2.22	85	0.00
	Others	3.06	2	2.61	103	
C	NR	5.14	3	7.9	459	0.03
	Others	5.93	3	12.5	1117	
W	NR	2.1	1	3	159	0.00
	Others	2.7	2	4.3	145	

background. We ignore outliers for legibility. Table 3.4 shows the mean, median, standard deviation, max and p-value (Mann-Whitney) for each comparison metric.[29] The results show that active time is significantly different, i.e., NR bug reports are on average *three months longer active* than non-NR bug reports. For the number of unique authors, comments, and CC/watchers, the results are statistically significant ($p < 0.05$), but the observed differences, having almost the same medians, are not indicative, meaning that NR bug reports receive as much attention from reporters and developers as any other resolution type.

3.4.2 Cause Categories (RQ2)

Table 3.3 shows the classification rules we used in our cause category investigation. Figure 3.6 shows the six main categories that emerged in our analysis, with their overall rate. As shown, "Interbug Dependencies" is the most common category with having 45% of the NR bugs, followed by "Environmental Differences" (24%), "Insufficient Information" (14%), "Conflicting Expectations" (12%), "Nondeterministic Behaviour" (3%) and "Other" (2%). Additionally, Figure 3.7 depicts the rate of the six cause categories per bug repository. We provide examples of each category below.

Interbug Dependencies. Bug reports in this category are those that cannot be reproduced because they have been indirectly fixed with or without explicit software patches. This category implies that there are bug reports that perhaps are not identical but semantically closely related to each other. Overall, this is the most common

[29] Min was 0 in all cases.

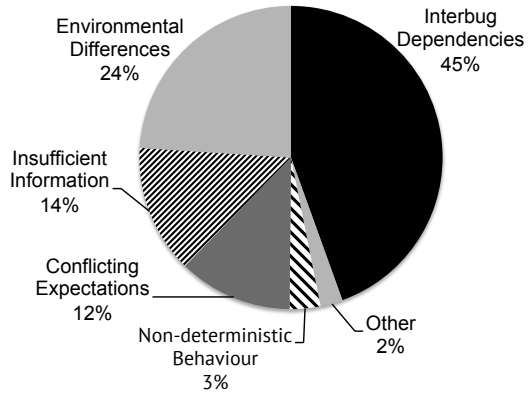


Figure 3.6: Overall Rate of NR Categories.

cause category we observed in the study (45%). Examples include:

#767543 in FIREFOX: "*D: Works for me for Beta 15, Aurora 16, and Nightly 17 with Swype Beta 1.0.3.5809 on Galaxy Nexus. I think my fix for bug #767597 fixed this bug.*"

#177769 in FIREFOX: "*D: Will resolve this as NR since we don't know which checkin fixed this.*"

#259652 in ECLIPSE: "*D: I remember fixing this but can't find the bug. Since it doesn't happen in HEAD, marking as NR.*"

#723250 in FIREFOX ANDROID: "*D: This should be fixed now with my latest changes on inbound. Specifically, bug 728369.*"

Environmental Differences. Bug reports in this category cannot be reproduced due to environmental settings that are different for developers/testers/end-users. This category accounts for 24%. Examples include:

#353838 in ECLIPSE: "*D: [..] your install got corrupted because of incompatible bundles. You could first try to disable or uninstall Papyrus and if that doesn't help try to remove the Object Teams bundles.*"

#DTP-01 in INDUSTRIAL: "*D: This has something to do with the* XCODE *settings on the build machine. Try to build it on another computer and see if it works. I cannot reproduce this on my iPhone, iPad + simulators.*"

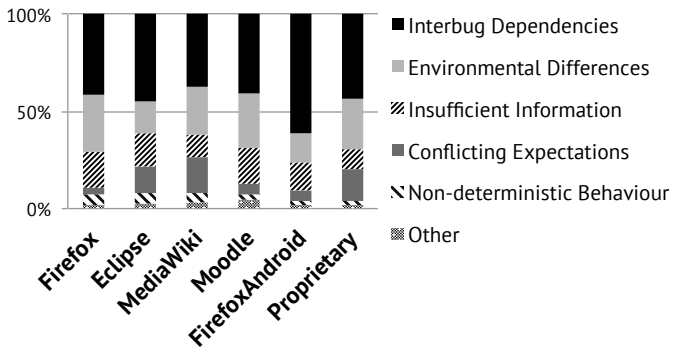


Figure 3.7: Rate of root cause categories in each bug repository.

#456734 in FIREFOX: "*R: I solved the problem by uninstalling firefox (without extensions) and installing version 3.0.1 again, and then updating it again to 3.0.2. It's a mystery for me but it helped so it's solved.*"

Insufficient Information. This is when developers need more specific and detailed information from the reporters. This category accounts for 14% of NR bug reports. Examples of this category include:

#125142 in ECLIPSE:: "*D: I haven't been able to reproduce this bug in the Java debugger [...]. Do you have a test case that displays the launch happening in the foreground? marking as NR. Please reopen with a reproducible test case if this is still occurring.*"

#3103 in MEDIAWIKI: "*D: I'm going to resolve this bug (as NR) on the grounds that without further details of the circumstances in which it occurs, there's really not much we can do...* "

#19880 in MEDIAWIKI: "*D: I've tested ru.wikipedia.org in IE5.5 on Windows 2000, IE6 on Windows XP, IE7 on Windows Vista, IE8 on Windows Vista. I was unable to reproduce this problem. Perhaps the reporter of this bug could be more specific.*"

Also tickets are also resolved as NR when there is no response from reporters for several months. For example:

#10014 in MEDIAWIKI: "*D: Closing 'support bug' due to lack of response; if the problem persists, please consider taking it up on the mediawiki-l mailing list.*"

In the FIREFOX project, an automated message is set up in the bug tracking system, which states "*This bug has had no comments for a long time. Statistically, we have found that bug reports that have not been confirmed by a second user after three months are highly unlikely to be the source of a fix to the code. [...] If this bug is not changed in any way in the next two weeks, it will be automatically resolved (NR).*"

Conflicting Expectations. This category represents bug reports in which there exist conflicting expectations of the software between end-users/developers/testers. Such conflicts could be related to a particular system behaviour, functionality, feature, software support, activity, input/output types and ranges, or specification documentation. In these scenarios the user believes there is a bug in the system since what they see is different from their mental model and/or expectations. As a result, the reported bugs are not really bugs and thus cannot be reproduced by developers. 12% of NR bug reports fall into this category. Some examples are:

#956483 in FIREFOX ANDROID: "*D: [...]* `getDefaultUAString` *is not what you think. That controls the UA of the Java HTTP requests we make in Fennec. This is not used by the Gecko networking and rendering engine. You need to use the normal Gecko preferences to change the UA. This might work: [...] R: Thanks. That works.*"

#12593 in MEDIAWIKI: "*R: [...] I can live with this because it is consistent and predictable behaviour. Thinking about it, it is probably desirable that the system works this way for migration purposes; for example: when importing a dump into a newer* MEDIAWIKI *version.*" says the reporter.

#19943 in MEDIAWIKI: "*D: Seems ok to me. As long as extensions are passing their path as either a full URL (with protocol) or relative from the docroot they should be fine [...] Checked all extensions in MW SVN that call this, and they all seem to be ok, [...]. Works for me, no real issue with* `addExtensionStyle()` *here. R: Ah, I see. Needs documenting, then [...].*"

#17265 in MEDIAWIKI: "*R: Preferably, the user and talk page of the other username should be deleted, because it'll be impracticable to merge. I hope this will be implemented and will help a lot of people. D: Works for me. There's an*

extension [...] that does this. Also, there's a maintenance script [...] that can be used for edit attribution, if someone wanted to manually merge two users."

Non-deterministic Behaviour. This category represents bugs that cannot be reproduced deterministically, meaning that the failure is intermittent and triggered at random; and thus difficult to analyze. 3% of NR bug reports are in this category. An example of a developer comment is given below:

#DTP-02 in INDUSTRIAL: "*D: This crash is very random, hard to reproduce. But my guess is it is network/analytics related. It may have to do with the user scrolling through a number of events in the Schedule section which the app cannot keep up with and then eventually crashes.*"

Other. Any other reason not covered in the other 5 categories would fall under this category (2%). One common instance in this category is bug reports that are mistakenly reported, such as opening an old ticket by mistake, or running the system with incorrect permissions.

#152 in MEDIAWIKI: "*R: For the last 3 hours I made the assumption that we could only import articles from the template namespace ... Additionally I made an error in my testing page that I just figured out. Closing...*"

#8966 in MEDIAWIKI: "*R: Shame on me, The function is not broken, I [mis] understood the syntax.*"

3.4.3 Common Transition Patterns (RQ3)

68% of NR bug reports are resolved directly from the initial status (New/Open→ Resolved(NR)). For the remaining 32%, there are various transition scenarios that NR bugs go through, changing their status and resolution. Table 3.5 presents some of the observed examples of the *status transitions* of NR bug reports. For instance, the bug report in row #6 changes resolutions 5 times: *Fixed* → *Fixed* → *Invalid* → *NR* → *Fixed*.

Table 3.5: Examples of STATUS (RESOLUTION) transitions of NR bug reports.

#	STATUS (RESOLUTION)
1	NEW→RESOLVED(**NR**)→REOPENED→RESOLVED(**NR**)→REOPENED→RESOLVED(**NR**)
2	NEW→RESOLVED(**NR**)→REOPENED→ASSIGNED→RESOLVED(FIXED)→REOPENED→ RESOLVED(FIXED)
3	NEW→RESOLVED(FIXED)→REOPENED→RESOLVED(WONTFIX)→RESOLVED(**NR**)
4	NEW→RESOLVED(FIXED)→REOPENED→RESOLVED(**NR**)→REOPENED→NEW→RESOLVED(WONTFIX)
5	NEW→RESOLVED(FIXED)→REOPENED→RESOLVED(**NR**)→REOPENED→RESOLVED(FIXED)
6	UNCONFIRMED→NEW→RESOLVED(FIXED)→REOPENED→RESOLVED(FIXED)→REOPENED→RESOLVED(INVALID) →REOPENED →RESOLVED(**NR**)→REOPENED→RESOLVED(FIXED)
7	NEW→ASSIGNED→NEW→RESOLVED(**NR**)→REOPENED→ASSIGNED→RESOLVED(FIXED)→VERIFIED
8	NEW→ASSIGNED→RESOLVED(**NR**)→REOPENED→ASSIGNED→RESOLVED(LATER)→REOPENED→NEW→ASSIGNED →NEW →ASSIGNED→RESOLVED(FIXED)
9	UNCONFIRMED→RESOLVED(INCOMPLETE)→UNCONFIRMED→RESOLVED(INCOMPLETE)→RESOLVED(FIXED) →RESOLVED(**NR**)

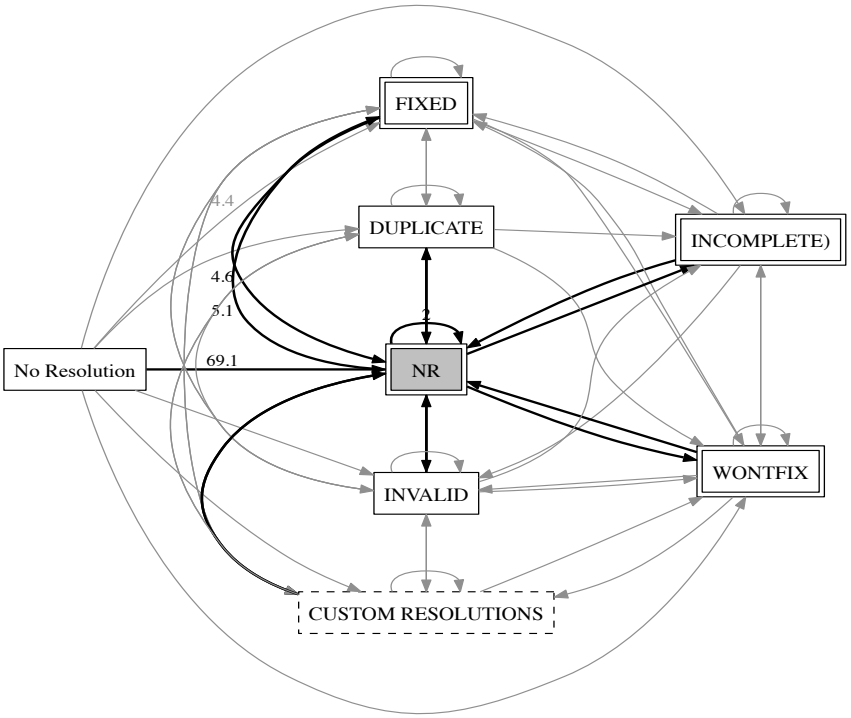


Figure 3.8: Resolution-to-Resolution Transition Patterns of NR Bug Reports (only weights larger than 2% are shown on the graph).

We examined *resolution transitions* of NR bug reports more closely, and plotted a resolution change pattern graph for the six bug repositories, which is depicted in Figure 3.8. In order to extract a common pattern for all the six repositories, we abstracted away custom (repository-specific) resolutions such as *Later*, *Remind*, *Expired*, *Rejected*, *Unresolved*, and *NotEclipse*. The custom resolutions are clustered as *Custom Resolutions* in Figure 3.8. The other resolutions shown in the graph were common in all the repositories.

We distinguish between two types of transitions in Figure 3.8: the black arrows indicate all the direct connections to the NR resolution, i.e., all the fan-ins and fan-outs; the grey arrows indicate the indirect connections between other resolutions and NR resolution. To avoid cluttering the figure, we only show weights larger than 2% on the graph. As the figure illustrates, 69% of the transitions are resolved as NR from the beginning. 4.6% of the transitions are from *Fixed* to NR. For

instance, #376902 in FIREFOX was first resolved as *Fixed* then changed to NR with a comment: "*Fixed refers to problems fixed by actual code changes to* FIREFOX. *Here NR is the correct resolution.*"

Interestingly, 5.1% of the transitions are from NR to *Fixed*. We explore fixed NR bug reports further in the following subsection.

3.4.4 Fixed Non-reproducible Bugs (RQ4)

The last column in Table 3.1 shows that, on average, 3% of all NR bugs become *Fixed*. From these, around 66% actually become reproducible as valid bugs and are fixed with code patches. They mainly fall into "Insufficient Information", "Environmental Differences", and "Conflicting Expectations" cause categories. Some examples include:

#209834 in ECLIPSE: "*D: Now, when you described the problem more precisely I realized it's a valid bug. I checked it in both 3.3.1.1 (which you're using) and N20071221-0010 (on which I'm on at this moment) and I can see the problem by clicking the 'Apply' button several times - a resource matched to *.a rule changes it's state even though the rule is enabled all the time. I'll put up a fix in a minute [...]*" ("Insufficient Information" category)

#533470 in FIREFOX: "*R: [...] I think I got to the bottom of it. The confusion was caused by kernel settings: I thought it was fixed, but actually it was just a ipv6 module getting automatically loaded. The problem still exists when there is no kernel ipv6 support available. I've submitted a simple patch to pulseaudio which will hopefully be accepted and solve the problem.*" ("Environmental Differences" category)

#245584 in FIREFOX: "*D: the problem was because NS_NewURI was failing - perhaps it was failing because there was something about the URL from IE's data that our networking system couldn't handle? Since this was particular to that URL in that person's set of typed URLs in IE, it didn't show up for everyone [...]*" ("Environmental Differences" category)

Interestingly, there were no code patches assigned to the rest of *Fixed* NR bug reports (34%). These are mislabelled reports, as the *Fixed* resolution is used when

"*a fix for a bug is checked into the tree and tested*" [24]. From these, around 24% are in the "Interbug Dependencies" category. For example:

#705166 in FIREFOX: "*D1: [...], guess this bug is fixed in the latest nightly. Working fine for me too. D2: [...] WorksForMe is not a correct resolution for this one. The bug was actually fixed by the patch in bug 704575.*"

3.5 Discussion

In this section, we discuss our general findings related to non-reproducible bug reports and discuss some of the threats to validity of our results.

3.5.1 Quantitative Analysis of NR Bug Reports

Our investigation in the quantitative attributes of NR and other types of bug reports shows that NR bug reports are as costly and important as the rest since they receive the same amount of attention as other bug report types, in terms of the number of comments and developers involved. Developers are typically reluctant to close these bug reports, and they try to involve more people and ask questions through comments. As a result, NR bug reports remain open substantially —around three months on average— longer than other types of bug reports. This clearly points to the uncertainly and low level of confidence developers have when dealing with NR bugs. Possible explanations for leaving NR bug report open longer could be that (1) they do not want to be responsible in case the NR bug turns out to be a real (reproducible) bug that needs fixing, (2) they hope more concrete information will be provided to help reproduce the bug, and/or (3) they wait for someone else to be assigned to the report who knows how to reproduce the bug.

3.5.2 Fixing NR Bugs

As our results from the six repositories have shown, on average 17% of all bug reports are resolved as NR. Among those, 3% are later marked as *Fixed*. A deeper investigation into the *Fixed* NR reports revealed that around 66% of them become indeed reproducible and fixed with code patches. The rest (34%) have no code patches assigned to them, from which around 24% are in the "Interbug Dependen-

cies" category. This means overall only 1.98% of all NR bug reports are fixed with an explicit code patch. This indicates that the majority of NR bug reports remain unreproducible.

3.5.3 Interbug Dependencies

On the other hand, 45% of all NR bugs were categorized as "Interbug Dependencies", where they were non-reproducible because they were implicitly fixed in other bug reports. Therefore, we expected the percentage of the explicit fixed NR bugs to be higher than 3%. However, it turns out that developers use the NR resolution for reports that are resolved as a consequence of other bug fixes. This implies that almost half of all NR bugs are actually (implicitly) fixed bugs. We believe coming up with automatic solutions that would cluster these interbug dependent reports based on inferred historical characteristics would help the developers in this regard.

3.5.4 Mislabelling

Our findings indicate that many reports are misclassified. These misclassifications happen not due to human errors but also because of the fact that the available resolutions in the repositories do not cover all possible scenarios. For instance, many developers use the NR (or WorksForMe) resolution when they actually mean the bug report is irrelevant, unimportant, or even fixed. This is different than the formal definition of NR bugs (see Section 3.2). We observed many inconsistencies and ambiguities around the usage of the *Fixed* and *NR* resolutions, in particular in cases where a bug report needs to be marked as "fixed with no code patches". Bugs 376902 and 705166 (subsections 3.4.3 and 3.4.4, respectively) are examples of these cases.

3.5.5 Different Domains and Environments

The active time of NR bug reports in the INDUSTRIAL repository is much lower than the open source repositories (see Figure 3.2). According to Table 3.1, NR bugs are more prevalent in the studied open-source projects, i.e., they pertain to 11–28% in the open source repositories and 4% in the industrial case. In addition, as presented in the last column of Table 3.1, although the rate of NR bug reports

is lower in the INDUSTRIAL case, the rate of fixed NR bug reports is higher, compared to the open source repositories. Although these findings apply to our sample repositories, possible reasons behind these differences could be that in commercial projects, there is more at stake and, therefore, developers (1) spend more time and effort in reproducing even hard to reproduce bugs, and (2) cannot afford to simply ignore NR bugs. It could also be that the company has a brute force policy in terms of closing bug reports as soon as possible. On the other side, developers in open source projects have less time to spend and less urgency to fix/close a bug report.

Additionally, in the mined repositories, the rate of NR bug reports in desktop applications is more than web and mobile apps, i.e., they are in the range of 13–28% for desktop, 11–12% for web, and 4–15% for mobile apps. Figure 3.2 indicates that NR bug reports have a lower active time in the repositories of the mobile apps, compared to the desktop and web applications. In addition, the difference between the medians of NR and other bug reports is the highest in the web applications, followed by the desktop, and mobile apps in our study.

3.5.6 Communication Issues

The two categories "Insufficient Information" (14%) and "Conflicting Expectations" (12%) indicate that there is a source of uncertainty and lack of proper communication between the reporters and resolvers. Herzig *et al.* [116] observed this uncertainty as a source of misclassification patterns in their recent bug report study. Equipping bug tracking systems with better collaboration tools would facilitate and enhance the communication needs between the two parties. For the category "Environmental Differences" (24%), techniques that make it easier to capture the steps leading to the bug through, e.g., record/replay methods [115], monitoring the dynamic execution of applications [59], or capturing user interactions [193] would be helpful to reproduce the bug report.

3.5.7 Threats to Validity

Our manual classification of the bug reports could be a source of internal threats to validity. In order to mitigate errors and possibilities of bias, we performed our manual classification in two phases where (1) the inference of rules was initially done

by the first author; the rules were cross-validated and uncertainties were resolved through extensive discussions and refinements between the first two authors; the generated categories were discussed and refined by all the three authors, (2) the actual distribution of bug reports into the 6 inferred categories was subsequently conducted by the first author following the classification rules inferred in the first step.

In addition, since this is the first study classifying NR bug reports, we had to infer new classification rules and categories. Thus, one might argue that our NR rules and categories are subjective with blurry edges and boundaries. By following a systematic approach and triangulation we tried to mitigate this threat. Another threat in our study is the selection and use of these bug repositories as the main source of data. However, we tried to mitigate this threat by selecting various large repositories and randomly selecting NR bug reports for analysis.

In terms of external threats, we tried our best to choose bug repositories from a representative sample of popular and actively developed applications in three different domains (desktop, web, and mobile). With respect to bug tracking systems, JIRA and BUGZILLA are well-known popular systems, although bug reports in projects using other bug tracking systems could behave differently. Thus, regarding a degree of generalizability, replication of such studies within different domains and environments (in particular for industrial cases) would help to generalize the results and create a larger body of knowledge.

All repositories except the INDUSTRIAL case are publicly available, making the quantitative findings of our study reproducible.

3.6 Related Work

We categorize related work into two classes: empirical bug report studies and failure reproduction studies.

Empirical Bug Report Studies. Empirical bug report studies have so far focused on different perspectives including understanding the quality of bug reports [61, 64, 118, 141], reassignments [109], bug report misclassifications [116], reopenings [202, 222], prediction and statistical models [96, 108, 145, 201], bug fixing and code reviewing process [219], and coordination patterns and activities around the

bug fixing process [53].

Herzig *et al.* [116] recently reported that every third 'bug report' is not really a bug report. In a manual examination of more than 7,000 bug reports of five open-source projects, they found 33.8% of all bug reports to be misclassified - that is, rather than referring to a code fix, they resulted in a new feature, an update to documentation, or an internal refactoring. This misclassification introduces errors in bug prediction models: on average, 39% of files marked as defective actually never had a bug. They estimated the impact of this misclassification on earlier studies and recommended manual data validation for future studies. The results of our study also confirm this finding.

Aranda *et al.* [53] report on a field study of coordination activities around bug fixing, through a combination of case study and a survey of software professionals. They found that the histories of even simple bugs are strongly dependent on social, organizational, and technical knowledge, which cannot be solely extracted through automation of electronic repositories, and that such automation provides incomplete and often erroneous accounts of coordination.

Zimmermann *et al.* [222] characterized how bug reports are reopened, by using the Microsoft Windows operating system project as a case study, using a mixed-methods approach. They categorized the reasons for reopening based on a survey of 358 Microsoft employees and ran a quantitative study of Windows bug reports, focusing on factors related to bug report edits and relationships between people involved in handling the bug. They propose statistical models to describe the impact of various metrics on reopening bugs ranging from the reputation of the opener to how the bug was found.

Guo *et al.* [109] present a quantitative and qualitative analysis of the bug reassignment process in the Microsoft Windows Vista project. They quantify social interactions in terms of both useful and harmful reassignments. They list five reasons for reassignments: finding the root cause, determining ownership, poor bug report quality, hard to determine proper fix, and workload balancing. Based on their study, they propose recommendations for the design of more socially-aware bug tracking systems.

To the best of our knowledge, our work is the first to report a characterization study on non-reproducible bug reports.

Failure Reproduction Studies. Apart from the empirical bug studies, there have been a number of studies [59, 115, 193] analyzing and proposing solutions for failure reproduction. Roehm *et al.* [193] present an approach to monitor interactions between users and their applications selectively at a high-level of abstraction, which enables developers to analyze user interaction traces. Herbold *et al.* [115] use a record/replay approach and monitor messages between GUI objects. Such messages are triggered by user interactions such as mouse clicks or key presses. We believe these techniques can help make NR bug reports easier to understand and reproduce. In this work, however, we perform a mining study of NR bug reports to understand their nature, leaving possible solutions for future work.

3.7 Conclusions

Working on non-reproducible bug reports is notoriously frustrating and time consuming for developers. In this work, we presented the first empirical study on the frequency, nature, and root cause categories of non-reproducible bug reports. We mined 6 bug tracking repositories from three different domains, and found that 17% of all bug reports are resolved as non-reproducible at least once in their life-cycles. Non-reproducible bug reports, on average, remain active around three months longer than other resolution types while they are treated similarly in terms of the extent to which they are discussed or the number of developers involved. In addition, our analysis of resolution transitions in non-reproducible bug repots revealed that such reports change their resolutions many times. Furthermore, around 2% of all NR bug reports are eventually fixed with code patches, while around half are implicitly 'fixed'.

Our manual examination revealed 6 common root cause categories. Our classification indicated that "Interbug Dependencies" forms the most common category (45%), followed by "Environmental Differences" (24%), "Insufficient Information" (14%), "Conflicting Expectations" (12%), and "Non-deterministic Behaviour" (3%). Our study shows that many NR bug reports are mislabelled pointing to the need for bug repository systems and developers to resolve inconstancies in the usage of the *Fixed* and *NR* resolutions.

Future work can focus on (1) bug reports in the "Interbug Dependencies" cate-

gory to design techniques that would facilitate identifying, linking, and clustering them upfront so that developers would not have to waste time on them, (2) incorporating better collaboration tools into bug tracking systems to facilitate better communication between different stakeholders to address the problem with the other NR categories.

Chapter 4

Same App, Two App Stores: A Comparative Study

Summary[30]

Each mobile platform has its own online store for distributing apps to users. To attract more users, implementing the same mobile app for different platforms has become a common industry practice. App stores provide a unique channel for users to share feedback on the acquired apps through ratings and textual reviews. To understand the characteristics of and differences in how users perceive the *same* app implemented for and distributed through *different* platforms, we perform a large-scale comparative study. We mine the characteristics of 80K app-pairs from a corpus of 2.4M apps collected from the Apple and Google Play app stores. We quantitatively compare their app-store attributes, such as ratings, versions, prices, and ask the developers about the identified differences. Further, we employ supervised machine learning to build classifiers for sentiment and feedback analysis, and classify 1.7M textual user reviews obtained from 2K of the mined app-pairs. We analyze discrepancies and root causes of user complaints at multiple levels of granularity across the two platforms.

4.1 Introduction

Online app stores are the primary medium for the distribution of mobile apps. Through app stores, users can download and install apps on their mobile devices. App stores also provide an important channel for app developers to collect user

[30]This chapter is submitted to an ACM SIGSOFT conference.

feedback, such as the overall rating of their app, and issues or feature requests through user reviews.

To attract as many users as possible, developers often implement the same app for multiple mobile platforms [86]. While ideally, a given app should provide the same functionality and high-level behavior across platforms, this is not always the case in practice [88]. For instance, a user of the Android STARBUCKS app complains: "*I downloaded the app so I could place a mobile order only to find out it's only available through the iPhone app.*" Or an iOS NFL app review reads: "*on the Galaxy you can watch the game live..., on this (iPad) the app crashes sometimes, you can't watch live games, and it is slow.*"

Currently, iOS [21] and Android [19] dominate the app market, each with over 1.5 million apps in their respective app stores; therefore, in this work, we focus on these two platforms. We present a large-scale study on mobile *app-pairs* — same app implemented for iOS and Android platforms — in order to analyze and compare their various app-store attributes, textual user reviews, and root causes of user complaints. We mine the two app stores and employ a mixed-methods approach using both quantitative and qualitative analysis. Our study can help developers to understand the dynamics of different markets, end-user perceptions, and reasons behind varying success rates for the same app across platforms.

Researchers have mined app stores for analyzing user-reviews [71, 124, 185], app descriptions [105, 143, 200], and app bytecode [54, 194, 195]. However, existing studies focus on one store at a time only. To the best of our knowledge, we are the first to study the *same* apps, published on *different* app stores.

Overall, our work makes the following main contributions:

- We present the first dataset of 80,169 app-pairs, extracted by analyzing the properties of 2.4M apps from the Google Play and Apple app stores. Our app-pair dataset is publicly available [165].
- We compare the app-pairs' attributes such as ratings, categories, prices, versions, updates and explore causes for variation among them.
- We identify app-pairs on the top rated 100 free and 100 paid apps listed on Google Play and Apple app stores and explore some of the obstacles that

prevent developers from publishing their apps in both stores.

- We extract and classify user reviews to compare user sentiment and complaints across app-pairs.
- Finally, we measure an app's success using a proposed metric that combines reviews, ratings, and stars. Based on developers' feedback, we provide insight into the varying success rates of app-pairs.

4.2 Methodology

Our analysis is based on a mixed-methods research approach [80], where we collect and analyze both quantitative and qualitative data. We address the following research questions in our study:

RQ1. How prevalent are app-pairs and what are the app-store characteristics of app-pairs?

RQ2. What portion of the top rated apps are app-pairs? Why are some apps only available on one platform?

RQ3. Are the app-pairs equally successful on both platforms?

RQ4. What are some of the major concerns or complaints on each platform?

Figure 4.1 depicts our overall approach. We use this figure to illustrate our methodology throughout this section. We first describe how we collect our datasets along with their attributes and descriptive statistics, and then describe how we detect app-pairs. Finally, we explain the analysis steps performed on the app-pairs. Additionally, to better understand our findings, we ask app developers about some of the reasons behind the differences in app-pair attributes such as prices, update frequencies, success rates, and top-rated apps existing only on one platform.

4.2.1 Data Collection

The first step in our work is to collect Android and iOS apps along with their attributes (Box 1 in Figure 4.1). To this end, we use two open-source crawlers,

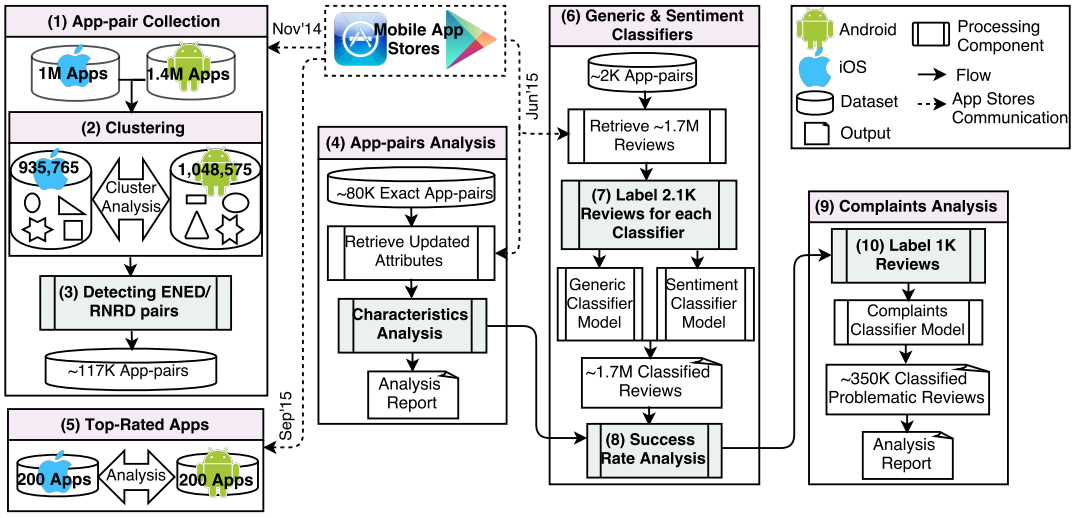


Figure 4.1: Overview of our methodology.

namely Google Play Store Crawler [103] and Apple Store Crawler [52] to mine apps from the two app stores, respectively. We collect app attributes that are available on two stores. Since each app store has a different set of attributes, we first map the attributes that exist in both app stores and ignore the rest. For instance, information about the number of downloads is only available for Android but not iOS, and thus, we ignore this attribute. This mining step was conducted in November 2014 and resulted in 1.4M Android apps and 1M iOS apps. Table 4.1 outlines the attributes we collect. We save this data in a MONGODB database [165], which takes up approximately 2.1GB of storage.

4.2.2 Matching Apps to Find App-Pairs

After creating the Android and iOS datasets separately, we set out to find app-pairs by matching similar apps in the two datasets. The unique IDs for iOS and Android apps are different and thus cannot be used to match apps, i.e., Android apps have an application ID composed of characters while iOS apps have a unique 8 digit number. However, app names are generally consistent across the platforms since they are often built by the same company/developer. Thus, we use app name and developer name to automatically search for app-pairs. This approach could result

Table 4.1: Collected app-pair attributes

#	iTunes; Google	Description
1	name; title	Name of the app.
2	developerName; developer_name	Name of the developer/company of the app.
3	description; description	Indicates the description of the app.
4	category; category	Indicates the category of the app; 23 Apple & 27* Google categories.
5	isFree; free	True if the app is free.
6	price; price	Price ($) of the app.
7	ratingsAllVersions; ratingsAllVersions	Number of users rating the app.
8	starsVersionAllVersions; star_rating	Average of all stars (1 to 5) given to the app.
9	version; version_string	User-visible version string/number.
10	updated; updated	Date the app was last updated.

*Google has its apps split into Games and Applications. We count Games as one category.

in multiple possible matches because (1) on one platform, developers may develop close variants of their apps with extra features that have *similar* names (See Figure 4.2); (2) the same app could have slightly different names across the platforms (See Figure 4.3–a); (3) the same app could have slightly different developer names across the platforms (See Figure 4.3–b).

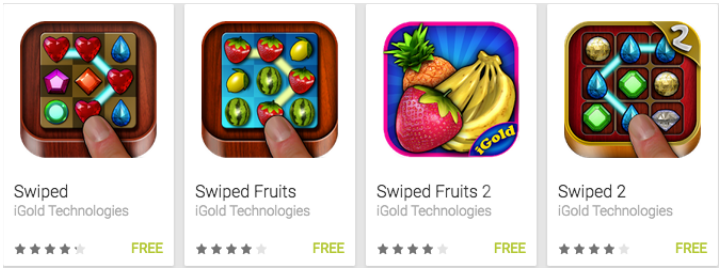


Figure 4.2: Android Cluster for Swiped app.

Clustering

To find app-pairs more accurately, we first cluster the apps on each platform. This step (outlined in Box 2 of Figure 4.1) groups together apps on each platform that belong to the same category, have *similar* app names (i.e., having the exact root word, but allowing permutations) and the same developer name. Figure 4.2 is an

Algorithm 1: Android & iOS Clustering Algorithm

```
   input  : Collection of Apps (APPS)
   output: APPS with Cluster IDs
 1 begin
 2     foreach i = 0, i <COUNT(APPS), i++ do
 3         appName ← APPS[i].name
 4         devName ← APPS[i].devName
 5         category ← APPS[i].category
 6         clusterID ← appName + devName +category
 7         if CHECK(APPS[i], clusterID) then
 8             APPS[i].clusterID ← clusterID
 9         end
10         foreach j = 0, j <COUNT(APPS), j++ do
11             if SIMILAR(APPS[j].name, appName) &
               APPS[j].devName ≡ devName then
12                 and APPS[j].category==category APPS[j].clusterID ← clusterID
13             end
14         end
15     end
16 end
```

example of a detected Android cluster. The apps in this cluster are all developed by *iGold Technologies*, belong to the Game category and have *similar* (but not exact) names.

To cluster the apps, we execute Algorithm 1 on the Android and iOS datasets, separately. The algorithm takes as input a collection of apps and annotates the collection to group the apps together. We loop through the entire collection of apps (line 2). For each app, we extract the app name, developer name, and category (lines 3, 4 and 5). Next, if an app has not been annotated previously we annotate it with a unique *clusterID* (line 7). Then we search for apps, which have a similar name, exact developer name, and belong to the same category (line 11). If a match is found, we annotate the found app with the same *clusterID* (line 12).

Detecting App-Pairs

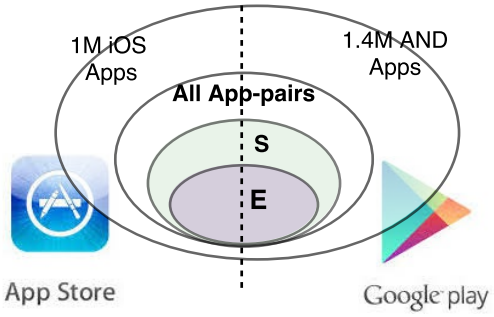
We consider an app-pair to consist of the iOS version and the Android version of the same app. In our attempt to find app-pairs (Box 3 in Figure 4.1), we noticed that Android and iOS apps have different naming conventions for app names and developer names. For instance, Figure 4.3–a depicts an app developed by *'Groupon, Inc.'*, with different naming conventions for app names; *'Groupon - Daily Deals,*

Coupons' on the Android platform whereas *'Groupon - Deals, Coupons & Shopping: Local Restaurants, Hotels, Beauty & Spa'* on the iOS platform. Similarly, Figure 4.3–b shows the *'Scribblenauts Remix'* app, which has the exact name on both platforms, but has differences in the developer's name.

(a) (b)

Figure 4.3: a) Groupon and b) Scribblenauts apps. Android apps are shown on top and iOS apps at the bottom.

Figure 4.4 shows the app-pairs we find using matching criteria with different constraints. Criteria E looks for app-pairs having *exact app and developer name* whereas Criteria S relaxes both the app and developer name, thus matching the apps in Figure 4.3 as app-pairs.



ID	App-pair Criteria
E	EXACT(AppName) & EXACT(DevName)
S	SIMILAR(AppName) & SIMILAR(DevName)

Figure 4.4: Matching App-pair Criteria.

To find app-pairs, we match the Android clusters with their iOS counterparts. First, we narrow down the search for a matching cluster by only retrieving those with a similar developer name. This results in one or more possible matching clusters and we identify the best match by comparing the items in each cluster.

Thus, for each app in the Android cluster, we look for an exact match (criteria E) in the iOS cluster. If no match is found, we relax the criteria and look for matches having a similar app and developer name (criteria S). The set of all possible app-pairs is a superset of S, and S is a superset of E, as depicted in the Venn diagram of Figure 4.4.

Exact App-Pairs

We perform the rest of our study using criteria E, which provides a large-enough set of app-pairs for analysis. To confirm that criteria E correctly matches app-pairs, we manually compared app names, descriptions, developers' names, app icons and screenshots of 100 randomly selected app-pairs and the results indicated that there are no false positives.

4.2.3 App-store Attribute Analysis

To address RQ1, we recapture (Box 4 in Figure 4.1) the app-pairs' attributes (see Table 4.1) and update our dataset with the latest data. This step was conducted in June 2015. We compare the updated attributes between the iOS and Android app-pairs and present the results in Section 4.3.

To address RQ2, we use the iTunes Store RSS Feed Generator [127] to retrieve the top rated apps, which enables us to create custom RSS feeds by specifying feed length, genres, country, and types of the apps to be retrieved. These feeds reflect the latest data in the Apple app store. The Google Play store provides the list of top rated Android apps [209] as well. We collected the top 100 free and 100 paid iOS apps belonging to all genres, as well as top 100 free and 100 paid Android apps belonging to all categories, in September 2015 (Box 5 in Figure 4.1). To check whether a top app exists on both platforms, we used our exact app-pair technique as described in the previous section. Since the lists were not long, we also manually validated the pairs using the app name, developer name, description and screenshots in the other market.

4.2.4 User Reviews

In addition to collecting app-store attributes for our app-pairs in RQ1, we analyze user reviews of app-pairs to see if there are any discrepancies in the way users experience the same app on two different platforms (RQ3 and RQ4).

To that end, we first select 2K app-pairs that have more than 500 ratings, from our app-pair dataset. This allows us to target the most popular apps with enough user reviews to conduct a thorough analysis. To retrieve the user reviews, we use two open-source scrapers, namely the iTunes App Store Review Scraper [126] and the Goole Play Store Review Scraper [104]. In total, we retrieve 1.7M user reviews from the 2K app-pairs.

The goal is to semi-automatically classify the user reviews of the app-pairs and compare them at the app and platform level. To achieve this, we use natural language processing and supervised machine learning to train two classifiers (Box 6 in Figure 4.1). Each classifier can automatically put a review into one of its three classes.

Generic Feedback Analysis. As shown in Table 4.2, our generic feedback classifier (C1) has three unique class labels {*Problem Discovery, Feature Request, Non-informative*}; where *Problem Discovery* implies that the user review pertains to a functional (bug), or non-functional (e.g., performance), or an unexpected issue with the app. *Feature Request* indicates that the review contains suggestions, improvements, requests to add/modify/bring back/remove features. Finally, *Non-informative* means that the review is not a constructive or useful feedback; such reviews typically contain user emotional expressions (e.g., 'I love this app', descriptions (e.g., features, actions) or general comments. We have adopted these classes from recent studies [71, 185] and slightly adapted them to fit our analysis of user complaints and feedback on the two platforms.

Sentiment Analysis. Additionally, we are interested in comparing the sentiment (C2 in Table 4.2) classes of {*Positive, Negative, Neutral*} between the reviews of app-pairs. We use these rules to assign class labels to review instances. Table 4.2 provides real review examples of the classes in our classifiers.

Table 4.2: Real-world reviews and their classifications.

C1 – Generic Feedback Classifier	
1	**Problem Discovery**: "*Videos don't work. The sound is working but the video is just a black screen.*"
2	**Feature Request**: "*I would give it a 5 if there were a way to exclude chain restaurants from dining options.*"
3	**Non-informative**: "*A far cry from Photoshop on the desktop, obviously, but still a handy photo editor for mobile devices with great support.*"
C2 – Sentiment Classifier	
1	**Positive**: "*Amazing and works exactly how I want it to work. Nothing bad about this awesome and amazing app!*"
2	**Negative**: "*The worst, I downloaded it with quite a lot of excitement but ended up very disappointed*"
3	**Neutral**: "*its aight good most of the time but freezes sometimes*"

Building Classifiers

Overall, we label 2.1K reviews for training each of the two classifiers (Box 7 in Figure 4.1).

We randomly selected 1,050 Android user reviews and 1,050 iOS user reviews from 14 app-pairs. We experimented with the number of app-pairs to find the best F-measures. Additionally, these app-pairs were in the list of the most popular apps and categories in their app stores. This diversity of apps and categories allows us to build robust classifiers capable of accurately labeling reviews which are from different contexts, contain different vocabularies, and written by different users. The manual labeling of reviews was first conducted by one author following the classification rules inferred in Table 4.2. Subsequently, any uncertainties were cross-validated and resolved through discussions and refinements between the authors.

To build our classifiers, we use the *bags of words* representation, which counts the number of occurrences of each word to turn the textual content into numerical feature vectors. Next, we preprocess the text, tokenize it and filter stop words. We use the feature vectors to train our classifier and apply a machine learning algorithm on the historical training data. In this work, we experimented with two well-known and representative semi-supervised algorithms, Naive Bayes (NB) and Support Vector Machines (SVM). We use the Scikit Learn Tool [198] to build our classifiers. The training and testing data for our classifiers are 1,575 and 525 re-

views. We repeated this trial 25 times to train both our generic and sentiment classifiers and compared the NB and SVM algorithms. We choose the generic (C1) and sentiment (C2) classifiers with the best F-measure. Finally, we use the generic and sentiment classifiers to classify ∼1.7M reviews of 2K app-pairs. The results are presented in Section 4.3.

4.2.5 Success Rates

An important indication of the success of an app is the number of its downloads by users. However, as explained in Section 4.2.1, although download counts are available for Android apps, Apple does not publish this information for iOS apps. This means we need to find another metric to measure and compare the success rates of app-pairs. Since user feedback can also be an indication of the success or failure of apps, we use the classified reviews from the previous step, along with their ratings and number of stars, to compute a success metric (Box 8 in Figure 4.1), which is defined as follows:

$$SuccessRate = \frac{R_{rev} + (\bar{\mathbb{S}} \times AR)}{7} * 100 \tag{4.1}$$

where

1. $R_{rev} = P_{rev} - (100 - \frac{N_{rev} + PD_{rev}}{2})$
2. $\bar{\mathbb{S}}$ = Average value for the stars of an app.
3. $AR = \begin{cases} 0.25 & \text{if } AppRatings \leq Q_1 \\ 0.50 & \text{if } Q_1 < AppRatings \leq Q_2 \\ 0.75 & \text{if } Q_2 < AppRatings \leq Q_3 \\ 1.00 & \text{if } Q_3 < AppRatings \end{cases}$

and P_{rev}, N_{rev}, and PD_{rev} depict the rates of *Positive*, *Negative*, and *Problem Discovery* reviews for each app. Also, each app has an overall ratings (*AppRatings*) on the app stores, which indicates the number of users who rate the app. We calculate the Q_1, Q_2, and Q_3 as the first, second (median) and third quartiles of app ratings for the 2K app-pairs for each platform. We choose to place more emphasis on the *ratings* and *stars* since that information is collected directly from the app stores and strongly reflects the users' view of the app. Furthermore, we multiply them together to fairly compare apps which have high *stars* and low *ratings* and

apps which have average *stars* but high *ratings*. We use a range based on quartiles for *AR* as opposed to the actual number of ratings to normalize the data and fairly compare the apps. As for the *reviews* R_{rev}, we add the N_{rev}, and PD_{rev} together and divide them by 2 and then subtract them from 100 to avoid negative numbers; we subtract the result from the P_{rev} to get an overall score. The star values $\bar{\mathbb{S}}$, ratings *AR*, reviews R_{rev}, and the overall *SuccessRate* are between [0–5], [0.25–1], [0–2], and [0–100%], respectively, for each app.

Major Complaints Analysis

The goal in RQ4 is to understand the nature of user complaints and how they differ on the two platforms (Box 9 in Figure 4.1). To address this, we first collect the *Problem Discovery* reviews for 20 app-pairs having (1) the biggest differences in success rates between the platforms, (2) over 100 problematic reviews. These 20 app-pairs are split into 10 in which Android is more successful than iOS and 10 in which iOS is more successful than Android. Then, we manually inspect and label 1K problematic reviews (Box 10 in Figure 4.1), by randomly selecting 25 Android user reviews and 25 iOS user reviews from each of the 20 app-pairs. We noticed that user complaints usually fall into the following five subcategories:

1. *Critical*: issues related to crashes and freezes;
2. *Post Update*: problems occurring after an update/upgrade;
3. *Price Complaints*: issues related to app prices;
4. *App Features*: issues related to functionality of a feature, or its compatibility, usability, security, or performance;
5. *Other*: irrelevant comments.

Table 4.3 provides real example reviews of these categories. We use the labelled dataset to build a complaints classifier to automatically classify ~350K problematic reviews of our 2K app-pairs.

Table 4.3: Reviews and subcategories of problem discovery.

C3 – Complaints Classifier (Classes and Examples)	
1	**Critical**: "*Crashing is terrible. This game crashes every two battles. Really annoying please fix.*"
2	**Post Update**: "*after the recent update, the music bar has disappeared and I'm not able to listen to my fav. Music anymore, have to log on from safari to listen to it!.*"
3	**Price Complaints**: "*Don't buy the pro I spent 4 dollars on the pro now it won't even let me assess the app without sending me back to the home screen! Four dollars is a lot when it comes to apps and now it's freakin gone!.*"
4	**App Feature**: "*Video video. Is not working !!!!!!!! Please fix the video player. Thats not right.*"

4.2.6 Datasets and Classifiers

All our extracted data, datasets for the identified app-pairs and the 2K app-pairs along with their extracted user reviews, as well as all our scripts and classifiers are available for download [165].

4.3 Findings

In this section, we present the results of our study for each research question.

4.3.1 Prevalence and Attributes (RQ1)

Cluster of apps

We found 1,048,575 (∼1M) Android clusters for 1,402,894 (∼1.4M) Android apps and 935,765 (∼0.9M) iOS clusters for 980,588 (∼1M) iOS apps in our dataset. The largest Android cluster contains 219 apps[31] and the largest iOS cluster contains 65 apps.[32] Additionally, 7,845 Android and 9,016 iOS clusters have more than one item. The first row of Table 4.4 shows descriptive statistics along with p-value (Mann-Whitney) for cluster sizes, ignoring clusters of size 1. Figure 4.5 depicts the results of comparing the cluster sizes between the two platforms. We ignore outliers for legibility. The results are statistically significant ($p < 0.05$) and show that while Android clusters deviate more than iOS clusters, the median in iOS is

[31] https://play.google.com/store/search?q=Kira-Kira&c=apps&hl=en
[32] https://itunes.apple.com/us/developer/urban-fox-production-llc/id395696788

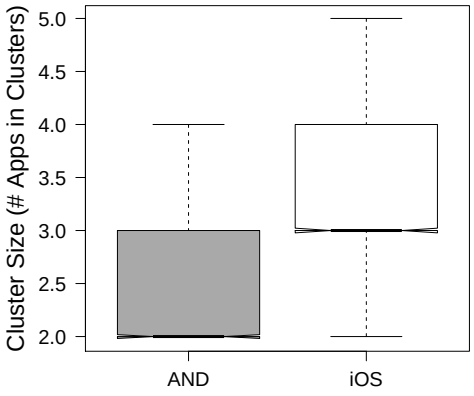


Figure 4.5: Clusters.

higher than Android by one. This could be explained perhaps by the following two observations: (1) not all iOS apps are universal apps (i.e., run on all iOS devices) and some apps have both iPhone-only and iPad-only apps instead of one universal app; (2) iOS has more free and pro versions of the same app than Android.

Prevalence of app-pairs

We found 80,169 (∼80K) exact app-pairs (Criteria E in Figure 4.4), which is *8% of the total iOS apps, and 5.7% of the total Android apps in our datasets*. When we relax both app and developer names, the number of app-pairs increases to 116,326 (∼117K) app-pairs, which is *13% of our iOS collection and 9.2% of our Android collection*. While our dataset contains apps from 22 Apple and 25 Google categories, most of the pairs belong to popular categories, which exist on both platforms: {*Games, Business, Lifestyle, Education, Travel, Entertainment, Music, Finance, Sports*}.

Ratings & Stars

Interestingly, 68% of Android and only 18% of iOS apps have ratings (the number of users who have rated the app). The Median is 0 for all iOS and 3 for all Android, as depicted in Table 4.4. However, when we only consider apps with at least one rating, the median increases to 21 for iOS and 11 for Android (See Figure 4.6).

Table 4.4: Descriptive statistics for iOS and Android (AND), on Cluster Size (C), Ratings (R), Ratings for all apps (R*), Stars (S), Stars for all apps (S*), and Price (P).

ID	Type	Min	Mean	Median	SD	Max	P
C	iOS	2	3.30	3.00	2.11	65	0
	AND	2	3.00	2.00	3.69	219	
R	iOS	5	1,935.00	21.00	26,827.24	1,710,251	0
	AND	1	4,892.00	11.00	171,362.40	28,056,146	
R*	iOS	0	353.10	0.00	11,483.19	1,710,251	0
	AND	0	3,302.00	3.00	140,807.60	28,056,146	
S	iOS	1	3.80	4.00	0.90	5	0
	AND	1	4.04	4.10	0.82	5	
S*	iOS	0	0.70	0.00	1.52	5	0
	AND	0	2.73	3.70	2.01	5	
P	iOS	0	3.41	1.99	9.53	500	0
	AND	0	3.38	1.88	9.13	210	

*Including apps that have no ratings/stars.

We ignore outliers for legibility. Furthermore, we compare the differences between ratings for each pair. In 63% of the pairs, Android apps have more users rating them (on average 4,821 more users) whereas in only 5% of the pairs, iOS apps have more users rating them (on average 1,966 more users). Additionally, the results of ratings in Table 4.4 are statistically significant ($p < 0.05$), indicating that *while Android users tend to rate apps more than iOS users, for the rated app-pairs, iOS apps have higher ratings.* The categories with the highest ratings were { *Personalization, Communication, Games*} on Android and {*Games, Social Networking, Photo & Video*} on iOS.

Similarly, 68% of Android and 18% of iOS apps have stars (i.e., the 1 to 5 score given to an app). When we consider the apps with stars, the median increases to 4 for iOS and 4.1 for Android (See Figure 4.7). Additionally, comparing the stars with the data, we had originally from November 2014, does not show considerable change on the two platforms. Comparing the differences between the stars for each pair, in 58% of the pairs, Android apps have more stars while in only 8% of the pairs, iOS apps have more stars. Additionally, while the results of stars are statistically significant ($p < 0.05$), the observed differences, having almost the same medians (see Table 4.4), are not indicative, meaning that *although Android users tend to star apps more than iOS users, the starred app-pairs have similar stars.* The categories with the highest number of stars were {*Weather, Music & Audio, Comics*} on Android and {*Games, Weather, Photo & Video*} on iOS.

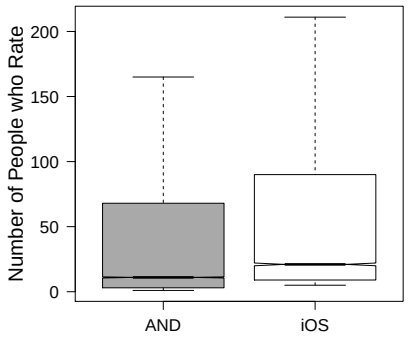


Figure 4.6: Ratings.

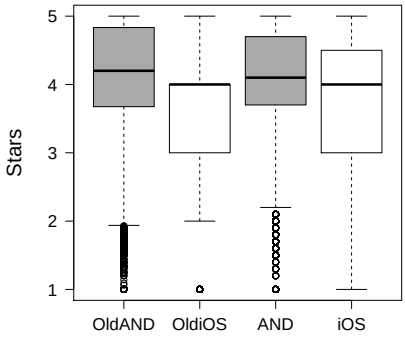


Figure 4.7: Stars.

Prices of app-pairs

The normal expectation is that the same app should have the same price on both platforms. However, comparing prices, *88% of app-pairs have different prices for their Android versus iOS versions*. Comparing the rate of free and paid apps, 10% of the Android and 12% of iOS apps are paid. In 34% of the pairs, iOS apps have a higher price whereas in 54% of the pairs, Android apps have a higher price (See Figure 4.8). As Table 4.4 shows, the median is 1.99 for iOS and 2.11 for Android. The results are statistically significant ($p < 0.05$), and indicate that for app-pairs *while more paid iOS apps exist, Android paid apps have slightly higher prices.* The categories with the most expensive apps were {*Medical, Books & Reference, Education*} on Android and {*Medical, Books, Navigation*} on iOS.

To understand the reasons behind the price differences, we sent emails to all the developers of app-pairs with price differences of more than $10 (US) and asked them why their app-pairs were priced differently on the two platforms. Out of 52 emails, we received 25 responses and categorized the reasons:

Different monetization strategies per app store, i.e., paid apps vs. in-app purchases vs. freemium vs. subscription. For instance, in the Android version, the features exist as part of the app, while in the iOS version, the features are optional or provided as an in-app purchase. A developer responded, "*The difference is that the Android version includes consolidation ($9.99), chart-*

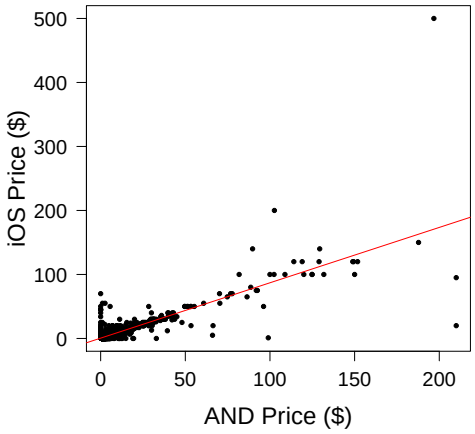


Figure 4.8: Prices.

ing ($14.99), reports ($9.99) and rosters ($14.99), whereas these are 'in app purchase' options on Apple devices."

Different set of features on the two platforms. e.g., a developer responded, "*the iOS version offers more features than the Android version.*"

Development/Maintenance cost of the app. Different parameters play a role in the development/maintenance cost of an app, the cost varies depending on the characteristics of the app e.g., a developer responded, "*Apple forces developers to constantly migrate the apps to their latest OS and tool versions, which causes enormous costs. ... the effort to maintain an App on iOS is much higher than on Android.*" While another developer stated, "*It's easier to develop for (less device fragmentation), Android is relatively expensive and painful to create for and much harder to maintain and support.*"

Demographic biases Developers price the app based on certain user demographics; e.g, "*since iOS phones are $1,000+ in this market, these users are rich and willing to pay more readily for apps than Android users.*"

Exchange rate differences e.g., "*price in both are set to 99 EUR as we are mainly selling this in Europe. Play Store apparently still used USD converted by the exchange rate of the day the app was published.*"

We have to note that some of the developers we contacted were unaware of price differences on the stores for their app-pairs.

Versions and last updated

While the app stores' guidelines suggest that developers follow typical software versioning conventions such as *semantic versioning*[33] — (major.minor.patch) — they do not enforce any scheme. Therefore, mobile apps exhibit a wide variety of versioning formats containing letters and numbers, e.g., date-based schemes (year.month.patch). Our analysis indicates that only *25% of the app-pairs have identical versions*. When we inspect the major digit only, 78% of the pairs have the same version. 13% of the Android apps have a higher version while 9% of the iOS apps have a higher version.

Comparing the date the apps were last updated, 58% of the app-pairs have an iOS update date which is more recent than Android; while 42% have a more recent Android update date. Interestingly, 30% of the app-pairs have update dates which are more than 6 months apart. To understand why developers update their apps inconsistently across the platforms, we emailed all the developers of app-pairs which were recently updated (after January 2016) on either platform; and in which the other platform has not been updated in 80 days or more. Out of 65 emails, we received 15 responses and categorized the reasons below:

Ease of releasing updates e.g., "*we are experimenting with a new 3D printing feature, and wanted to try it on Android before we released it on iOS. As you know, developers can release updates quickly to fix any problems on Android, but on iOS, we have to wait a week or two while Apple reviews the game.*"

Preferring one platform over the other for various reasons, e.g., "*while there are many Android handsets and potentially many downloads, this doesn't translate well to dollars spent, relative to iOS.*"

Developer skills and expertise The developers might be more skilled at building apps for one of the platforms than the other; e.g., "*I learned iOS first and am developing for iOS full time, so everything is easier for me with iOS.*"

[33] http://semver.org

Update due to a platform-specific feature e.g., "*only updated the iOS version to switch over to AdMob as the advertising network for iOS. Apple announced that iAd is being discontinued.*"

We have to note that, similar to the reasons behind the price differences, some of the developers we contacted, mentioned that the development/maintenance cost of the app could affect updates on either platform.

Finding 1*: We found 80K exact app-pairs. While Android users tend to rate and star apps much more than iOS users, the Stars were equal and the Ratings were higher on the iOS platform. Also, it is common to have price and update date mismatches for app-pairs.*

4.3.2 Top Rated Apps (RQ2)

Interestingly, our analysis on the top 100 *free* iOS and Android apps shows that 88% of the top iOS and 86% of the top Android apps have pairs. 37 app-pairs are in the top 100 list for both platforms. On the other hand, for the top 100 *paid* iOS and Android apps, 66% of the top iOS and 79% of the top Android apps have pairs. 30 of the paid pairs are in the top 100 for both platforms.

Furthermore, we sent emails to all the developers of apps with no pairs and asked why their top rated Android or iOS app was only published for one platform. One respondent stated the following: "*while the situation is unique with every company's strategy and different with every individual product, there's always a set of driving factors with expertise, success record, target market size and cost of development that influence the general strategy, starting from which market to enter first, [to] when and if to enter another market (if at all).*" Out of 81 emails, we received 29 responses and categorized the reasons below:

Lack of resources e.g., "*building the same app across two platforms is actually twice the work given we can't share code ... so we'd rather make a really good app for one platform than make a mediocre one for two.*"

Platform restrictions e.g., "*I've only focused on the Android platform simply because Apple doesn't allow for much customization to their UI.*"

Revenue per platform e.g., "*In my experience, iOS users spend more money, which means a premium [paid app with a price higher than 2.99] is more likely to succeed. ... while the Android platform has the market size, it proves to be harder for small [companies] to make good money.*"

Fragmentation within a platform e.g., "*my app is very CPU intensive and thus, I must test it on every model. With a much-limited number of models for iOS, it's feasible. On Android, it's impossible to test on every model and quality would thus suffer.*"

Similar apps already exist on the other platform e.g., "*Apple devices already have a default podcast app.*"

Platform-specific apps e.g., *iTunes U* app developed by Apple whereas *Google Play Games* app developed by Google Play.

A common response from developers was that the app for the other platform is under development. We also observed more antivirus programs are available for Android. Thus, security could be more of an issue for the open-source Android than the restricted iOS platform.

***Finding 2**: On average, 80% of the top rated apps are app-pairs. Main reasons for apps existing only on one platform, include: lack of resources, platform restrictions and fragmentation, and revenue per platform.*

4.3.3 Success Rate (RQ3)

Classification. To evaluate the accuracy of the classifiers, we measure the F-measure = $\frac{2\times Precision\times Recall}{Precision+Recall}$ for the Naive Bayes and SVM algorithms, listed in Table 4.5, where Precision = $\frac{TP}{TP+FP}$ and Recall = $\frac{TP}{TP+FN}$. We found that SVM achieves a higher F-measure. On average, F(SVM) = 0.84 for the generic classifier and F(SVM) = 0.74 for the sentiment classifier. The F-measures obtained by our classifiers are similar to related studies such as [185] (0.72) and [71] (0.79). We selected the classifiers with the best F-measures and used them to classify 1,702,100 ($\sim$1.7M) reviews for 2,003 ($\sim$2K) app-pairs.

Table 4.5: Statistics of 14 Apps used to build the classifiers (C1 = Generic Classifier, C2 = Sentiment Classifier, NB = Naive Bayes Algorithm, SVM = Support Vector Machines Algorithm, Train = Training pool).

#	App	GoogleCategory	AppleCategory	Train	Test	F(C1-NB)	F(C2- NB)	F(C1-SVM)	F(C2-SVM)
1	FruitNinja	Game(Arcade)	Game	150	50	0.77	0.68	0.83	0.75
2	UPSMobile	Business	Business	150	50	0.80	0.69	0.82	0.76
3	Starbucks	Lifestyle	Food & Drink	150	50	0.75	0.63	0.84	0.77
4	YellowPages	Travel & Local	Travel	150	50	0.78	0.62	0.85	0.75
5	Vine	Social	Photo & Video	150	50	0.81	0.70	0.84	0.76
6	Twitter	Social	Social Networking	150	50	0.79	0.67	0.84	0.75
7	AdobePhotoShop	Photography	Photo & Video	150	50	0.82	0.72	0.85	0.75
	Total / Average of 7 Apps			1,050	350	0.78	0.67	**0.85**	**0.76**
8	YahooFinance	Finance	Finance	75	25	0.75	0.66	0.84	0.73
...	...			...	...	...	...	...	...
	Total / Average of 14 Apps			1,575	525	0.77	0.65	**0.84**	**0.74**

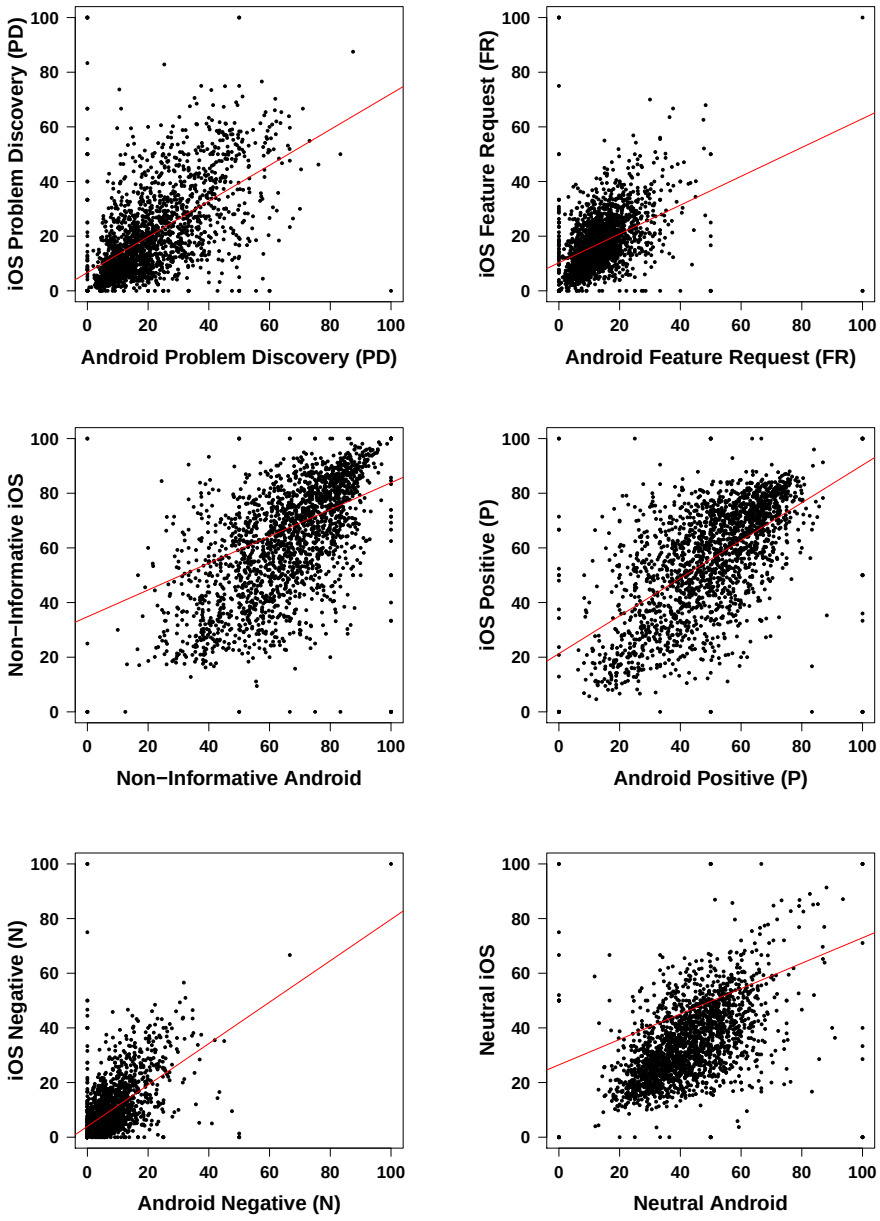


Figure 4.9: The rates of classifiers' categories for our 2K app-pairs, where each dot represents an app-pair.

Figure 4.9 and Figure 4.10 plot the distribution of the rates for the main categories in the sentiment and generic classifiers for our app-pairs as well as the success rate for our app-pairs. Each dot represents an app-pair. The statistical results are depicted in Table 4.6. On average, *Feature Request*, *Positive*, and *Negatives* re-

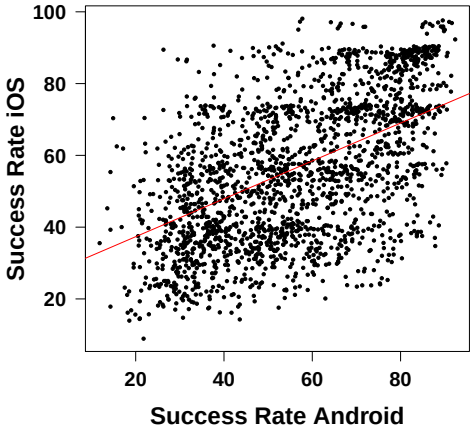


Figure 4.10: The success rates for our 2K app-pairs, where each dot represents an app-pair.

Table 4.6: Descriptive statistics for the iOS and Android (AND) reviews for the app-pairs: Problem Discovery (PD), Feature Request (FR), Non-informative (NI), Positive (P), Negative (N), Neutral (NL), and SR (Success Rate).

ID	Type	Min	Mean	Median	SD	Max	P
PD	iOS	0.00	20.47	15.62	16.65	100.0	0.00
	AND	0.00	21.06	17.54	14.61	100.0	
FR	iOS	0.00	17.50	16.03	10.81	100.0	0.00
	AND	0.00	13.71	12.50	8.88	100.0	
NI	iOS	0.00	62.04	64.95	20.77	100.0	0.00
	AND	0.00	65.23	67.10	17.45	100.0	
P	iOS	0.00	55.62	59.26	20.41	100.0	0.00
	AND	0.00	49.74	51.36	17.64	100.0	
N	iOS	0.00	9.80	6.66	10.07	100.0	0.00
	AND	0.00	7.72	5.74	7.39	100.0	
NL	iOS	0.00	34.57	32.45	14.87	100.0	0.00
	AND	0.00	42.54	41.73	13.97	100.0	
SR	iOS	8.93	55.31	54.68	19.98	98.1	0.27
	AND	11.84	55.97	55.22	18.83	92.4	

views are more among iOS apps whereas *Problem Discovery*, *Non-informative* and *Neutral* are more among Android apps. In addition, the average length of reviews on the iOS platform is 103 characters and 76 characters on the Android platform.

Success Rate. Figure 4.11 shows the success rates for our app-pairs. The app-pairs are arranged based on the difference of their success rates between the two platforms. The far ends on the figure (ellipsed regions) indicate apps which are very

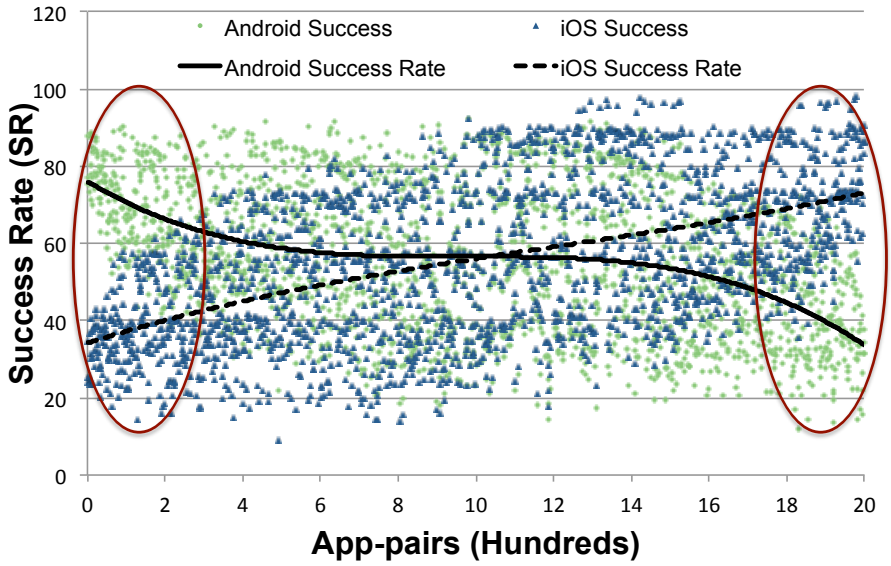


Figure 4.11: Success rates for 2K app-pairs. The green round shape refers to Android apps and the blue triangular shape refers to iOS apps.

successful on one platform but not on the other. The results indicate that *17.4%* (348 app-pairs) of the app-pairs have a difference of *25%* or more in their success rate. The categories with the most successful apps were {*Games, Entertainment, Finance*} on Android and {*Games, Entertainment, Education*} on iOS.

***Finding 3**: The majority of app-pairs are similarly successful on the two app stores. However, 17% have a difference of more than 25% in their success rates.*

Success Rate Differences. The method used to implement the app might affect its success. We randomly selected 30 app-pairs with close success rates (within 5% range) and downloaded their Android source code; we found that 8 of them were implemented using a *hybrid* approach. The hybrid approach uses web technologies such as `HTML`, `CSS`, and `Javascript` to build mobile apps that can run on multiple platforms. We also analyzed 30 app-pairs that are more successful on iOS than Android (difference greater than 20%) and 30 app-pairs that are more successful on Android. We found only 4 in each set used the hybrid approach. In total, we found 16 hybrid apps, which represents 17.7% of 90 app-pairs we inspected. This result is in line with a recent study [210], which found that 15% of android apps are developed using a hybrid approach. The results of this analysis indicate that *some*

app-pairs are equally successful because they use a hybrid approach, meaning they have the same implementation on the two platforms.

Furthermore, to verify our success rate results, we sent emails to all the developers of app-pairs which have a difference of more than 30% in their success rates. We asked if they have noticed the difference and possible reasons that their two apps are not equally successful on both platforms. Although this is a sensitive topic, out of 200 emails, we received 20 responses; all the developers agreed with our findings and were aware of the differences, for example, one developer said: "*our app was by far more successful on iOS than on Android (about a million downloads on iOS and 5k on Android).*"

Overall, developers mentioned (release/update) timing and first impressions can make a large difference in how users perceive the app on the app stores. The variation in success can also be attributed to developers providing more (timely) support on either side. Additionally, app store support and promotional opportunities were mentioned to help developers, e.g., "*Apple ... promote your work if they find it of good quality, this happened to us 4–5 times and this makes a big difference indeed*". Furthermore, some respondents find the Google Play's quick review process helpful to release bug fixes and updates quickly.

4.3.4 Major Complaints (RQ4)

The goal in RQ4 is to understand the nature of user complaints and whether they differ on the two platforms.

Our complaint classifier has, on average, an F-measure of F(SVM) = 0.7. We used the classifier to classify 350,324 (∼350K) problematic reviews for our 2K app-pairs. The results, depicted in Figure 4.12 and Table 4.7, show that the complaints about the apps vary between the two platforms. On average, iOS apps have more critical and post update problems than their Android counterparts, which could be due to Apple regularly forcing developers to migrate their apps to their latest OS and SDK.

On the other hand, Android apps have more complaints related to *App Features* subcategory (i.e., functionality of a feature, compatibility, usability, security, performance), which could be due to device fragmentation on Android. The

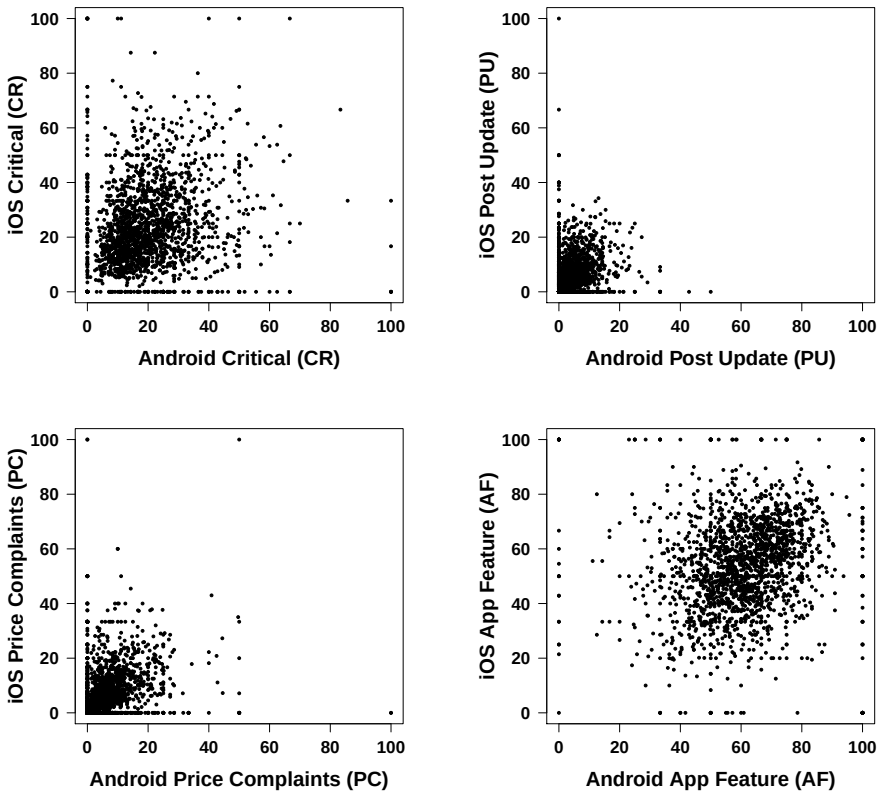


Figure 4.12: The rates of complaints categories for our 2K app-pairs, where each dot represents an app-pair.

wide array of Android devices running with different versions of Android, different screen sizes, and different CPUs can cause security, performance or usability problems. This negative side-effect of fragmentation is also discussed in the literature [86, 94, 111, 215].

Examples of iOS post update problems include users unable to login, features no longer working, loss of information or data, and unresponsive or slow UI. Examples of Android problems include dissatisfaction with a certain functionality, incompatibility with a certain device/OS, network and connection problems.

Table 4.7: Descriptive statistics for the problematic reviews of the app-pairs: Critical (CR), Post Update (PU), Price Complaints (PC), and App Feature (AF).

ID	Type	Min	Mean	Median	SD	Max	P
AF	iOS	0.00	**53.71**	54.29	18.15	100.0	0.00
	AND	0.00	**60.55**	60.92	16.25	100.0	
CR	iOS	0.00	**23.72**	21.05	16.40	100.0	0.00
	AND	0.00	**19.98**	17.65	13.66	100.0	
PU	iOS	0.00	6.08	4.24	7.44	100.0	0.00
	AND	0.00	3.91	2.33	5.17	50.0	
PC	iOS	0.00	7.76	5.00	9.41	100.0	0.00
	AND	0.00	6.70	4.54	8.20	100.0	

***Finding 4**: On average, iOS apps have more critical and post update problems while Android apps have more complaints related to app features and non-functional properties.*

4.4 Discussion

In this section, we discuss several implications for app developers, app stores, and researchers as well as some of the threats to validity of our results.

4.4.1 Implications

For App Developers

Our findings and developer feedback indicate that developers wishing to test new ideas or features find the Android platform more convenient since Google has less formal guidelines and a fast review process compared to Apple's strict guidelines and lengthy review process. Also, paid apps seem to be more successful on the iOS platform; thus, depending on the financial model adopted, developers might want to prioritize building the app for iOS first assuming resources to build the apps for both platforms is limited. Our automated classifiers can be useful for developers to make sense of the feedback in user reviews and handle a large number of reviews some apps receive. Based on our analysis of user reviews, iOS apps suffer from more critical and post update problems. We would suggest that developers focus their attention and resources on fixing such issues first, since many users seem to be easily annoyed by them. On the other hand, the Android platform suffers

more from compatibility, usability and performance issues so this is where Android developers should spend more time on.

For App Stores

App Stores' support and promotional opportunities greatly benefit the developers. We found evidence that Apple rewards well-made apps with promotional opportunities, which could drive the success of the app, an approach that could be adopted by Google Play. Google Play's quick review process is appealing to many developers and something that can be improved on the Apple app store.

For Researchers

Our results indicate that 80% of the top rated apps exist on both the Apple and Google Play app stores. While both platforms are popular and equally important, Android has gained the majority of the attention from the software engineering research community. Our suggestion is to look at both Apple Store and Google Play in future studies to have a more representative coverage. Additionally, examining app-pairs closely can help developers bridge the gap in terms of success for their apps. As recently identified by Nagappan and Shihad [172] one of the obstacles with cross-platform analysis is the lack of a dataset for such apps. Our dataset of 80K app-pairs [165] can help to mitigate this issue; our dataset can be leveraged by other researchers for further cross-platform analysis.

4.4.2 Threats to Validity

Our manual labelling of the reviews to train the classifiers could be a source of internal threat to validity. In order to mitigate this threat, uncertainties were cross-validated and resolved through discussions and refinements between the authors.

As shown in Figure 4.4, the app-pairs detected in our study are a subset of all possible app-pairs. Our study only considers exact matches for app-pairs, which means there exist app-pairs that are not included in our analysis. For instance, an app named *The Wonder Weeks*[34] on iOS has a pair on the Android platform with

[34] https://itunes.apple.com/app/the-wonder-weeks/id529815782?mt=8

the name *Baby Wonder Weeks Milestones*,[35] but not included in our study. While our study has false negatives, our manual validation of 100 randomly selected app-pairs shows that there are no false positives.

In terms of representativeness, we chose app-pairs from a large representative sample of popular mobile apps and categories. With respect to generalizability, iTunes and Google Play are the most popular systems currently, although apps in other app stores could have other characteristics. Regarding replication, all our data is publicly available [165], making the findings of our study reproducible.

4.5 Related Work

Mobile app stores provide app developers with a new and critical channel to extract user feedback. As a result, many studies have been conducted recently through mining and analysis of app store content such as user-reviews [68, 71, 95, 113, 124, 134–136, 140, 147, 156, 181, 183, 185, 208, 211], app descriptions [72, 105, 143, 200], and app bytecode [54, 194, 195, 218].

A number of studies [71, 93, 95, 110, 122] have focused on extracting valuable information for developers from user reviews in app stores. Lacob *et al.* [122] found that 23% of reviews represent feature requests. They proposed a prototype for automatic retrieval of mobile app feature requests from online reviews. Chen *et al.* [71] found that 35% of app reviews contain information that can directly help developers improve their apps. They proposed AR-Miner, a technique to extract the most informative user reviews. First, they filter out non-informative reviews through text analysis and machine learning. Then, they use topic modelling to recognize topics in the reviews classified as informative. Panichella *et al.* [185] proposed an approach built on top of AR-Miner to automatically classify app reviews into different categories. Khalid *et al.* [134, 136] manually analyzed and tagged reviews of iOS apps to identify different issues that users of iOS apps complain about. They studied 6,390 low star-rating reviews for 20 free iOS apps and uncovered 12 types of complaints. They found that functional errors, feature requests and app crashes are the most frequent complaints while privacy and ethical issues, and hidden app costs are the complaints with the most negative impact on

[35] https://play.google.com/store/apps/details?id=org.twisevictory.apps&hl=en

app ratings. Chen *et al.* [72] developed mechanisms to verify the maturity ratings of apps based on app descriptions and user reviews and investigated the possible reasons behind incorrect ratings. They discovered that over 30% of Android apps have unreliable maturity ratings.

Our work, on the other hand, aims at characterizing the differences in mobile app-pairs across two different platforms. To the best of our knowledge, this is the first work to report a large-scale study targeting iOS and Android mobile app-pairs.

4.6 Conclusions

In this work, we present the first quantitative and qualitative study of mobile app-pairs. We mined 80K iOS and Android app-pairs and compared their app-store attributes. We built three automated classifiers and classified 1.7M reviews to understand how user complaints and concerns vary across platforms. Additionally, we contacted app developers to understand some of the major differences in app-pair attributes such as prices, update frequencies, success rates and top rated apps existing only on one platform.

For future work, the testing and analysis of apps across multiple platforms could be explored. While our recent study [88] is a step toward better understanding of it, with the increased fragmentation in devices and platforms, it still remains a challenge to test mobile apps across varying hardware and platforms [172]. Among other directions, the release dates of the app-pairs can be investigated to understand which platform developers target first when they release a new app. Additionally, app features, extracted from app descriptions, can be used to compare on different platforms. Finally, while we combined the stars, ratings and user reviews to measure app success, future studies could explore ways of measuring user retention, number of downloads, user loyalty, recency, or monetization.

Chapter 5

Reverse Engineering iOS Mobile Applications

Summary[36]

As a result of the ubiquity and popularity of smartphones, the number of third party mobile apps is explosively growing. With the increasing demands of users for new dependable applications, novel software engineering techniques and tools geared towards the mobile platform are required to support developers in their program comprehension and analysis tasks. In this work, we propose a reverse engineering technique that automatically (1) hooks into, dynamically runs, and analyzes a given iOS mobile app, (2) exercises its user interface to cover the interaction state space and extracts information about the runtime behaviour, and (3) generates a state model of the given application, capturing the user interface states and transitions between them. Our technique is implemented in a tool called ICRAWLER. To evaluate our technique, we have conducted a case study using six open-source iPhone apps. The results indicate that ICRAWLER is capable of automatically detecting the unique states and generating a correct model of a given mobile app.

5.1 Introduction

According to recent estimations [178], by 2015 over 70 percent of all handset shipments will be smartphones, capable of running mobile apps.Currently, there are over 600,000 mobile apps on Apple's AppStore [21] and more than 400,000 on

[36]This chapter appeared at the 19th IEEE Working Conference on Reverse Engineering (WCRE 2012) [85].

Android Market [19].

Some of the challenges involved in mobile app development include handling different devices, multiple operating systems (Android, Apple iOS, Windows Mobile), and different programming languages (Java, Objective-C, Visual C++). Moreover, mobile apps are developed mostly in small-scale, fast-paced projects to meet the competitive market's demand [137]. Given the plethora of different mobile apps to choose from, users show low tolerance for buggy unstable applications, which puts an indirect pressure on developers to comprehend and analyze the quality of their applications before deployment.

With the ever increasing demands of smartphone users for new applications, novel software engineering techniques and tools geared towards the mobile platform are required [82, 169, 213] to support mobile developers in their program comprehension, analysis and testing tasks [91, 128].

According to a recent study [192], many developers interact with the graphical user interface (GUI) to comprehend the software by creating a mental model of the application. For traditional desktop applications, an average of 48% of the application's code is devoted to GUI [170]. Because of their highly interactive nature, we believe the amount of GUI-related code is typically higher in mobile apps.

To support mobile developers in their program comprehension and analysis tasks, we propose a technique to automatically reverse engineer a given mobile app and generate a comprehensible model of the user interface states and transitions between them. In this work, we focus on native mobile apps for the iOS platform. To the best of our knowledge, reverse engineering of iOS mobile apps has not been addressed in the literature yet.

Our work makes the following contributions:

- A technique that automatically performs dynamic analysis of a given iPhone app by executing the program and extracting information about the runtime behaviour. Our approach exercises the application's user interface to cover the interaction state space;
- A heuristic-based algorithm for recognizing a new user interface state, composed of different UI elements and properties.

- A tool implementing our technique, called ICRAWLER (iPhone Crawler), capable of automatically navigating and generating a state model of a given iPhone app. This generated model can assist mobile developers to better comprehend and visualize their mobile app. It can also be used for analysis and testing purposes (i.e., smoke testing, test case generation).
- An evaluation of the technique through a case study conducted on six different open-source iPhone apps. The results of our empirical evaluation show that ICRAWLER is able to identify the unique states of a given iPhone app and generate its state model accurately, within the supported transitional UI elements.

5.2 Related Work

We divide the related work in three categories: mobile app security testing, industrial testing tools currently available to mobile developers, and GUI reverse engineering and testing.

Mobile App Security Testing. Security testing of mobile apps has gained most of the attention from the research community when compared to other areas of research such as functional testing, maintenance, or program comprehension. Most security testing approaches are based on static analysis of mobile apps [67] to detect mobile malware. Egele *et al.* [84] propose PIOS to perform static taint analysis on iOS app binaries. To automatically identify possible privacy gaps, the mobile app under test is disassembled and a control flow graph is reconstructed from Objective-C binaries to find code paths from sensitive sources to sinks. Extending on PIOS, the same authors discuss the challenges involved in dynamic analysis of iOS apps and propose a prototype implementation of an Objective-C binary analyzer [205]. Interestingly, to exercise the GUIs, they use image processing techniques. This work is closest to ours. However, their approach randomly clicks on an screen area and reads the contents from the device's frame buffer and applies image processing techniques to compare screenshots and identify interactive elements. Since image comparison techniques are known to have a high rate of false positives, in our approach we "programmatically" detect state changes by using a heuristic-based approach.

Industrial Testing Tools. Most industrial tools and techniques currently available for analyzing mobile apps are manual or specific to the application in a way that they require knowledge of the source code and structure of the application. For instance, KIF (Keep It Functional) [34] is an open source iOS integration test framework, which uses the assigned accessibility labels of objects to interact with the UI elements. The test runner is composed of a list of scenarios and each scenario is composed of a list of steps. Other similar frameworks are FRANK [31] and INSTRUMENTS [125]. A visual technology, called SIKULI [39], uses fuzzy image matching algorithms on the screenshots to determine the positions of GUI elements, such as buttons, in order to find the best matching occurrence of an image of the GUI element in the screen image. SIKULI creates keyboard and mouse click events at that position to interact with the element. There are also record and playback tools for mobile apps such as MONKEYTALK [36]. However, using such tools requires application-specific knowledge and much manual effort.

GUI Reverse Engineering and Testing. Reverse engineering of desktop user interfaces was first proposed by Memon *et al.* in a technique called GUI Ripping [159]. Their technique starts at the main window of a given desktop application, automatically detects all GUI widgets and analyzes the application by executing those elements. Their tool, called GUITAR, generates an event-flow graph to capture a model of the application's behaviour and generate test-cases.

For web applications, Mesbah *et al.* [163] propose a crawling-based technique to reverse engineer the navigational structure and paths of a web application under test. The approach, called CRAWLJAX, automatically builds a model of the application's GUI by detecting the clickable elements, exercising them, and comparing the DOM states before and after the event executions. The technique is used for automated test case generation [164] and maintenance analysis [160] in web applications.

Amalfitano *et al.* [49] extend on this approach and propose a GUI crawling technique for Android apps. Their prototype tool, called A2T2, manages to extract models of a small subset of widgets of an Android app.

Gimblett *et al.* [97] present a generic description of UI model discovery, in which a model of an interactive software is automatically discovered through simulating its user actions. Specifically they describe a reusable and abstract API for

user interface discovery.

Further, Chang *et al.* [69] build on SIKULI, the aforementioned tool, to automate GUI testing. They help GUI testers automate regression testing by programming test cases once and repeatedly applying those test cases to check the integrity of the GUI.

Hu *et al.* [119] propose a technique for detecting GUI bugs for Android applications using Monkey [35], an automatic event generation tool. Their technique automatically generates test cases, feeds the application with random events, instruments the VM, and produces log/trace files to detect errors by analyzing them post-run.

To the best of our knowledge, no work has been done so far to reverse engineer Objective-C iPhone apps automatically. Our approach and algorithms are different from the aforementioned related work in the way we track the navigation within the application, retrieve the UI views and elements, and recognize a new state, which are geared towards native iPhone user interfaces.

5.3 Background and Challenges

Here, we briefly describe the relevant iPhone programming concepts [125] required for understanding our approach in Section 5.4.

Objective-C is the primary programming language used to write native iOS apps. The language adds a thin layer of object-oriented and Smalltalk-style messaging to the C programming language. Apple provides a set of Objective-C APIs collectively called *Cocoa*. Cocoa Touch is a UI framework on top of Cocoa. One of the main frameworks of Cocoa Touch is `UIKit`, which provides APIs to develope iOS user interfaces.

The Model-View-Controller design pattern is used for building iOS apps. In this model, the controller is a set of *view controllers* as well as the `UIApplication` object, which receives events from the system and dispatches them to other parts of the system for handling. As soon as an app is launched, the `UIApplication` main function creates a singleton *application delegate* object that takes control. The application delegate object can be accessed by invoking the shared application class method from anywhere in code.

At a minimum, a *window* object and a *view* object are required for presenting the application's content. The window provides the area for displaying the content and is loaded from the main *nib file*.[37] Standard UI elements, which are provided by the `UIKit` framework for presenting different types of content, such as labels, buttons, tables, and text fields are inherited from the `UIView` class. Views draw content in a designated rectangular area and handle *events*.

Events are objects sent to an application to inform it of user actions. Many classes in `UIKit` handle touch events in ways that are distinctive to objects of the class. The application sends these events to the view on which the touch occurred. That view analyzes the events and responds in an appropriate manner. For example, buttons and sliders are responsive to gestures such as a tap or a drag while scroll views provide scrolling behaviour for tables or text views. When the system delivers a touch event, it sends an action message to a target object when that gesture occurs.

View controllers are used to change the UI state of an application. A view controller is responsible for handling the creation and destruction of its views, and the interactions between the views and other objects in the application. The `UIKit` framework includes classes for view controllers such as `UITabBarController`, `UITableViewController` and `UINavigationController`. Because iOS apps have a limited amount of space in which to display content, view controllers also provide the infrastructure needed to swap out the views from one view controller and replace them with the views of another view controller. The most common relationships between source and destination view controllers in an iPhone app are either by using a *navigation controller*, in which a child of a navigation controller pushes another child onto the *navigation stack*, or by presenting a view controller *modally*. The navigation controller is an instance of the `UINavigation Controller` class and used for structured content applications to navigate between different levels of content in order to show a screen flow, whereas the modal view controllers represent an interruption to the current workflow.

Challenges. Dynamic analysis of iOS apps has a number of challenges. Most iOS apps are heavily based on event-driven graphical user interfaces. Simply launching

[37] A nib file is a special type of resource file to store the UI elements in.

Figure 5.1: The Olympics2012 iPhone app going through a UI state transition, after a generated event.

an application will not be sufficient to infer a proper understating of the application's runtime behaviour [205]. Unfortunately, most iOS apps currently do not come with high coverage test suites. Therefore, to execute a wide range of paths and reverse engineer a representative model, an approach targeting iOS apps needs to be able to automatically change the application's state and analyze state changes.

One challenge that follows is defining and detecting a new state of an application while executing and changing its UI. In other words, automatically determining whether a state change has occurred is not that straightforward.

Another challenge, associated with tracking view controllers, revolves around the fact that firing an event on the UI could result in several different scenarios as far as the UI is concerned, namely, (1) the current view controller could go to the next view controller (modally, by being pushed to the navigation stack, or changing to the next tab in a tab bar view controller) or (2) UI element(s) in the current interface could be dynamically added/removed/changed, or (3) the current view controller goes back to the previous view controller (dismissed modally or popped from the navigation stack), or (4) nothing happens. Analyzing each of these scenarios requires a different way of monitoring the UI changes and the navigation stack.

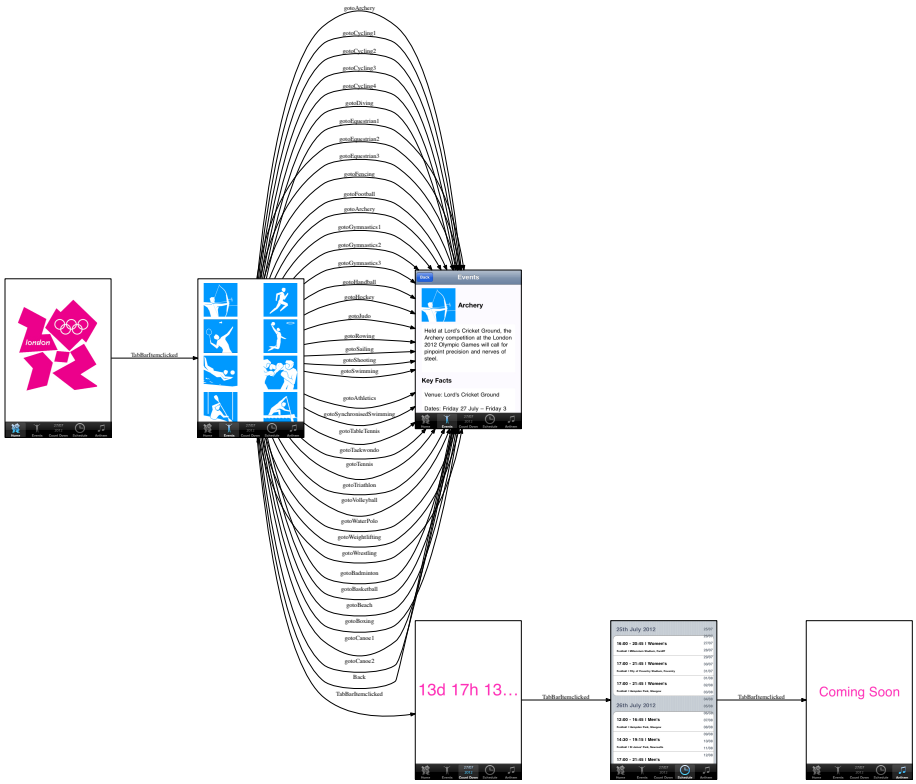


Figure 5.2: The generated state graph of the Olympics2012 iPhone app.

5.4 Our Approach

Our approach revolves around dynamically running a given iOS mobile app, navigating its user interface automatically, and reverse engineering a model of the application's user interface states and transitions between them. Figure 5.1 shows snapshots of an iPhone app (called Olympics2012, used in our case study in Section 5.6) UI state transition after an event. Figure 5.2 shows the automatically generated state graph of the same application. The figure is minimized because of space restrictions, and it is depicted to give an impression of the graph inferred by our approach.

Figure 5.3 depicts the relation between our technique and a given mobile app. The following seven steps outline our technique's operation.

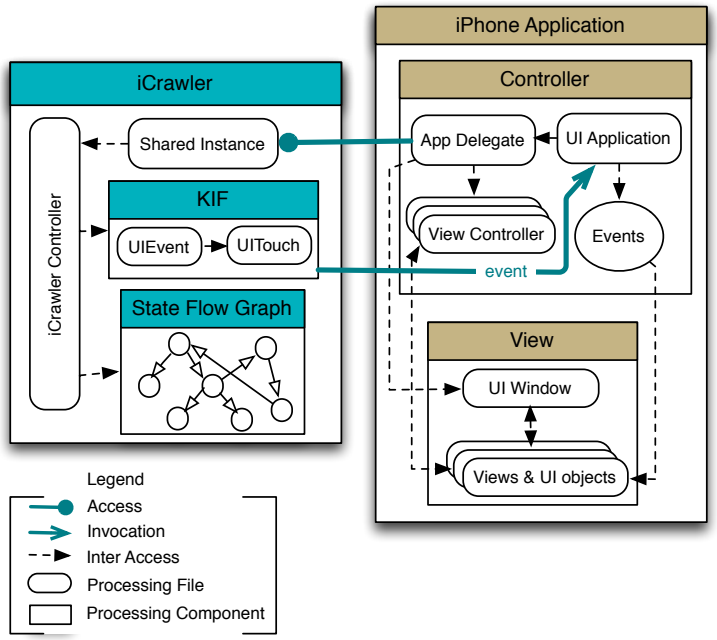


Figure 5.3: Relation between ICRAWLER and a given iPhone app. The right side of the graph shows key components of an iPhone app taken from [125].

Step 1 - Hooking into the Application: As soon as the application is started, our technique kicks in by setting up a shared instance object. As shown in Figure 5.3, we immediately hook into and monitor the application delegate object to identify the initial view controller and infer information about its UI components.

Step 2 - Analyzing UI Elements: After obtaining the initial view controller, we have access to all its UI elements. We keep this information in an array associated to the view controller. Meanwhile, our technique recognizes the different types of UI elements, such as labels, buttons, table cells, and tabs, and identifies which UI elements have an event listener assigned to them.

Step 3 - Exercising UI Elements: To exercise a UI element, we look for an unvisited UI element that has an event listener. As depicted in Figure 5.3, after gathering all the information about the event listeners, the UI object, and its action and target, we generate an event on that element and pass it to

`UIApplication` object, which is responsible for receiving the events and dispatching them to the code for further handling.

Step 4 - Accessing Next View Controller: By observing the changes on the current view controller, we obtain the next view controller and analyze the behaviour. The event could lead to four scenarios: no user interface change, the changes are within the current view controller, going to a new view controller, or going to the previous view controller.

Step 5 - Analyzing New UI Elements: After getting a new view controller, we collect all its UI elements. If the action has resulted in staying in the current view controller, we record the changes on the UI elements.

Step 6 - Comparing UI States: Once we get the new view controller and its UI elements, we need to compare the new state with all the perviously visited unique states. This way, we can determine if the action changes the current state or ends up on a state that has already been analyzed. If the state is not visited before, it is added to the set of unique visited states.

Step 7 - Recursive Call: We recursively repeat from step 3 until no other executable UI elements are left within the view controller and we have traversed all the view controllers.

We further describe our approach in the following subsections.

5.4.1 Hooking into the Application

The process of accessing the initial view controller is different from the rest of the view controllers. Since our goal is to be as nonintrusive and orthogonal to the application's source code as possible, we determine the initial view controller by performing a low-level Objective-C program analysis on the application delegate object. To that end, we employ a number of runtime functions to deduce the initial view controller. We use the Objective-C runtime reference library [38], which provides support for the dynamic properties of the Objective-C language and works with classes, objects, and properties directly. It is effective primarily for low-level debugging and meta-programming. In addition, the Key-Value Coding

(KVC) protocol [37] is used to access UI objects at runtime. The KVC protocol assists in accessing the properties of an object indirectly by key/value, rather than through invocation of an accessor method or as instance variables [37].

Once the application delegate is accessed, we retrieve all the properties of this class and their names. After getting the property names in the application delegate, we call a KVC method to access an instance variable of the initial view controller using the property name string. This way, we are able to identify the type of initial view controller (e.g., `UITabBarController`, `UINavigationController`, just a custom `UIViewController`). This knowledge is required for setting up the initial state.

5.4.2 Analyzing UI Elements

In our approach, a UI *state* includes the current view controller, its properties, accompanied by its set of UI elements. Once we get the view controller, we read all the subviews, navigation items, as well as tool bar items of the view controller in order to record the corresponding UI elements in an array associated to the view controller. Having the required information for a state, we set a global variable to point to the current state throughout the program.

5.4.3 Exercising UI Elements

We fire an event (e.g., a tap) on each unvisited UI element that has an event-listener assigned to it. Since events are handled in different ways for different UI classes, for each UI element type, such as tables, tabs, text views, and navigation bar items, we recognize its type and access the appropriate view. As shown in Figure 5.3, we use KIF's methods to handle the event.

After an element is exercised, we use a delay to wait for the UI to update, before calling the main function recursively. Based on our experience, a 1 second waiting time is enough after firing different event types such as tapping on a table cell or a button, scrolling a table up and down, and closing a view.

```
1 (void)icDismissModalVC:(BOOL) animated {
2   [NSUserDefaults
        standardUserDefaults setBool:YES forKey:@"IC_isDismissed"];
3   // Call the original (now renamed) method
4   self icDismissModalVC:animated;
5 }
```

Figure 5.4: The new method in which we inject code to set the dismissed boolean and then call the original method.

5.4.4 Accessing the Next View Controller

After exercising a UI element, we need to analyze the resulting UI state. An event could potentially move the UI forward, backward, or have no effect at all.

At a low level, going back to a previous view controller in iPhone apps happens either by popping the view controller from the navigation stack or by dismissing a modal view controller. We monitor the navigation stack after executing each event to track possible changes on the stack and thus, become aware of the pop calls. However, being aware of dismissing a modal view needs to be addressed differently. Our approach combines reflection with code injection to track if a dismiss method is called. To that end, we employ the *Category and Extension* [65] feature of Objective-C, which allows adding methods to an existing class without subclassing it or knowing the original classes. We also use a technique called *Method Swizzling* [76], which allows the method implementation of a class to be swapped with another method.

We define a category extension to the `UIViewController` class and add a new method in this category (See Figure 5.4). We then swap a built-in method of the view controller, responsible for dismissing a view controller class, with the new method (See Figure 5.5). The static `+load` method is also added to the category and called when the class is first loaded. We use the `+load` method to swap the implementation of the original method with our replaced method. The `swap` method call swaps the method implementations such that calls to the original method at run-time result in calls to our method defined in the category. As show in Figure 5.4, we also call the original method, which is now renamed. Our method stores a boolean data in the defaults system. The iOS defaults system is available throughout the application, and any data saved in the defaults system will persist through application sessions. Therefore, after a dismiss call occurs, we set

```
1 (void)load {
2   if (self == UIViewController class) {
3     Method originalMethod = class_getInstanceMethod(self, @selector(↩
          dismissModalViewControllerAnimated:));

5     Method replacedMethod = class_getInstanceMethod(self, @selector(↩
          icDismissModalVC:));

7     swap(originalMethod, replacedMethod);
8   }
9 }
```

Figure 5.5: Swapping the original built-in method with our new method in the `+load` function.

the dismissed boolean to `true`. At runtime, each time an action is executed, we check the dismissed boolean in the `NSUserDefaults` object to see if dismiss has occurred. We set this back to `false` if that is the case. This way we are able to track if the event results in going back to a previous view controller to take the proper corresponding action.

A new view controller could be pushed to the navigation stack, presented modally, or be a new tab of a tab bar controller. If the action results in staying in the current view controller, different state changes could still occur such as UI element(s) dynamically being changed/added/removed, or a pop-up message or an action sheet appearing. If we do not notice any changes within the current state, we move further with finding the next clickable UI element. Otherwise, we need to conduct a state comparison to distinguish new states from already visited states. If the state is recognized as a new state, a screen shot of the interface is also recoded.

5.4.5 Comparing States

Another crucial step in our analysis is determining whether the state we encounter after an event is a new UI state. As opposed to other techniques that are based on image-based comparisons [69, 205], in order to distinguish a new state from the previously detected states, we take a programmatic, heuristic-based approach in which we compare the view controllers and all their UI elements of the application before and after the event is executed.

Deciding what constitutes a UI state change is not always that straightforward. For instance, consider when a user starts typing a string to a text field and that

action changes the value of the text field's property, or when the sent button of an email application is enabled as soon as the user starts typing a body of the email. We need a way to figure out if these changes (changing text of a text field/label or enabling a button) should be seen as a new state. To that end, we propose a similarity-based heuristic to emphasize or ignore changes on specific properties of view controllers, their accompanying UI elements, and the elements' properties.

Our state recognition heuristic considers the following properties of view controllers: `class`, `title`, and the number of UI elements. In addition, for each UI element, it considers `class`, `hidden`, `enable`, `target`, and `action`. Although our algorithm can handle as many properties as required, we are interested in these attributes because we believe they are most likely to cause a visible UI state change. We consider a set of distinct weights for each of the aforementioned attributes of a view controller, denoted as $WVC = \{wvc1, wvc2, ..\}$ as well as another set of distinct weights for each of the aforementioned attributes of a UI element as $WE = \{we1, we2, ..\}$. The value of each weight is a number between 0 and 1. All weights have default values that can be overridden by the user's input if required. These default values are obtained for each weight through an experimental trial and error method (discussed in Section 5.6). The similarity, σ, between two UI states is a percentage calculated as follows:

$$\sigma = \left(\frac{\sum_{i=1}^{Size(WVC)} |WVC_i| VC_i + \sum_{j=1}^{Ne} \sum_{k=1}^{Size(WE)} |WE_k| El_j}{Size(WVC) + N_e \times Size(WE)} \right) \times 100$$

where `VC` returns 1 if the property of the two view controllers are equal and 0 otherwise, `Size(WVC)` and `Size(WE)` return the total number of properties considered for a view controller and a UI element respectively. The second part of the summation calculates similarity of each of the elements' properties. `El` returns 1 if the property of the two UI elements is equal and `Ne` is the total number of UI elements. The total summation of the view controllers and elements is divided by the total number of properties.

Algorithm 2 shows our algorithm for checking the similarity of the current state (after an event) with all the visited states. It returns a similar state if one is found among the visited states. As input the algorithm gets two sets of distinct weights

Algorithm 2: State Change Recognition

```
   input  : Set of weights for view controller properties (Wvc)
   input  : Set of weights for UI element properties (We)
   input  : Similarity threshold (τ)
   input  : Set of the unique states visited (VS)
   input  : Current state (cs)
   output: Similar state (s ∈ VS, otherwise nil)
 1 begin
 2     σ ← 0
 3     foreach s ∈ VS do
 4         σ ←
 5         (Wvc(class) × (s.viewController.class ≡ cs.viewController.class) +
 6         Wvc(title) × (s.viewController.title ≡ cs.viewController.title) +
 7         Wvc(elements) × (s.uiElementsCount ≡ cs.uiElementsCount))
 8         foreach e1 ∈ s.uiElementsArray do
 9             e2 ← GETELEMENTATINDEX(cs, e1)
10             σ ← σ + (We(class) × (e1.class ≡ e2.class) + We(hidden) × (e1.hidden
               ≡ e2.hidden) + We(enable) × (e1.enable ≡ e2.enable) + We(target) ×
               (e1.target ≡ e2.target) + We(action) × (e1.action ≡ e2.action))
11         attributes ← Size(Wvc) + (s.uiElementsCount × Size(We))
12         if ((σ/attributes) × 100) >= τ then
13             return s
14     return nil
```

for view controller and UI element, a similarity threshold (τ), the set of unique states visited so far, and the current state.

For each visited state (line 3), we calculate the similarity of the two states by adding the similarity of the two view controllers' classes, titles and the number of UI elements (line 7). Then for each UI element in a visited state (line 8), the corresponding UI element in the current state (line 9) is retrieved and their similarity is calculated. Finally, we divide the similarity by the total number of attributes, which are considered so far, calculate the percentage (line 12) and compare it to the threshold. The algorithm assumes the two interfaces to be equivalent if the calculation of the aforementioned weight-based attributes are more than or equal to τ. In other words, we consider two UI states equal, if they have the same view controller, title, set of UI elements, with the same set of selected properties, and the same event listeners.

5.4.6 State Graph Generation

To explore the state space, we use a depth-first search algorithm and incrementally create a multi-edge directed graph, called a state-flow graph [163], with the nodes representing UI states and edges representing user actions causing a state transition.

5.5 Tool Implementation: ICRAWLER

We have implemented our approach in a tool called ICRAWLER. ICRAWLER is implemented in Objective-C using Xcode 3. We use a number of libraries as follows.

DCINTROSPECT [30] is a library for debugging iOS user interfaces. It listens for shortcut keys to toggle view outlines and print view properties as well as the action messages and target objects, to the console. We have extended DCINTROSPECT in a way to extract a UI element's action message, target object, it's properties and values. We further use our extension to this library to output all the reverse engineered UI elements' properties within one of our output files.

To generate an event or insert textual input, we use and extend the KIF framework [34]. At runtime, ICRAWLER extracts UI elements with event-listeners assigned to them and collects information about the action message and target object of each UI elements. By recognizing the type of a UI element, ICRAWLER gains access to its appropriate view. Then it uses KIF's internal methods to generate an event on the view.

At the end of the reverse engineering process, the state graph is transformed into an XML file using XSWI [42], which is a standalone XML stream writer implemented in Objective-C.

The output of ICRAWLER consists of the following three items: (1) an XML file, representing a directed graph with actions as edges and states as nodes. (2) screenshots of the unique states, and (3) a log of all the reverse engineered UI elements (including their properties, values, actions and targets), generated events and states.

Table 5.1: Experimental objects.

ID	Exp. Object	Resource
1	Olympics2012	https://github.com/Frahaan/2012-Olympics-iOS--iPad-and-iPhone--source-code
2	Tabster	http://developer.apple.com/library/ios/#samplecode/Tabster/Introduction/Intro.html#
3	TheElements	http://developer.apple.com/library/ios/#samplecode/TheElements/Introduction/Intro.html#
4	Recipes & Printing	http://developer.apple.com/library/ios/#samplecode/Recipes_+_Printing/Introduction/Intro.html#
5	NavBar	http://developer.apple.com/library/ios/#samplecode/NavBar/Introduction/Intro.html#
6	U Decide	http://appsamuck.com/day12.html

5.6 Empirical Evaluation

To assess the effectiveness of our reverse engineering approach, we have conducted a case study using six open-source iPhone apps.

We address the following research questions in our evaluation:

RQ1 Is ICRAWLER capable of identifying unique states of a given iPhone application correctly?

RQ2 How complete is the generated state model in terms of the number of edges and nodes?

RQ3 How much manual effort is required to set up and use ICRAWLER? What is the performance of ICRAWLER?

5.6.1 Experimental Objects

We include six open-source experimental objects from the official Apple sample code, Guithub, and other online resources. Table 5.1 shows each objects's ID, name, and resource. Table 5.2 presents the characteristics of these applications in terms of their size and complexity. We use XCODE STATISTICIAN[38] for collecting metrics such as the number of header and main files, lines of code (LOC) and

[38] http://xcode-statistician.mac.informer.com/

Table 5.2: Characteristics of the experimental objects.

ID	.m/.h Files	LOC (Objective-C)	Statements (;)	Widgets
1	22	2,645	1,559	398
2	21	1,727	286	14
3	28	2,870	690	21
4	23	2,127	508	7
5	20	1,487	248	10
6	13	442	162	15

statements. The table also shows the number of UI widgets within each application. The UI widget is a UI element, such as a tab bar view with all of its tab icons, a table view with all of its cells, a label or a button. The number of UI widgets is collected through ICRAWLER's output file, which logs all the UI elements and their properties.

5.6.2 Experimental Design

In order to address RQ1, we need to compare unique states generated by ICRAWLER to the actual unique states for each application. As mentioned before, ICRAWLER identifies the unique states through Algorithm 3 and keeps the screen-shots of the unique states in a local folder. To form a comparison baseline, we manually run and navigate each application and count the unique states and compare that with the output of ICRAWLER.

To assess the ICRAWLER's generated state model (RQ2), we also require to form a baseline of the actual number of edges (i.e. user's actions that change the states) and states (unique and repetitive) to compare with the ICRAWLER's state model. Therefore, we manually run and navigate each application and count the edges and the states. Note that there are currently no other similar tools available to compare ICRAWLER's results against.

In order to address RQ3, we measure the time required to set up ICRAWLER and employ it to each of the given iPhone apps. The following series of manual tasks are required before ICRAWLER can start the analysis:

- The ICRAWLER framework should be added to the application's project un-

der analysis.

- In order to enable ICRAWLER to access the delegating application object, the ICRAWLER's initialization line of code should be added to the built-in method, `application: didFinishLaunchingWithOptions:`.

- Finally, a preprocessor flag (`RUN_ICRAWLER`) needs to be added to the created Xcode target.

Further to investigate the performance of ICRAWLER for each application under test, we measure the time between calling ICRAWLER and when ICRAWLER finishes its job.

As we mentioned earlier, we obtain default values for the threshold and similarity weights by an experimental trial and error method for each of the applications. The best values that we have observed are: threshold (%70); weights include: view controller's `class` (0.8), `title` (0.8), and number of UI elements (0.8); UI element's `class` (0.7), `hidden` (0.7), `enable` (0.7), `target` (0.7), and `action` (0.7). These are also the values used in our evaluation, for all the experimental objects.

5.6.3 Results

Setting up ICRAWLER and utilizing it for a given iPhone app takes 10 minutes on average. The results of our study are shown in Table 5.3. The table shows the number of `Unique States`, `Total States`, and `Edges` counted manually and by ICRAWLER. Further, the total number of `Generated Events` and `Total Time` for ICRAWLER are presented. We should note that the total time depends on the application and includes the delay (1 sec) we use after each action. The number of generated events is different from the number of detected edges. The events include all the user actions, while the edges are only those actions that result in a state change (including back-ward edges). For instance, scrolling a table or a view up and down counts as an event while it is not an edge in the state model. Another example, related to our state comparison algorithm, is a label that changes after executing a button, which ICRAWLER does not consider as a new state.

Below, we describe some of the results in Table 5.3.

Table 5.3: Results.

ID	Manual			iCrawler				
	Unique States	Tot. States	Edges	Unique States	Total States	Edges	Gen. Events	Total Time (Sec)
1	6	81	43	6	81	43	85	88
2	11	16	17	9	12	11	18	18
3	6	16	15	6	16	15	27	29
4	6	13	10	3	5	4	8	10
5	8	14	13	3	5	4	7	9
6	2	13	2	2	13	2	12	13

The `Olympics2012` (#1) application provides information about 38 sports in the Olympics 2012 as well as a timetable and a count down (See Figure 5.1 and Figure 5.2). According to Table 5.3, iCrawler is capable of identifying the correct number of uniques states, total states, and edges within this application. The events include tapping on a tab bar item, scrolling up/down a view, scrolling up/down a table, tapping on a backward/forward button and tapping on a button which flips the view. The number of user actions, i.e., generated events, is 85 while the number of edges is 43 (including a back-ward edge). This is because user actions such as scrolling are not changing states and as a result they are not counted as edges. The number of uniques states is 6 while the number of total states is 81. This is because there are 38 buttons in this application which lead to a same UI state while presenting different data for 38 types of sports.

Events within `Tabster` (#2) include tapping on a tab bar item, scrolling up-/down a table, tapping on a table cell, tapping on a backward/forward button, tapping on a dismiss/present button and writing a text. When exercising UI elements which require text input through keyboard, we used a dummy string based on the keyboard type e.g., numeric, alphanumeric, url or email address input. As it is shown in Table 5.3, our approach is able to identify 11 edges and 9 uniques states. However `Tabster` has the tab bar view with a "more page" feature and iCrawler supports an ordinary tab bar view (without the "more" feature) at this time. As a result, there is a difference between the number of uniques states and edges in baseline and iCrawler.

Actions within `TheElements` (#3) application include tapping on a tab bar item, scrolling up/down a table, tapping on a table cell, tapping on a back-ward/forward

button and tapping on a button which flips the view. ICRAWLER is successfully able to cover the states and edges of `TheElements`. Here, we disabled a button, which closes the application and forwards the user to the AppStore.

The `Recipes & Printing` application (#4) browses recipes and has the ability to print the browsed recipes. Here, the difference between manual and ICRAWLER results in Table 5.3 is due to ignoring the states and actions involved with printing.

For tables, one could think of different strategies to take: (1) generate an event on each and every single table cell, (2) randomly click on a number of table cells (3) generate an event on the first table cell. In our technique, once ICRAWLER encounters a table view, it scrolls down and up to ensure the scrolling action works properly and it does not cause to any unwanted crashes, e.g., by having a specific character in an image's url and trying to load the image on a table cell. ICRAWLER then generates an event on the first row and moves forward. This works well for table cells that result to the same next view. However, there are cases in which table cells lead to a different view. `NavBar` (#5) is such a case. There are five different table cells within this application, which go to different UI states. Thus we witness a difference between the number of edges or states counted manually and by ICRAWLER. This is a clear empirical evidence suggesting that we need to improve our table cell analysis strategy.

5.6.4 Findings

The results of our case study show that ICRAWLER is able to identify the unique states of a given iPhone app and generate its state model correctly, within the supported UI elements and event types. Generally, it takes around 10 minutes to set up and use ICRAWLER. The performance of ICRAWLER is acceptable. For the set of experimental objects, the minimum analysis time was 9 seconds (5 states, 4 edges, 7 events) and the maximum was 88 seconds (81 states, 43 edges, 85 events).

5.7 Discussion

Limitations. There are some limitations within our current implementation of the approach. Although it is minimal, the users still need to complete a few tasks

to set up ICRAWLER within their applications manually. There are also some UI elements such as the tool bar, slider, page control, and search bar, which are not supported currently. In addition, while ICRAWLER currently supports the most common gestures in iOS apps such as tapping on a UI element, inserting text, and scrolling views, there is no support yet for other advanced gestures such as swiping pages and pinching (e.g., zooming in and out images).

Threats to Validity. The fact that we form the comparison baselines manually could be a threat to internal validity. We did look for other tools to compare our results against, without success. Manually going through the different applications to create baselines is labour intensive and potentially subject to errors and author's bias. We tried to mitigate this threat by asking two other students to create the comparison baselines.

Additionally, the independent variables of weights and threshold within our state recognition algorithm have a direct effect on our dependent variables such as number of unique states and edges. As a result, choosing other values for these independent variables rather than our default values, could result in difference in the outcome. As mentioned in the evaluation section, we chose these optimal values through a series of trial and error experiments.

In our attempt to gather the experimental objects, we noticed that there is a small collection of open-source iPhone apps available online – note that we could not use applications available in AppStore for our experiment since we needed access to their source code. Even though, this made it difficult to select applications that reflect the whole spectrum of different UI elements in iPhone apps, we believe the selected objects are representative of the type of applications ICRAWLER can reverse engineer. However, we acknowledge the fact that, in order to draw more general conclusions, more mobile apps are required.

Applications. There are various applications for our technique. First of all, our technique enables automatic interaction with the mobile app. This alone can be seen as performing smoke testing (e.g., to detect crashes). In addition, the state model inferred can be used for automated test case generation. Further, using the model to provide a visualization of the state space supports developers to obtain a better understanding of their mobile apps. The approach can be extended to

perform cross-platform testing [161], i.e., whether an application is working correctly on different platforms such as iOS and Android, by comparing the generated models. Finally, other application areas could be in performance and accessibility testing of iOS apps.

5.8 Conclusions

As smartphones become ubiquitous and the number of mobile apps increases, new software engineering techniques and tools geared towards the mobile platform are required to support developers in their program comprehension, analysis, and testing tasks.

In this work, we presented our reverse engineering technique to automatically navigate a given iPhone app and infer a model of its user interface states. We implemented our approach in ICRAWLER, which is capable of exercising and analyzing UI changes and generate a state model of the application. The results of our evaluation, on six open source iPhone apps, point to the efficacy of the approach in automatically detecting unique UI states, with a minimum level of manual effort required from the user. We believe our approach and techniques have the potential to help mobile app developers increase the quality of iOS apps.

There are several opportunities in which our approach can be enhanced and extended for future research. The immediate step would be to extend the current version of ICRAWLER to support the remaining set of UI elements within `UIKIT` such as the tool bar, slider, page control, and search bar. Other directions can use this technique for smoke testing of iPhone apps as well as generating test cases from the inferred state model. Furthermore, ICRAWLER can be extended to support iPad apps as well as reverse engineering analysis at the binary level. This is beneficial as AppStore distributes binary code of the applications, and this would be interesting to apply automated testing to any application disregarding having accessibility to its source code.

Chapter 6

Detecting Inconsistencies in Multi-Platform Mobile Apps

Summary[39]

Due to the increasing popularity and diversity of mobile devices, developers write the same mobile app for different platforms. Since each platform requires its own unique environment in terms of programming languages and tools, the teams building these multi-platform mobile apps are usually separate. This in turn can result in inconsistencies in the apps developed. In this work, we propose an automated technique for detecting inconsistencies in the same native app implemented for iOS and Android platforms. Our technique (1) automatically instruments and traces the app on each platform for given execution scenarios, (2) infers abstract models from each platform execution trace, (3) compares the models using a set of code-based and GUI-based criteria to expose any discrepancies, and finally (4) generates a visualization of the models, highlighting any detected inconsistencies. We have implemented our approach in a tool called CHECKCAMP. CHECKCAMP can help mobile developers in testing their apps across multiple platforms. An evaluation of our approach with a set of 14 industrial and open-source multi-platform native mobile app-pairs indicates that CHECKCAMP can correctly extract and abstract the models of mobile apps from multiple platforms, infer likely mappings between the generated models based on different comparison criteria, and detect inconsis-

[39]This chapter appeared at the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015) [88].

tencies at multiple levels of granularity.

6.1 Introduction

Recent industry surveys [50, 51] indicate that mobile developers are mainly interested in building *native apps*, because they offer the best performance and allow for advanced UI interactions. Native apps run directly on a device's operating system, as opposed to web-based or hybrid apps, which run inside a browser.

Currently, iOS [21] and Android [19] native mobile apps[40] dominate the app market each with over a million apps in their respective app stores. To attract more users, implementing the same mobile app across these platforms has become a common industry practice. Ideally, a given mobile app should provide the same functionality and high-level behaviour on different platforms. However, as found in our recent study [86], a major challenge faced by industrial mobile developers is to keep the app consistent across platforms. This challenge is due to the many differences across the platforms, from the devices' hardware, to operating systems (e.g., iOS/Android), and programming languages used for developing the apps (e.g., Objective-C/Java). We also found that developers currently treat the mobile app for each platform *separately* and *manually* perform screen-by-screen comparisons, often detecting many cross-platform inconsistencies [86]. This manual process is, however, tedious, time-consuming, and error-prone.

In this work, we propose an automated technique, called CHECKCAMP (Checking Compatibility Across Mobile Platforms), which for the same mobile app implemented for iOS and Android platforms (1) instruments and generates traces of the app on each platform for a set of user scenarios, (2) infers abstract models from the captured traces that contain code-based and GUI-based information for each pair, (3) formally compares the app-pair using different comparison criteria to expose any discrepancies, and (4) produces a visualization of the models, depicting any detected inconsistencies. Our work makes the following main contributions:

- A technique to capture a set of run-time code-based and GUI related metrics used for generating abstract models from iOS and Android app-pairs;

[40] In this work, we focus on native apps; henceforth, we use the terms 'mobile app' or simply 'app' to denote 'native mobile app'.

- Algorithms along with an effective combination of mobile specific criteria to compute graph-based mappings of the generated abstract models targeting mobile app-pairs, used to detect cross-platform app inconsistencies;
- A tool implementing our approach, called CHECKCAMP, which visualizes models of app-pairs, highlighting the detected inconsistencies. CHECKCAMP is publicly available [27];
- An empirical evaluation of CHECKCAMP through a set of seven industrial and seven open-source iOS and Android mobile app-pairs.

Our results indicate that CHECKCAMP can correctly extract abstract models of the app-pairs to infer likely mappings between the generated abstract models based on the selected criteria; CHECKCAMP also detects 32 valid inconsistencies in the 14 app-pairs.

6.2 Pervasive Inconsistencies

A major challenge faced by industrial mobile developers is to keep the app consistent across platforms. This challenge and the need for tool support emerged from the results of our qualitative study [86], in which we interviewed 12 senior app developers from nine different companies and conducted a semi-structured survey, with 188 respondents from the mobile development community.

In this work, to identify the most pervasive cross-platform inconsistencies between iOS and Android mobile app-pairs, we conducted an exploratory study by interviewing three industrial mobile developers, who actively develop apps for both platforms. The following categories and examples are extracted from the interviews as well as a document shared with us by the interviewees, containing 100 real-world cross-platform mobile app inconsistencies. Ranked in the order of impact on app behaviour, the most pervasive inconsistency categories are as follows:

Functionality: The highest level of inconsistencies is missing functionality; e.g., "*Notes cannot be deleted on Android whereas iOS has the option to delete notes.*" Or "*After hitting send, you are prompted to confirm to upload – this prompt is missing on iOS.*"

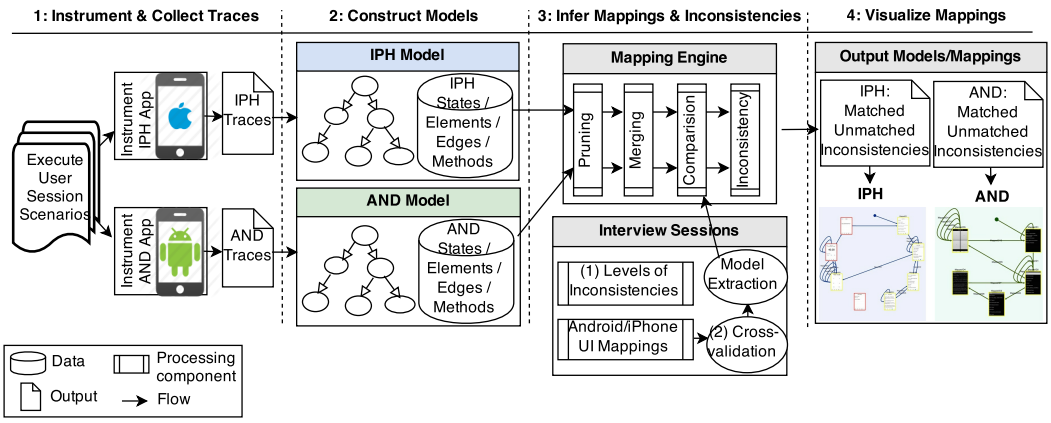


Figure 6.1: The overview of our technique for behaviour checking across mobile platforms.

Data: When the presentation of any type of data is different in terms of order, phrasing/wording, imaging, or text/time format; e.g., "*Button on Android says 'Find Events' while it should say 'Find' similar to iOS.*"

Layout: When a user interface element is different in terms of its layout such as size, order, or position; e.g., "*Android has the 'Call' button on the left and 'Website' on the right - iPhone has them the other way around.*"

Style: The lowest level of inconsistency pertains to the user interface style; i.e., colour, text style, or design differences, e.g., "*iOS has Gallery with a blue background while Android has Gallery with a white background*".

We propose an approach that is able to automatically detect such inconsistencies. Our main focus is on the first two since these can impact the behaviour of the apps.

6.3 Approach

Figure 6.1 depicts an overview of our technique called CHECKCAMP. We describe the main steps of our approach in the following subsections.

6.3.1 Inferring Abstract Models

We build separate dynamic analyzers for iOS and Android, to instrument the app-pair. For each app-pair, we execute the same set of user scenarios to exercise similar actions that would achieve the same functionality (e.g., reserving a hotel or creating a Calendar event). As soon as the app is started, each analyzer starts by capturing a collection of traces about the runtime behaviour, UI structures, and method invocations. Since the apps are expected to provide the same functionality, our intuition is that their traces should be mappable at an abstract level. The collected traces from each app are used to construct a *model*:

Definition 2 (Model). *A Model μ for a mobile app M is a directed graph, denoted by a 4-tuple $< \alpha, \eta, V, E >$ where:*

1. *α is the initial edge representing the action initiating the app (e.g., a tap on the app icon).*
2. *η is the node representing the initial state after M has been fully loaded.*
3. *V is a set of vertices representing the states of M. Each $\upsilon \in V$ represents a unique screen of M annotated with a unique ID.*
4. *E is a set of directed edges (i.e., transitions) between vertices. Each (υ_1, υ_2) $\in E$ represents a clickable c connecting two states if and only if state υ_2 is reached by executing c in state υ_1.*
5. *μ can have multi-edges and be cyclic.*

Definition 3 (State). *A state $s \in V$ represents the user interface structure of a single mobile app screen. This structure is denoted by a 6-tuple, $< \gamma, \theta, \tau, \lambda, \Omega, \delta >$, where γ is a unique state ID, θ is a classname (e.g., name of a `View Controller` in iOS or an `Activity` in Android), τ is the title of the screen, λ is a screenshot of the current screen, Ω is a set of user interface elements with their properties such as type, action, label/data, and δ is a set of auxiliary properties (e.g., tag, distance) used for mapping states.*

Definition 4 (Edge). *An edge $e \in E$ is a transition between two states representing user actions. It is denoted by a 6-tuple, $< \gamma, \theta, \tau, \lambda, \Omega, \delta >$, where γ is a unique edge ID, θ is a source state ID, τ is a target state ID, λ is a list of methods invoked when the action is triggered, Ω is a set of properties of a touched element*[41] *(i.e. type, action, label/data) and δ is a set of auxiliary properties (e.g., tag, distance) used for mapping purposes.*

iOS App Model Inference

In iOS, events can be of different types, such as touch, motion, or multimedia events. We focus on touch events since the majority of actions are of this type. A touch event object may contain one or more finger gestures on the screen. It also includes methods for accessing the UI view in which the touch occurs. We track the properties of the UI element that the touch event is exercised on. To capture this information, we employ the *Category and Extension* [65] feature of Objective-C, which allows adding methods to an existing class without subclassing it or knowing the original classes. We also use a technique called *Method Swizzling* [76], which allows the method implementation of a class to be swapped with another method. To that end, we define a category extension to the `UIApplication` class and a new method in this category. We then swap a built-in method, responsible for sending an event, with the new method. The `swap` method call modifies the method implementations such that calls to the original method at runtime result in calls to our method defined in the category. Additionally, we capture the invoked method calls after an event is fired. We use aspects to dynamically hook into methods and log method invocations. Once an *event* is fired at runtime, all the invoked methods and their classes are traced and stored in a global dataset.

For each event fired, we add an edge to the model. Figure 6.2 shows an *edge* object of an iPhone app (called MTG, used in our evaluation in Section 6.5) including its captured touched element and invoked methods.

To construct the model, we need to capture the resulting state after an event is triggered. In iPhone apps, a UI state includes the current visible view controller, its properties, accompanied by its set of UI elements. We use a delay to wait for the UI

[41] A touched element is the UI element which has been exercised when executing a scenario (e.g., a cell in a table, a button, a tab in a tab bar).

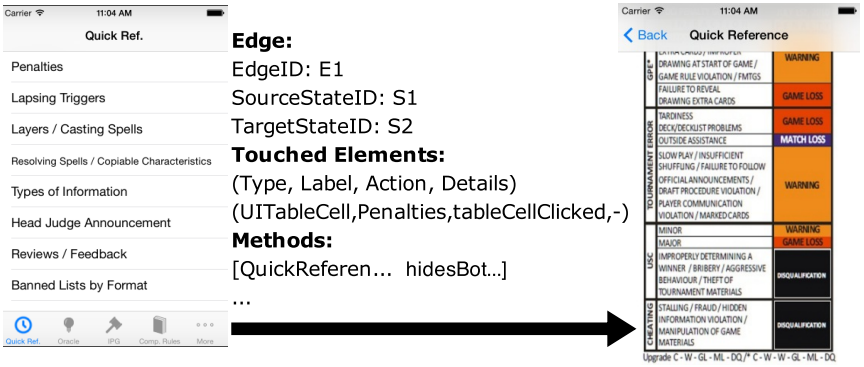


Figure 6.2: An *edge* object of MTG iPhone app with its touched element and methods.

to update properly after an event, before triggering another event on a UI element. Based on our empirical analyses, a two second waiting time is enough for most iOS apps. An event could potentially move the UI forward, backward, or have no effect at all. If the action results in staying in the current view controller, different state mutations could still occur. For instance, UI element(s) could dynamically be changed/added/removed, or the main view of the view controller be swapped and replaced by another main view with a set of different UI element(s). At a low level, moving the UI forward or backward loads a view controller in iPhone apps. Similar to capturing properties of each edge, our approach for capturing UI-structure of each state, combines reflection with code injection to observe *loading view controller* methods.

Once we obtain a reference to the view controller, our approach takes a snapshot of the state and captures all the UI *element* objects in an array associated to the view controller, such as tables with cells, tab bars with tab items, tool bar items, navigation items (left, right, or back buttons), and it loops through all the subviews (e.g., labels, buttons) of the view controller. For each of them, we create an *element* object with its `ID`, `type`, `action`[42], `label`, and `details`.

Figure 6.3 shows a snapshot of a state in the MTG iPhone app including its UI *element* objects. For instance, the top left button in Figure 6.3 has 'UIButton' as `type`, '1' as `label`, 'button1Pressed' as `action` (the event handler). We set `details` for extra information such as the number of cells in a list. Using this

[42] `action` pertains to the event handler, representing the method that will handle the event.


```
1 State(ID,ClassName,Title,#Elements)
2 (S4,DecklistCounterController,-,8)

4 UIElements
5 (Type,Label,Action,Details)
6 (UIButton,1,button1Pressed,-)
7 (UIButton,2,button2Pressed,-)
8 (UIButton,3,button3Pressed,-)
9 (UIButton,4,button4Pressed,-)
10 (UILabel,Total: 0 (+0),-,-)
11 (UIButton,Reset,resetPressed,-)
12 (UIButton,Undo,undoPressed,-)
13 (UITabBar,-,itemClicked,5tabs)
```

Figure 6.3: A snapshot of a *state* in MTG iPhone app with its captured UI *element* objects.

information, we create a state node in the model.

Android App Model Inference

At a high-level, our Android dynamic analyzer intercepts method calls executed while interacting with an app and captures UI information (state) upon the return of these methods. Similar to iOS, Android has different types of events. In our approach, we focus on user-invoked events since they contribute to the greatest changes in the UI and allow the app to progress through different states. These types of events get executed when a user directly interacts with the UI of an app, for instance by clicking a button or swiping on the screen. When a user interacts with a UI element, the associated event listener method is invoked, and the element is passed as one of its arguments. To create a new edge in our model, we inspect these arguments and extract information about the UI element that was interacted with by the user. This inspection also allows us to separate user-invoked events from other types, by checking whether the method argument was a UI element such as a button or table that the user can interact with. We compare the argument against the *andoird.widget* package [1], which contains visual UI elements to be used in apps.

In our android analyzer, a UI state includes the current visible screen, its properties, accompanied by its set of UI elements. When an executed method returns, we use the activity that called the method to retrieve information about the state of the UI. To access the UI layout of the current view, we use a method provided by the Android library called *getRootView* [1]. This method returns a `ViewGroup`

object, which is a tree-like structure of all the UI elements present in the current screen of the app. We traverse this tree recursively to retrieve all the UI elements. Additionally, we capture some unique properties of the UI elements such as labels for `TextViews` and `Buttons`, and number of items for `ListViews`. These properties are used during the mapping phase to compare iOS and Android states at a lower level.

6.3.2 Mapping Inferred Models

Next, we analyze each model-pair to infer likely mappings implied by the states and edges through a series of phases. Prior to the *Mapping* phase, two preprocessing steps are required namely *Pruning* and *Merging*.

Pruning

The first step in our analysis, is to prune the graph obtained for each platform, in order merge duplicate states. This step is required as our dynamic analyzers capture any state we encounter after an event is fired without checking if it is a unique state or a duplicate state. This check can be carried out either separately in each analyzer tool or once in the mapping phase. Having it in the mapping phase ensures that the pruning procedure is consistent across platforms. Identifying a new state of a mobile app while executing and changing its UI is challenging. In order to distinguish a new state from previously detected states, we compare the state nodes along with their properties, as shown in Algorithm 3.

As input, Algorithm 3 takes all *States* and *Edges*, obtained from the graph (G), and outputs a pruned graph (P). We loop through all the states captured (line 4), and compare each state with the rest of state space (line 6) based on their classes and number of UI elements (line 8). Next, we proceed by checking their UI elements (line 10) for equivalency of `types` and `actions` (line 12). Thus, data changes do not reflect a unique state in our algorithm. In other words, two states are considered the same if they have the same class and set of UI elements along with their respective properties. Detected duplicate states are removed (line 18) and the source and target state IDs for the edges are adjusted accordingly (line 19).

Algorithm 3: Pruning a Given Model

```
   input  : State Graph (G) of a Given Model (M)
   output: Pruned State Graph (P)
 1 begin
 2     S ← GETVERTICES(G)
 3     E ← GETEDGES(G)
 4     foreach i = 0, i <COUNT(S), i++ do
 5         s1 ← S[i]
 6         foreach j = i+1, j <COUNT(S), j++ do
 7             s2 ← S[j]
 8             if s1(class) ≡ s2(class) &
               s1(#elements) ≡ s2(#elements) then
 9                 elFlag ← TRUE
10                 foreach e1 ∈ s1.Elements do
11                     e2 ← GETELEMENTATINDEX(s1, e1)
12                     if e1.type ≠ e2.type ||
                       e1.action ≠ e2.action then
13                         elFlag ← FALSE
14                         break
15                     end
16                 end
17                 if elFlag then
18                     REMOVEDUPLICATESTATE(S,s2)
19                     UPDATEEDGES(E,s1,s2)
20                 end
21             end
22         end
23     end
24     return P(S,E)
25 end
```

Merging

Platform-specific differences that manifest in our models are abstracted away in this phase. This step is required since such irrelevant differences can occur frequently across platforms. For instance, the iPhone app may offer *More* as an option in its tab controller which is different from the Android app. If the iPhone app has more than five items, the tab bar controller automatically inserts a special view controller (called the *More view controller*) to handle the display of additional items. The More view controller lists the additional view controllers in a table, which appears automatically when it is needed and is separate from custom content. Thus, our approach merges the *More* state with the next state (view con-

Algorithm 4: Mapping two (iOS & Android) Models

```
   input  : iPhone State Graph (IG)
   input  : Android State Graph (AG)
   output: IG with Mapping Properties (MIG)
   output: AG with Mapping Properties (MAG)
 1 begin
 2     IS ← GETVERTICES(IG)
 3     AS ← GETVERTICES(AG)
 4     IE ← GETEDGES(IG)
 5     AE ← GETEDGES(AG)
 6     edgePairs[0] ← INSERTEDGEPAIR(IE[0], AE[0])
 7     foreach i = 0, i <COUNT(edgePairs), i++ do
 8         pair ← edgePairs[i]
 9         if NOTMAPPED(pair) then
10             s1 ← GETSTATE(IS,pair[iphTrgtId])
11             s2 ← GETSTATE(AS,pair[andTrgtId])
12             iphEdges ← GETOUTGOINGEDGES(s1,IE)
13             andEdges ← GETOUTGOINGEDGES(s2,AE)
14             /*Find closest edge-pairs*/
15             nextPairs ← FINDEDGEPAIRS(iphEdges,andEdges)
16             SETSTATEMAPPINGPROPERTIES(s1,s2)
17             SETEDGEMAPPINGPROPERTIES(nextPairs)
18         end
19         foreach j = 0, j <COUNT(nextPairs),j++ do
20             edgePairs[i+j+1] ← INSERTEDGEPAIR(nextPairs[j])
21         end
22     end
23     return (MIG,MAG)
24 end
```

troller) to abstract away iPhone differences that are platform-specific and as such irrelevant for our analysis. Similarly, the Android app may offer an option *Menu* panel to provide a set of actions. The contents of the options menu appear at the bottom of the screen when the user presses the *Menu* button. When a state is captured on Android and then the option *Menu* is clicked, our approach merges the two states together to abstract away Android differences. Other differences such as Android's hardware back button vs. iPhone's soft back button are taken into account in our graph representations.

Mapping

The collected code-based (e.g., `classname`) and GUI-based (e.g., screen `title`) data for states and edges are used in this phase to map the two models, as shown in Algorithm 4. As input, Algorithm 4 takes iPhone (IG) and Android (AG) graphs, produced after the pruning and merging phases, and outputs those models with a set of computed auxiliary mapping properties for their states and edges (MIG and MAG). The algorithm operates on the basis of the following assumptions (1) the model of an app starts with an initial edge that leads to an initial state and (2) conceptually, both models start with the same initial states. An array, *edgePairs*, holds the initial iPhone and Android edges (line 6) and other edge-pairs are inserted through the main loop (line 20). To find the edge-pairs, we first obtain the initial iPhone and Android states (line 10 and 11) based on the target state IDs in the initial edge-pair. We then obtain all the outgoing iPhone edges (`iphEdges` in line 12) and Android edges (`andEdges` in line 13) from the already mapped state-pair. To identify closest iPhone and Android edge-pairs (line 15), we loop through the outgoing edges and calculate σ_{Ed}, based on a set of comparison criteria as defined in Formula 6.1:

$$\sigma_{Ed} = \min_{\substack{\forall Ed_{iph} \in iphEdges \\ \forall Ed_{and} \in andEdges}} \left(\frac{f(Ed_{iph}, Ed_{and})}{\sum_{i=1}^{N_{flags}} F_i}\right) * 100 \tag{6.1}$$

where

$$\begin{aligned} f(Ed_{iph}, Ed_{and}) = {} & F_{action} * LD(Iph_{action}, And_{action}) \\ & + F_{label} * LD(Iph_{label}, And_{label}) \\ & + F_{type} * Corresponds(Iph_{type}, And_{type}) \\ & + F_{class} * LD(Iph_{class}, And_{class}) \\ & + F_{title} * LD(Iph_{title}, And_{title}) \\ & + F_{elms} * \sum_{i=1}^{N_{EIPairs}} Similarity(Iph_{elms}, And_{elms}) \\ & + F_{methods} * LD(Iph_{methods}, And_{methods}) \end{aligned}$$

with the `action`, `label`, and `type` of the touched element, `classname`, `title` and attributes of UI elements in the target state, and the `method` calls invoked by the event.

The edge-pair with the lowest computed σ_{Ed} value is selected as the closest Android-iPhone edge-pair and their mapping properties are appended to the model accordingly (line 17). To instantiate different combinations of this metric, we use a set of binary flags, denoted as F_{action}, F_{label}, F_{type}, F_{class}, F_{title}, F_{elms} and $F_{methods}$. The value of each flag is 1 or 0 to activate or ignore a criterion. We propose six different instantiations, listed in Table 6.1, and compare them in our evaluation to assess their effectiveness (discussed in Section 6.5).

Table 6.1: Six combinations for mapping.

ID	Combinations of Comparison Criteria
Comb1	ClassName
Comb2	TouchedElement (action, label, type)
Comb3	TouchedElement+ClassName
Comb4	TouchedElement+ClassName+Title
Comb5	TouchedElement+ClassName+Title+UIElements
Comb6	TouchedElement+ClassName+Title+UIElements+Methods

LD in Formula 6.1 is a relative *Levenshtein Distance* [144] between two strings, calculated as the absolute distance divided by the maximum length of the given strings (See Formula 6.2). Some string patterns that are known to be equivalent are chopped from the strings before calculating their distance. For instance, the words "Activity" in Android `classname` and "ViewController"/"Controller" in iPhone `classname` are omitted.

$$LD(str, str') = \frac{distance(str, str')}{maxLength(str, str')} \tag{6.2}$$

Corresponds in Formula 6.1 is used for comparing the element's type based on the corresponding Android-iPhone UI element equivalent mappings. Since iOS and Android have different UI elements, a mapping is needed to find equivalent widgets. We analyzed GUI elements that exist for both native Android [20] and iPhone [13] platforms and identified the differences and similarities on the two platforms. We used and extended upon existing mappings that are available online [11]. During the interview sessions (See Section 6.2), we cross-validated over 30 control, navigation, and UI element mappings (such as button, label, picker and slider) that function equivalently on the two platforms, so that the generated models

can be used in this phase. We have made these UI equivalent mappings publicly available [27]. *Corresponds* returns 1 if two elements are seen as equivalent and thus can be mapped, and 0 otherwise.

Further, *Similarity* in Formula 6.1 is a relative number ([0,1]) between two sets of elements in the two (target) states calculated as follows:

$$Similarity(elAry, elAry') = \frac{elPairCount(elAry, elAry')}{maxCount(elAry, elAry')} \tag{6.3}$$

where the number of elements that can be mapped is divided by the maximum size of the given arrays. Similar to the touched element, `action`, `label`, and `type` properties of UI elements are used to compute mapping between them.

Finally, going back to our algorithm, mapped edge-pairs are inserted to the main array (line 20), and the next set of states and edges are considered for mapping recursively until no other outgoing edges are left.

Detecting Inconsistencies

Any *unmatched* state left without mapping properties from the previous phase is considered as a *functionality* inconsistency. For a *matched* state-pair, since their incoming edges are mapped, we assume that these target states should be equivalent conceptually. *Data* inconsistencies pertain to text properties of the screen such as titles, labels, buttons, and also the number of cells in a table and tabs. Image related and style related properties are out of scope. We calculate data inconsistencies, σ_{State}, in a pair of mapped states by computing LD between two titles as well as text properties of the elements-pairs.

$$\sigma_{State} = \lceil LD(Iph_{title}, And_{title}) \rceil + \sum_{i=1}^{N_{EIPairs}} \lceil LD(Iph_{txt}, And_{txt}) \rceil \tag{6.4}$$

To compute the correspondence between the elements, we loop through the two arrays of elements. First, we compare the elements' types based on the corresponding Android-iPhone UI element equivalent mappings [14]. For any two elements with the same type and a textual label, we compute LD. We ignore image element types e.g., a button with an image. Where we have multiple elements of the same type, the lowest computed LD is selected as the closest elements-pairs. The σ_{State}

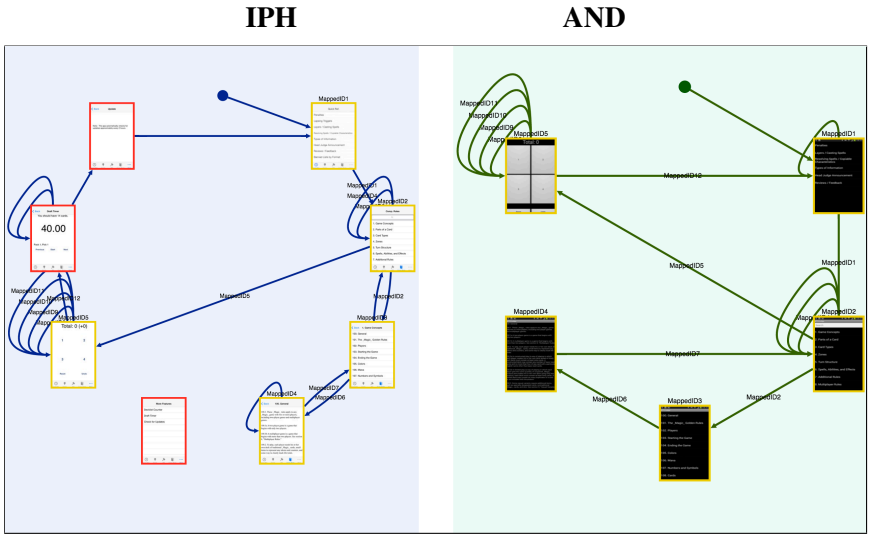


Figure 6.4: Visualization of mapping inferences for MTG iPhone (left) and Android (right) app-pairs. The result indicates 3 *unmatched states* shown with red border (including 2 *functionality* inconsistencies where iPhone has more states than Android and 1 platform specific inconsistency with *MoreViewsController* on iPhone). Other 5 *matched* states have *data* inconsistencies shown with yellow border.

is added as *mapping distance* to the models with the same *mapping tag* for the two states (line 16). Additionally, the detected inconsistencies are added to *mapping result* which are later manifested through our visualization.

Eventually, at the end of this phase, each state is marked as either *unmatched*, *matched with inconsistencies* or *completely matched* in the two models, ready to be visualized in the next phase. Thus, we automatically detect mismatched screens by using one platform's model as an oracle to check another platform's model and vice versa.

6.3.3 Visualizing the Models

After calculating the likely mappings and detecting potential inconsistencies, we visualize the iOS and Android models, side-by-side, colour coding the mapping results. Red, yellow and dark green border colours around states show *unmatched*, *matched with inconsistencies* and *completely matched* states, respectively. Matched states and edges share the same *mapping tag*. Figure 6.4 depicts an example of the output of the visualization phase (it is minimized because of space restrictions). The models can be zoomed in and list detected inconsistencies as well as

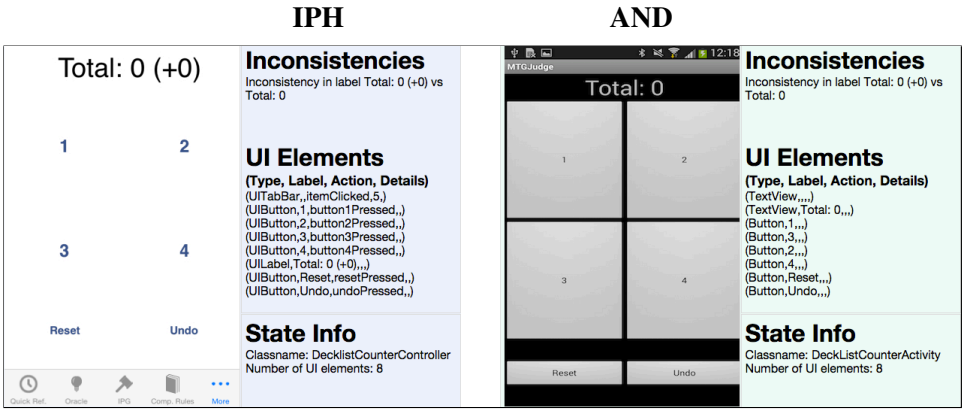


Figure 6.5: Zooming into a selected *State* (or *Edge*) represents detected inconsistencies and UI-structure (or touched element and methods) information of iPhone (left) and Android (right) app-pairs.

UI-structure information on selected state(-pair) or touched element and methods information on selected edge(-pair) (See Figure 6.5).

6.4 Tool Implementation

Our approach is implemented in a tool called CHECKCAMP [27].

Its iPhone analyzer is implemented in Objective-C. We use and extend a number of external libraries. ASPECTS [22] uses Objective-C message forwarding and hooks into messages to enable functionality similar to Aspect Oriented Programming for Objective-C. DCINTROSPECT [30] is a library for debugging iOS user interfaces. We extend DCINTROSPECT to extract a UI element's action message, target object, it's properties and values.

The Android analyzer is implemented in Java (using Android 4.3). To intercept method calls, we rely mainly on ASPECTJ.

Our Mapping and visualization engine is written in Objective-C and implements the states recognition and the states/edges mapping steps of the technique. The output of the mapping engine is an interactive visualization of the iOS and Android models, which highlights the inconsistencies between the app-pairs. The visualization is implemented as a web application and uses the CYTOSCAPE.JS library [29], which is a graph theory library to create models.

6.5 Evaluation

To evaluate the efficacy of our approach we conducted an empirical evaluation, which addresses the following research questions:

RQ1. How accurate are the models inferred by CHECKCAMP?

RQ2. How accurate are the mapping methods? Which set of comparison criteria provides the best results?

RQ3. Is CHECKCAMP capable of detecting valid inconsistencies in cross-platform apps?

6.5.1 Experimental Objects

We include a set of seven large-scale industrial and seven open-source iPhone and Android app-pairs (14 app-pairs in total). The industrial app-pairs are collected from two local mobile companies in Vancouver. The open-source app-pairs are collected from Github. We require the open-source app-pairs to be under the same GitHub repository to ensure that their functionally is meant to be similar across iPhone and Android. Table 6.2 shows the app-pairs included in our evaluation. Each objects's ID, name, resource, and their characteristics in terms of their size and complexity is also presented. XCODE STATISTICIAN [15] and ECLIPSEMETRICS [5] are used to measure lines of code (LOC) in the iOS and Android apps, respectively.

Table 6.2: Characteristics of the experimental objects, together with total number of edges, unique states, elements and manual unique states counts (MC) across all the scenarios.

ID	App [URL] (#Scenarios)	#LOC		#Edges		#Unique States		#Elements		#MC States	
		AND	IPH	AND	IPH	AND	IPH	AND	IPH	AND	IPH
1	MTG-Judge [8] (2)	3,139	1,822	23	38	11	14	118	125	11	14
2	Roadkill-Reporter [18, 33] (1)	1,799	474	3	17	1	5	48	103	4	5
3	NotifyYDP [16] (1)	1,673	1,960	5	18	2	5	101	96	2	5
4	Family [6] (1)	∼12K	∼14K	10	24	3	4	93	372	3	4
5	Chirpradio [17, 32] (1)	1,705	881	3	4	1	1	9	24	1	1
6	Whistle [14] (1)	702	111	3	4	1	1	6	4	1	1
7	Redmine [12] (1)	1,602	48	6	8	5	4	68	26	5	4
8	Industry App A (2)	8,376	4,015	37	46	13	14	1,041	1,286	13	13
9	Industry App B (4)	∼70k	∼28K	49	53	22	22	715	796	22	22
10	Industry App C (6)	∼68K	∼30K	76	87	37	36	1,142	1,028	37	36
11	Industry App D (4)	∼69K	∼28K	66	71	29	31	940	1,803	29	29
12	Industry App E (2)	∼68K	∼26K	23	28	11	12	353	265	11	12
13	Industry App F (3)	∼68K	∼28K	53	57	28	28	635	2,182	28	28
14	Industry App G (4)	∼69K	∼29K	53	56	27	27	813	1,128	27	27

6.5.2 Experimental Procedure

We used iOS 7.1 simulator and a Samsung Galaxy S3, to run the iPhone and Android apps, respectively. To collect traces, two graduate students were recruited. First, they installed a fresh version of each pair of the apps, which were then instrumented by CHECKCAMP. Next, to collect consistent traces, we wrote a set of scenarios for our collected app-pairs and gave each student one scenario for each app to access all use-cases of the Android or iPhone versions of the apps according to the given scenarios. Note that the same user scenario is used for both the iOS and Android versions of an app. The scenarios used in our evaluation are available online [27].

Once traces were collected, CHECKCAMP was executed to obtain the models and mappings. To asses the accuracy of the models generated (RQ1), we compare the number of generated unique states to the actual number of unique states for each app-pair. To form a comparison baseline, we manually examine and navigate the user scenarios for each app-pair and document the number of unique states.

To evaluate the accuracy of the mappings (RQ2), we measure precision, recall, and F-measure for each combination, listed in Table 6.1, and app-pair as follows:

Precision is the rate of mapped states reported by CHECKCAMP that are correct: $\frac{TP}{TP+FP}$

Recall is the rate of correct mapped states that CHECKCAMP finds: $\frac{TP}{TP+FN}$

F-measure is the harmonic mean of precision and recall: $\frac{2\times Precision\times Recall}{Precision+Recall}$

where TP (true positives), FP (false positives), and FN (false negatives), respectively, represent the number of states that are correctly mapped (both fully matched or matched with inconsistencies), falsely mapped, and missed. To document TP, FP, and FN, associated with each app for our combinations of comparison criteria, we manually examine the apps and compare the formed baseline against the reported output.

To validate detected inconsistencies (RQ3), for the best combination calculated in RQ2, we manually examine the reported inconsistencies in each app-pair. The results from our analysis are presented in the next section.

Note that, to the best of our knowledge, there are currently no similar tools to compare the results of CHECKCAMP against. That is why our baselines are created manually.

6.5.3 Results and Findings

RQ1: Inferred models. We ran multiple *Scenarios* to cover all the screens/states in each app. For each scenario, the initial model is constructed over its traces and analyzed by CHECKCAMP. Table 6.2 presents the total number of *Edges*, *Unique States*, and *UI Elements* for all the scenarios running on each Android and iPhone app, produced by CHECKCAMP. The last column of the table also shows the number of *Unique States* counted *manually*. As far as RQ1 is concerned, our results show that CHECKCAMP is able to identify unique states of a given iPhone and Android app-pair and generate their state models correctly for each scenario. However, there is a few cases in our industry iPhone apps (IDs 8 and 11) and Android app (ID 2) where the number of manual unique states does not exactly match the number of unique states collected by the dynamic analyzer. This is mainly because our approach currently takes into account the type of the class (either *Activity* in Android or *View Controller* in iOS) in defining a state and thus separate states are captured for different *View Controllers* (discussed in Section 6.6 under Limitations).

RQ2: Different mapping combinations. The precision and recall rates, measured for the first five combinations, listed in Table 6.1, for our 14 app-pairs, are presented in Figure 6.6. The F-measure is shown in Figure 6.7. We do not include Combination 6 in these figures since apart from the touched element's event-handler (i.e., `action`), comparing the rest of the method calls did not improve the mapping (discussed in Section 6.6 under Conclusive Comparison Criteria). As far as RQ2 is concerned, our results show that CHECKCAMP is highly accurate in mapping state-pairs. As expected, the results are higher in the open-source apps due to the relative simplicity compared to the industry apps. The comparisons in Figure 6.6 and Figure 6.7 reveal that Combination 5 followed by Combination 4 provide the best mapping results in recall, precision, and F-measure for the industry apps. While the results of the combinations have less variation in the open-source

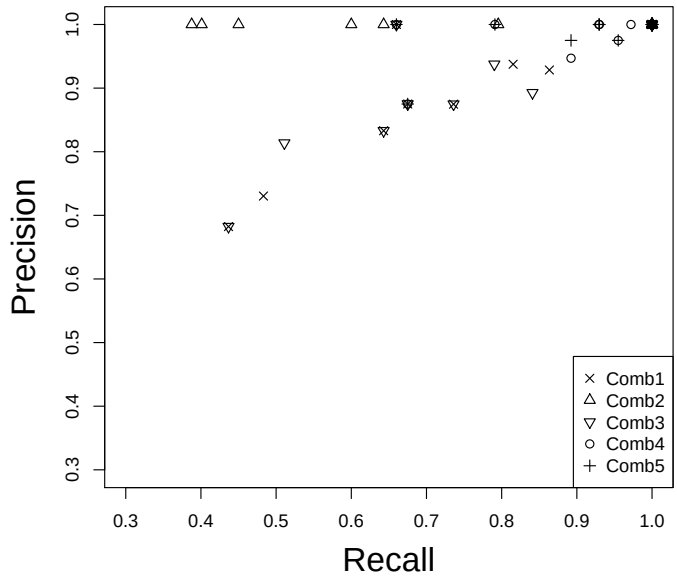


Figure 6.6: Plot of precision and recall for the five mapping combinations of each app-pair.

apps, Combination 2 shows the best results for them. For the best combinations:

- The recall is 1 for the open-source apps, and for the industry apps it oscillates between 0.68–1 (average 0.88) meaning that our approach can successfully map most of the state-pairs present in an app-pair.
- The precision is 1 for the open-source apps, and for the industry apps it oscillates between 0.88–1 (average 0.97), which is caused by a low rate of false positives (discussed in Section 6.6 under Limitations).
- The F-measure is 1 for open-source apps, and varies between 0.75–1 (average 0.92), for industry apps.

RQ3: Valid inconsistencies. As far as RQ3 is concerned, for the best combinations calculated in RQ2, Table 6.3 depicts the number of reported inconsistencies by CHECKCAMP along with some examples. We manually examined and validated (inconsistency categories) in each app-pair across the scenarios. We also

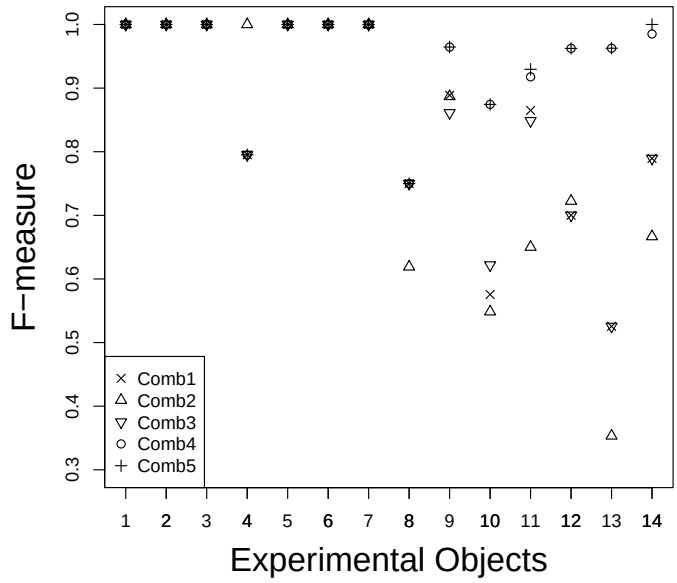


Figure 6.7: F-measure obtained for the five mapping combinations on each app-pair.

computed the average rank and percentage of severity of the valid detected inconsistencies. The used severity ranks are presented in Table 6.4, which are adopted from Bugzilla [25] and slightly adapted to fit inconsistency issues in mobile apps. We computed the percentage of the valid inconsistencies' severity as the ratio of the average severity rank to the maximum severity rank (which is 5).

We found a number of valid *functionality* inconsistencies in the open-source apps, and interestingly, two in the industrial apps (IDs 10 and 12). However, in some app-pairs, functions such as email clients or opening browsers behaved differently on the two platforms. For instance, in the case of app-pair with ID 3, opening browsers and email clients take the Android app user to outside of the application while that is not the case in the iPhone app. As such, the two models have mismatched states in Table 6.2 as CHECKCAMP is not capturing states outside of the app.

Table 6.3: Number of reported inconsistencies by CHECKCAMP, validated, average and percentage of their severity with examples in each app-pair.

ID	#Reported (Categories)	#Validated (Categories)	**Severity (Avg,%)**		Examples of Reported Inconsistencies
1	13 (2 func, 11 data)	13 (2 func, 11 data)	2.6	52%	Android missing 'Draft Time'/'Update' functionality (Figure 6.4) # table cells: iPhone(12628) vs. Android(6336)
2	4 (3 func, 1 data)	1 (1 func)	5	100%	Android missing 'Help' functionality
3	2 (2 data)	2 (2 data)	2	40%	Title: iPhone 'Notify YDP' vs. Android ''
4	3 (1 func, 2 data)	1 (1 func)	5	100%	Android missing 'Change Password' functionality
5	1 (1 data)	1 (1 data)	2	40%	Button: iPhone '' vs. Android 'Play'
6	0	0	–	–	–
7	5 (1 func, 4 data)	5 (1 func, 4 data)	2.6	52%	iPhone missing a functionality
8	14 (14 data)	2 (2 data)	2	40%	Button: iPhone 'Reset' vs. Android 'RESET'
9	2 (2 data)	0	–	–	–
10	5 (1 func, 4 data)	3 (1 func, 2 data)	3	60%	iPhone missing 'Map' functionality
11	2 (2 data)	1 (1 data)	2	40%	Title: iPhone 'May 14' vs. Android 'Schedule'
12	2 (1 func, 1 data)	2 (1 func, 1 data)	3.5	70%	Android missing 'Participants' functionality
13	1 (1 data)	1 (1 data)	2	40%	Title: iPhone 'Details' vs. Android 'Hotels'
14	0	0	–	–	–
All	54 (9 func, 45 data)	32 (7 func, 25 data)	3	60%	–

Table 6.4: Bug severity description.

Severity	Description	Rank
Critical	Functionality loss, no work-around	5
Major	Functionality loss, with possible work-around	4
Normal	Makes a function difficult to use	3
Minor	Not affecting functionality, behaviour is not natural	2
Trivial	Not affecting functionality, cosmetic issue	1

Among the data inconsistencies in Table 6.3, are inconsistencies in the number of cells and text of titles, labels, and buttons. Most of the false positives in the reported data inconsistencies (in particular in app-pair with ID 8) are due to the UI structure of a state being implemented differently on the two platforms (discussed in Section 6.6 under Limitations). Thus, CHECKCAMP could not map the elements correctly and reported incorrect inconsistencies.

6.6 Discussion

In this section, we discuss our general findings, limitations of CHECKCAMP, and some of the threats to validity of our results.

6.6.1 Comparison Criteria

Among the code-based and GUI-based comparison criteria, our evaluation shows that the most effective in the mapping phase pertains to information about the text, action, and type of UI elements that events are fired on, as well as the classname and title of the states. In addition, while we extract a set of method calls after an event fires, our investigation shows that only the action of the touched UI element is effective. We found that even after omitting OS built-in methods, such as delegate methods provided by the native SDK, or library API calls, the method names are quite different in the two platforms and thus provided no extra value in the mapping phase.

6.6.2 Limitations

There are some limitations to our current implementation. First, deciding what constitutes a UI state is not always straightforward. For instance, consider two

screens with a list of different items. In the Android version of an app the same *Activity* is used to implement the two screens while on the iPhone version separate *View Controllers* exist and currently as shown in Algorithm 3, the type of the class (either *Activity* in Android or *View Controllers* in iOS) is checked (line 8) for identifying a state and thus (mistakenly) separate states are captured in iPhone.

Next, the low rate of false positives in RQ2 include examples where even considering our selected properties all together, CHECKCAMP still lacks enough information to conclude correct mappings. For instance, if an `ImageButton` which contains an image as a background is exercised, there would be no text/label to be compared. Another limitation is with respect to the string edit distance used in our algorithm; for instance, the two classnames *DetailedTipsViewController* and *TipsDetailActivity* are falsely reported as being different based on their distance. This means, if the outgoing edges can not be mapped correctly in Algorithm 4, CHECKCAMP halts and cannot go any further. Backtracking based approaches can be considered to recover if it performs incorrect matches.

Another limitation is related to the high false-positive rate in the reported data inconsistencies in RQ3. In states with multiple elements of the same type, e.g., buttons with images or text properties, our programmatic approach in CHECKCAMP cannot map them correctly. Another reason, occurred in some cases, is the UI structure of a state-pair is implemented differently. For instance, in an Android state, buttons exist with text properties whereas in the corresponding iPhone state, those texts are implemented through labels along with buttons. However, this limitation could be addressed through image-processing techniques [69, 205] on the iPhone and Android screenshots collected by the dynamic analyzers. This could enable the detection of other types of inconsistencies between app-pairs including image-related data, layout, or style.

6.6.3 Applications

There are various applications for our technique. First of all, our technique supports mobile developers in comprehending, analyzing, and testing their native mobile apps that have implementations in both iOS and Android. Many developers interact with GUI to comprehend the software by creating a mental model of the

application [192]. On average, 48% of a desktop applications's code is devoted to GUI [170]. We believe the amount of GUI-related code is higher in mobile apps due to their highly interactive nature. Thus, using the models to provide a visualization of the apps accompanied with the UI-structure and method calls in the visualization output, would support mobile developers and testers in their program comprehension and analysis tasks and to obtain a better understanding of their mobile apps. The models inferred by CHECKCAMP can also be used for generating test cases. In terms of scalability, the results in Table 6.2 show that our approach is scalable to large industrial mobile apps consisting of tens of thousands of LOC and many states.

6.6.4 Threats to Validity

The fact that we form the comparison baselines manually could be a threat to internal validity. We did look for other similar tools to compare our results against, without success. Manually going through the different applications to create baselines is labour intensive and potentially subject to errors and author's bias. We tried to mitigate this threat by asking the first two authors to create the comparison baselines together before conducting the experiment. Additionally, we had a small number of scenarios in particular for the open source apps. We tried to mitigate this threat by assuring that these scenarios covered the app screens/states fully. A threat to the external validity of our experiment is with regard to the generalization of the results to other mobile apps. To mitigate this threat, we selected our experimental objects from industrial and open-source domains with variations in functionality, structure and size. With respect to reproducibility of our results, CHECKCAMP, the open-source experimental objects, their scenarios and results are publicly available [27].

6.7 Related Work

Dealing with multiple platforms is not specific to the mobile domain. The problem also exists for cross-browser compatibility testing. However, in the mobile domain, each mobile platform is different with regard to the OS, programming languages, API/SDKs, and supported tools, making it much more challenging to detect incon-

sistencies automatically.

Mesbah and Prasad [161] propose a functional consistency check of web application behaviour across different browsers. Their approach automatically analyzes the given web application, captures the behaviour as a finite-state machine and formally compares the generated models for equivalence to expose discrepancies. Their model generation [163] and mapping technique is based on DOM states of a web application while CHECKCAMP deals with native iOS and Android states and mappable code-based and GUI related metrics of the two mobile platforms. Choudhary *et al.* [74] propose a technique to analyze the client-server communication and network traces of different versions of a web application to match features across platforms.

In the mobile domain, Rosetta [100] infers likely mappings between the JavaME and Android graphics APIs. They execute application pairs with similar inputs to exercise similar functionality and logged traces of API calls invoked by the applications to generate a database of functionally equivalent trace pairs. Its output is a ranked list of target API methods that likely map to each source API method. Cloud Twin [117] natively executes the functionality of a mobile app written for another platform. It emulates the behaviour of Android apps on a Windows Phone where it transmits the UI actions performed on the Windows Phone to the cloud server, which then mimics the received actions on the Android emulator. To our best knowledge, none of the related work addresses inconsistency detection across iOS and Android mobile platforms.

6.8 Conclusions

This work is motivated by the fact that implementation of mobile apps for multiple platforms – iOS and Android – has become an increasingly common industry practice. As a result, a challenge for mobile developers and testers is to keep the app consistent, and ensure that the behaviour is the same across multiple platforms. In this work, we proposed CHECKCAMP, a technique to automatically detect and visualize inconsistencies between iOS and Android versions of the same mobile app. Our empirical evaluation on 14 app-pairs shows that the GUI model-based approach can provide an effective solution; CHECKCAMP can correctly infer mod-

els, and map them with a high precision and recall rate. Further, CHECKCAMP was able to detect 32 valid functional and data inconsistencies between app versions.

While we are encouraged by the evaluation results of CHECKCAMP, there are several opportunities in which our approach can be enhanced and extended for future research. The immediate step would be to conduct an in-depth case study, carried out in an industrial setting with a number of developers using CHECKCAMP. This would help validate the efficiency of the mapping and the visualizations. Additionally, the execution of consistent scenarios can be enhanced by the use of mobile apps that have test suites such as CALABASH [26] scripts. The traces generated by test suites can be leveraged in the mapping engine to enhance the approach.

Systematically crawling to recover models is also an alternative to using scenarios. While there are limitations of automated model recovery, it could complement human-provided scenarios, to ensure better coverage. We have taken the first required steps for automatically generating state models of iPhone apps [85] through a reverse engineering technique. There have been similar techniques for Android apps [55, 73, 102, 217].

Another direction is to improve the current dynamic analyzers to capture information regarding each device's network communication (client-server communication of platform-specific versions of a mobile app), as well as the API calls made to utilize the device's native functionality such as GPS, SMS, Calendar, Camera, and Gallery.

Chapter 7

Conclusions and Future Work

Mobile app development on platforms such as Android and iOS has gained tremendous popularity recently. This dissertation aims at advancing the state-of-the-art by 1) obtaining insights regarding current practices, real challenges and concerns in mobile app development as well as 2) proposing a new set of techniques and tools based on the identified challenges. To this end, we designed five research questions. The first three research questions addressed the first part of our goal, in particular, each responded to a gap in the current state-of-the-art. The last two research questions are follow-up studies, which address the identified challenges by proposing techniques and tools. We believe that our primary contributions and publications, presented in this dissertation, have addressed our goal and research questions.

7.1 Revisiting Research Questions

RQ1. *What are the main challenges developers face in practice when they build mobile apps?*

Chapter 2. We presented the first qualitative field study [86] targeting mobile app development practices and challenges, which is considered "*a very strong and influential contribution to the whole SE community*" by our reviewers. We started by conducting and analyzing interviews with 12 senior mobile app developers, from nine different industrial companies. We followed a Grounded Theory approach to analyze our interviews. Based on the outcome of these interviews, we designed and distributed an online survey, targeted the popular Mobile Development Meetup and LinkedIn groups related to native mobile development. We kept the survey live for

two and a half months, which was fully completed by 188 mobile app developers worldwide.

However, similar to quantitative research, qualitative studies could suffer from threats to validity, which is challenging to assess as outlined by Onwuegbuzie *et al.* [179]. For instance, in codification, the researcher bias can be troublesome, skewing results on data analysis [132]. We tried to mitigate this threat through triangulation; The codification process was conducted by two researchers, one of whom had not participated in the interviews, to ensure minimal interference of personal opinions or individual preferences. Additionally, we conducted a survey to challenge the results emerging from the interviews. Both the interview and survey questionnaire were designed by a group of three researchers, with feedback from four external people – one senior Ph.D. student and three industrial mobile app developers – in order to ensure that all the questions were appropriate and easily comprehensible. Another concern was a degree of generalizability. We tried to draw representative mobile developer samples from nine different companies. Thus, the distribution of participants includes different companies, development team sizes, platforms, application domains, and programming languages – representing a wide range of potential participants. Of course, the participants in the survey also have a wide range of background and expertise. All this gives us some confidence that the results have a degree of generalizability. One risk within Grounded Theory is that the resulting findings might not fit with the data or the participants [99]. To mitigate this risk, we challenged the findings from the interviews with an online survey, filled out by 188 practitioners worldwide. The results of the survey confirmed that the main concepts and codes, generated by the Grounded Theory approach, are in line with what the majority of the mobile development community believes. Lastly, in order to make sure that the right participants would take part in the survey, we shared the survey link with some of the popular Mobile Development Meetup and LinkedIn groups related to native mobile app development. Furthermore, we did not offer any financial incentives nor any special bonuses or prizes to increase response rate.

RQ2. *What are the characteristics of non-reproducible bug reports and the challenges developers deal with?*

Chapter 3. To obtain better insights of issues and concerns in software develop-

ment, in general, it is a very common practice that researchers investigate other sources of software development data. In particular, bug repository systems have become an integral component of software development activities. Ideally, each bug report should help developers find and fix a software fault. However, there is a subset of reported bugs that is not (easily) reproducible, on which developers spend considerable amounts of time and effort. While we started with mobile non-reproducible (NR) bugs, we noticed that none of the related work investigates non-reproducible bug reports in isolation. Thus, we expanded the study to other software environment and domains (desktop, web, and mobile). In this work [87], we presented the first empirical study on the frequency, nature, and root cause categories of non-reproducible bug reports. We mined six bug tracking repositories from three different domains and found that 17% of all bug reports are resolved as non-reproducible at least once in their life-cycles. Non-reproducible bug reports, on average, remain active around three months longer than other resolution types while they are treated similarly in terms of the extent to which they are discussed or the number of developers involved. Furthermore, we manually examined and classified six common root cause categories. Our classification indicated that "Interbug Dependencies" forms the most common category (45%), followed by "Environmental Differences" (24%), "Insufficient Information" (14%), "Conflicting Expectations" (12%), and "Non-deterministic Behaviour" (3%).

However, our manual classification of the bug reports could be a source of internal threats to validity. In order to mitigate errors and possibilities of bias, we performed our manual classification in two phases where (1) the inference of rules was initially done by the first author; the rules were cross-validated and uncertainties were resolved through extensive discussions and refinements between two researchers; the generated categories were discussed and refined by a group of three researchers, (2) the actual distribution of bug reports into the six inferred categories was subsequently conducted by myself following the classification rules inferred in the first step. In addition, since this is the first study classifying NR bug reports, we had to infer new classification rules and categories. Thus, one might argue that our NR rules and categories are subjective with blurry edges and boundaries. By following a systematic approach and triangulation we tried to mitigate this threat. Another threat in our study is the selection and use of these bug

repositories as the main source of data. However, we tried to mitigate this threat by selecting various large repositories and randomly selecting NR bug reports for analysis. In terms of external threats, we tried our best to choose bug repositories from a representative sample of popular and actively developed applications in three different domains (desktop, web, and mobile). With respect to bug tracking systems, JIRA and BUGZILLA are well-known popular systems, although bug reports in projects using other bug tracking systems could behave differently. Thus, regarding a degree of generalizability, replication of such studies within different domains and environments (in particular for industrial cases) would help to generalize the results and create a larger body of knowledge.

RQ3. *What are the app-store characteristics of the same mobile app, published in different marketplaces? How are the major concerns or complaints different on each platform?*

Chapter 4. Online app stores are the primary medium for the distribution of mobile apps. Through app stores, users can download and install apps on their mobile devices and subsequently rate the apps. As such, app stores provide an important channel for app developers to collect user feedback such as the overall rating of their app, issues or bugs detected and new feature requests. To attract as many users as possible, developers often implement the same app for multiple mobile platforms [86]. We presented the first large-scale study on mobile *app-pairs*, i.e., the same app implemented for iOS and Android platforms, in order to analyze and compare their various attributes, user reviews, and root causes of user complaints at multiple levels of granularity. We mined the two most popular app stores and employ a mixed-methods approach using both quantitative and qualitative analysis. Our results show that on average the stars and prices are similar on both platforms, with some fluctuations in price. Reasons for price fluctuations include different monetizing strategies, offering different features and different efforts and costs required to maintain the app. The number of ratings is greatly in favour of Android where in 63% of the app-pairs it has 4,821 more ratings than the iOS platform. Further, some top rated apps only exist on one platform, reasons for this include lack of resources, platform restrictions and revenue per platform. We combined the stars, ratings, and user reviews to measure apps' success on the Android and iOS platforms for 2K app-pairs and found that 17.4% have a difference of 25% or

more in their success rate between the two platforms. Finally, we looked closely at user complaints and concerns. We found that, on average, iOS apps have more critical and post-update problems while Android apps have more complaints related to compatibility, usability, performance, security or functionality. It connects the developers comments to our findings as they mentioned, "*Apple forces the developers to constantly migrate the apps to their latest OS and tool versions.*" On the other hand, there is more device fragmentation on Android and the compatibility, usability, performance, and functionality are more related to dealing with a variety of devices.

However, our manual labelling of the reviews to train the classifiers could be a source of internal threat to validity. In order to mitigate this threat, uncertainties were cross-validated and resolved through discussions and refinements between the authors. With respect to the detection of app-pairs, our technique cannot retrieve all the possible app-pairs since it only considers two apps a pair if their app name and developer name start with the same root word. For instance, an app named *The Wonder Weeks*[43] on iOS has a pair on the Android platform with the name *Baby Wonder Weeks Milestones*.[44] Such a pair would not be retrieved by our technique since it does not start with the same root word. Thus, as shown in Figure 4.4, the app-pairs detected in our study are a subset of all possible app-pairs. In terms of external threats, we tried our best to choose app-pairs from a representative sample of popular mobile apps and categories. With respect to app store systems, iTunes and Google Play are the most popular systems currently, although apps in other app stores could have other characteristics. Regarding generalizability, replication of such studies within different app stores would help to generalize the results and create a larger body of knowledge. Additionally, the classifiers alone could be useful to group the reviews for developers.

RQ4. *How can we help developers to better understand their mobile apps?*

Chapter 5. Many developers interact with the graphical user interface (GUI) to comprehend the software by creating a mental model of the application [192]. For traditional desktop applications, an average of 48% of the application's code is devoted to GUI [170]. Because of their highly interactive nature, we believe the

[43] https://goo.gl/ofLWim
[44] https://goo.gl/tMKWTx

amount of GUI-related code is typically higher in mobile apps. To support mobile developers in their program comprehension and analysis tasks, we presented the first reverse engineering technique to automatically navigate a given iPhone app and infer a model of its user interface states [85]. We implemented our approach in ICRAWLER, which is capable of exercising and analyzing UI changes and generate a state model of the application. The results of our evaluation, on six open source iPhone apps, point to the efficacy of the approach in automatically detecting unique UI states, with a minimum level of manual effort required from the user. We believe our approach and techniques have the potential to help mobile app developers increase the quality of iOS apps.

However, there are some limitations within our current implementation of the approach. Although it is minimal, the users still need to complete a few tasks to set up ICRAWLER within their applications manually. There are also some UI elements such as the tool bar, slider, page control, and search bar, which are not supported currently. In addition, while ICRAWLER at the moment supports the most common gestures in iOS apps such as tapping on a UI element, inserting text, and scrolling views, still there is no support for other advanced gestures such as swiping pages and pinching (e.g., zooming in and out images). Furthermore, the fact that we form the comparison baselines manually could be a threat to internal validity. We did look for other tools to compare our results against, without success. Manually going through the different applications to create baselines is labour intensive and potentially subject to errors and author's bias. We tried to mitigate this threat by asking two other students to create the comparison baselines. Further, in our attempt to gather the experimental objects, we noticed that there is a small collection of open-source iPhone apps available online – note that we could not use applications available in AppStore for our experiment since we needed access to their source code. Even though, this made it difficult to select applications that reflect the whole spectrum of different UI elements in iPhone apps, we believe the selected objects are representative of the type of applications ICRAWLER can reverse engineer. However, we acknowledge the fact that, in order to draw more general conclusions, more mobile apps are required.

RQ5. *How can we help developers to automatically detect inconsistencies in their same mobile app across multiple platforms?*

Chapter 6. This work [88] is motivated by the fact that implementation of mobile apps for multiple platforms – iOS and Android – has become an increasingly common industry practice. Therefore, as we identified in our initial field study [86], a major challenge for mobile developers and testers is to keep the app consistent and ensure that the behaviour is the same across multiple platforms. We also found that developers currently treat the mobile app for each platform *separately* and *manually* perform screen-by-screen comparisons, often detecting many cross-platform inconsistencies [86]. This manual process is, however, tedious, time-consuming, and error-prone. Thus, we proposed [88] the first automated technique, called CHECKCAMP (Checking Compatibility Across Mobile Platforms), which automatically detect and visualize inconsistencies between iOS and Android versions of the same mobile app.

However, there are some limitations to our current implementation. First, deciding what constitutes a UI state is not always straightforward. For instance, consider two screens with a list of different items. In the Android version of an app the same *Activity* is used to implement the two screens while on the iPhone version separate *View Controllers* exist and currently as shown in Algorithm 3, the type of the class (either *Activity* in Android or *View Controllers* in iOS) is checked (line 8) for identifying a state and thus (mistakenly) separate states are captured in iPhone. Next, the low rate of false positives in RQ2 include examples where even considering our selected properties all together, CHECKCAMP still lacks enough information to conclude correct mappings. For instance, if an `ImageButton` which contains an image as a background is exercised, there would be no text/label to be compared. Another limitation is with respect to the string edit distance used in Algorithm 4 for mapping two classnames based on their distance. This means if the outgoing edges can not be mapped correctly, CHECKCAMP halts and cannot go any further. Backtracking-based approaches can be considered to recover if it performs incorrect matches. Another limitation is related to the high false-positive rate in the reported data inconsistencies in RQ3. In states with multiple elements of the same type, e.g., buttons with images or text properties, our programmatic approach in CHECKCAMP cannot map them correctly. Another reason, occurred in some cases, is the UI structure of a state-pair is implemented differently. For instance, in an Android state, buttons exist with text properties whereas in the cor-

responding iPhone state, those texts are implemented through labels along with buttons. However, this limitation could be addressed through image-processing techniques [69, 205] on the iPhone and Android screenshots collected by the dynamic analyzers. This could enable the detection of other types of inconsistencies between app-pairs including image-related data, layout, or style. Additionally, the fact that we form the comparison baselines manually could be a threat to internal validity. We tried to mitigate this threat by asking two researchers to create the comparison baselines together before conducting the experiment. Additionally, we had a small number of scenarios in particular for the open source apps. We tried to mitigate this threat by assuring that these scenarios covered the app screens/states fully. A threat to the external validity of our experiment is with regard to the generalization of the results to other mobile apps. To mitigate this threat, we selected our experimental objects from industrial and open-source domains with variations in functionality, structure and size.

7.2 Future Work and Concluding Remarks

To summarize, we have identified the practices and challenges in software development for mobile devices, that go beyond the anecdotic evidence. We started with a qualitative field study. We provided a list of important software engineering research issues related to mobile development. We also conducted two empirical studies on software development data for mobile apps. After our qualitative studies, in order to address the identified challenges, we conducted follow-up research and proposed automated model-based techniques for generating a model of a mobile app through ICRAWLER approach as well as detecting inconsistencies of a mobile app across multiple platforms through CHECKCAMP approach. Overall, our findings showed the effectiveness of the proposed model generation and mapping techniques in terms of accuracy and inconsistency detection capability.

Future work on non-reproducible bug reports can focus on (1) bug reports in the "Interbug Dependencies" category to design techniques that would facilitate identifying, linking, and clustering them upfront so that developers would not have to waste time on them, (2) incorporating better collaboration tools into bug tracking systems to facilitate better communication between different stakeholders to

address the problem with the other NR categories. Future work regarding the app-pairs study can focus on the release dates of the app-pairs to understand which platform developers will target first when they release a new app. Further, app descriptions can be used to compare the app features provided on different platforms.

Additionally, there are several opportunities in which ICRAWLER approach can be enhanced and extended for future research. The immediate step would be to extend the current version of ICRAWLER to support the remaining set of UI elements within `UIKIT` such as date picker, action sheet, alert view, the tool bar, slider, page control, and search bar. Other directions we will pursue are using ICRAWLER technique for smoke testing of iPhone apps as well as generating test cases from the inferred state model. Furthermore, ICRAWLER can be expanded to support iPad apps. Additionally, ICRAWLER can be extended with reverse engineering analysis at the binary level. It would be beneficial as Apple app store distributes binary code of the applications, and this would be interesting to apply automated testing to any application disregarding having accessibility to its source code. It also would reduce/omit the users manual effort to set up the analysis environment. Similarly, there are various opportunities in which CHECKCAMP approach can be improved and extended for future research. The immediate step would be to conduct an in-depth case study, carried out in an industrial setting with a number of developers using CHECKCAMP. Additionally, the execution of consistent scenarios can be enhanced by the use of mobile apps that have test suites such as CALABASH [26] scripts. The traces generated by test suites can be leveraged in the mapping engine to enhance the approach. Systematically crawling to recover models is also an alternative to using scenarios. While there are limitations of automated model recovery, it could complement human-provided scenarios, to ensure better coverage. We have taken the first required steps for automatically generating state models of iPhone apps through ICRAWLER reverse engineering technique [85]. There have been similar techniques for Android apps [55, 73, 102, 217]. Another direction is to improve the current dynamic analyzers to capture information regarding each device's network communication (client-server communication of platform-specific versions of a mobile app), as well as the API calls made to utilize the device's native functionality such as GPS, SMS, Address Book, E-mail, Calendar, Camera, and Gallery. Finally, CHECKCAMP approach can be extended to support third-

party testing, where no source code is available, as well as other mobile platforms such as Windows Phone and Blackberry.

Nonetheless, our research is only scratching the surface of mobile development domain with its fast paced and highly frequent changes. We open sourced all of our empirical data and tools, making our techniques and findings applicable in future research.

Bibliography

[1] The developer's guide - Android developers. https://developer.android.com/guide/index.html. Accessed: 2015-12-15. → pages 144

[2] Eclipse Bugzilla. https://bugs.eclipse.org/bugs/, . Accessed: 2015-12-15. → pages 63

[3] MediaWiki Bugzilla. https://bugzilla.wikimedia.org/, . Accessed: 2015-12-15. → pages 63

[4] Bugzilla@Mozilla. https://bugzilla.mozilla.org, . Accessed: 2015-12-15. → pages 63

[5] Eclipse Metrics plugin. http://metrics2.sourceforge.net/. Accessed: 2015-12-15. → pages 153

[6] Family App for iPhone and Android. https://github.com/FamilyLab/Family. Accessed: 2015-12-15. → pages 154

[7] JIRA. https://confluence.atlassian.com/display/JIRA050/JIRA+Documentation. Accessed: 2015-12-15. → pages 58, 60

[8] MTGJudge App for iPhone and Android. https://github.com/numegil/MTG-Judge. Accessed: 2015-12-15. → pages 154

[9] Non-reproducible bug report analyser and empirical data. https://github.com/saltlab/NR-bug-analyzer. Accessed: 2015-12-15. → pages 60, 64, 65

[10] Moodle Tracker! https://tracker.moodle.org/issues/?jql=. Accessed: 2015-12-15. → pages 63

[11] PortKit: UX Metaphor Equivalents for iOS & Android. http://kintek.com.au/blog/portkit-ux-metaphor-equivalents-for-ios-and-android/. Accessed: 2015-12-15. → pages 149

[12] Redmine App for iPhone and Android. https://github.com/webguild/RedmineMobile. Accessed: 2015-12-15. → pages 154

[13] UIKit Framework Reference. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/. Accessed: 2015-12-15. → pages 149

[14] Whistle App for iPhone and Android. https://github.com/yasulab/whistle. Accessed: 2015-12-15. → pages 154

[15] Xcode Statistician. http://www.alexcurylo.com/blog/2010/11/01/xcode-statistician/. Accessed: 2015-12-15. → pages 153

[16] YDP App for iPhone and Android. https://github.com/alakinfotech/YDP. Accessed: 2015-12-15. → pages 154

[17] The Android version of Chirpradio. https://github.com/chirpradio/chirpradio-android. Accessed: 2015-12-15. → pages 154

[18] The Android version of Roadkill Reporter. https://github.com/calebgomer/Roadkill_Reporter_Android. Accessed: 2015-12-15. → pages 154

[19] Android Market Stats. http://www.appbrain.com/stats/, . Accessed: 2015-12-15. → pages 85, 115, 138

[20] android.widget Package. https://developer.android.com/reference/android/widget/package-summary.html, . Accessed: 2015-12-15. → pages 149

[21] App Store Metrics. http://148apps.biz/app-store-metrics/. Accessed: 2015-12-15. → pages 85, 114, 138

[22] Aspects. https://github.com/steipete/Aspects. Accessed: 2015-12-15. → pages 152

[23] Bugzilla: Eclipse Bug #106396. https://bugs.eclipse.org/bugs/show_bug.cgi?id=106396, . Accessed: 2015-12-15. → pages 60

[24] Bugzilla. http://www.bugzilla.org/docs/, . Accessed: 2015-12-15. → pages 58, 60, 77

[25] Bugzilla Severity Definitions. https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Severity, . Accessed: 2015-12-15. → pages 158

[26] Calabash. http://calaba.sh/. Accessed: 2015-12-15. → pages 164, 173

[27] iOS and Android Dynamic Analyzers, Mapping and Visualization Engine together with Open-source Experimental Scenarios and Results. https://github.com/saltlab/camp. Accessed: 2015-12-15. → pages 139, 150, 152, 155, 162

[28] What is an issue? https://confluence.atlassian.com/display/JIRA/What+is+an+Issue. Accessed: 2015-12-15. → pages 60

[29] Graph theory (a.k.a. network) library for analysis and visualisation. http://js.cytoscape.org/. Accessed: 2015-12-15. → pages 152

[30] DCIntrospect. https://github.com/domesticcatsoftware/DCIntrospect. Accessed: 2015-12-15. → pages 129, 152

[31] Frank: Automated Acceptance Tests for iPhone and iPad. http://www.testingwithfrank.com/. Accessed: 2015-12-15. → pages 117

[32] The iOS version of Chirpradio. https://github.com/chirpradio/chirpradio-ios. Accessed: 2015-12-15. → pages 154

[33] The iOS version of Roadkill Reporter. https://github.com/calebgomer/Roadkill_Reporter_iOS. Accessed: 2015-12-15. → pages 154

[34] KIF iOS Integration Testing Framework. https://github.com/square/KIF. Accessed: 2015-12-15. → pages 117, 129

[35] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html, . Accessed: 2015-12-15. → pages 118

[36] MonkeyTalk for iOS & Android. http://www.gorillalogic.com/testing-tools/monkeytalk, . Accessed: 2015-12-15. → pages 117

[37] NSKeyValueCoding Protocol Reference. https://developer.apple.com/library/ios/navigation/, . Accessed: 2015-12-15. → pages 124

[38] Objective-C Runtime Reference. https://developer.apple.com/library/ios/navigation/, . Accessed: 2015-12-15. → pages 123

[39] Project SIKULI. http://sikuli.org. Accessed: 2015-12-15. → pages 117

[40] Works on my machine - How to fix non-reproducible bugs? http://stackoverflow.com/questions/1102716/works-on-my-machine-how-to-fix-non-reproducible-bugs. Accessed: 2015-12-15. → pages 58, 60

[41] Bug fields. https://bugzilla.mozilla.org/page.cgi?id=fields.html. Accessed: 2015-12-15. → pages 60

[42] XSWI: XML stream writer for iOS. http://skjolber.github.io/xswi/. Accessed: 2015-12-15. → pages 129

[43] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93. ACM, 2015. → pages 47, 49

[44] S. Adolph, W. Hall, and P. Kruchten. Using grounded theory to study the experience of software development. *Empirical Softw. Engg.*, 16(4): 487–513, 2011. → pages 3, 11, 12, 54

[45] S. Adolph, P. Kruchten, and W. Hall. Reconciling perspectives: A grounded theory of how people manage the process of software development. *J. Syst. Softw.*, 85(6):1269–1286, 2012. → pages 12, 54

[46] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 22:1–22:6. ACM, 2010. → pages 11, 50, 51

[47] Y. Agarwal and M. Hall. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 97–110. ACM, 2013. → pages 47

[48] M. Ahmad, N. Musa, R. Nadarajah, R. Hassan, and N. Othman. Comparison between android and iOS operating system in terms of security. In *Information Technology in Asia (CITA), 2013 8th International Conference on*, pages 1–4. IEEE, 2013. → pages 35

[49] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A GUI crawling-based technique for Android mobile application testing. In *Proceedings of the Workshops at IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 252–261. IEEE Computer Society, 2011. → pages 4, 47, 117

[50] Appcelerator. Appcelerator / IDC Q3 2014 Mobile Trends Report. http://www.appcelerator.com/enterprise/resource-center/research/appcelerator-2014-q3-mobile-report/, . Accessed: 2015-12-15. → pages 1, 138

[51] Appcelerator. Appcelerator / IDC Q4 2013 Mobile Trends Report. http://www.appcelerator.com.s3.amazonaws.com/pdf/q4-2013-devsurvey.pdf, . Accessed: 2015-12-15. → pages 1, 138

[52] Apple Store Crawler. https://github.com/MarcelloLins/Apple-Store-Crawler. Accessed: 2015-12-15. → pages 87

[53] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 298–308. IEEE Computer Society, 2009. → pages 58, 81

[54] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015. ACM, 2015. → pages 2, 47, 50, 52, 85, 112

[55] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. *SIGPLAN Not.*, 48(10):641–660, Oct. 2013. → pages 164, 173

[56] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 112–121. ACM, 2014. → pages 52

[57] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 588–598. ACM, 2014. → pages 47

[58] G. Bavota, M. Linares-Vasquez, C. Bernal-Cardenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of Android apps. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2015. → pages 47, 53

[59] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371. IEEE Press, 2013. → pages 79, 82

[60] C. Bernal-Cárdenas. Improving energy consumption in android apps. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 1048–1050. ACM, 2015. → pages 47

[61] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318. ACM, 2008. → pages 58, 80

[62] S. Beyer and M. Pinzger. A manual categorization of Android app development issues on Stack Overflow. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 531–535. IEEE, Sept 2014. → pages 2, 50, 51

[63] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. Koduru. An empirical analysis of bug reports and bug fixing in open source Android apps. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 133–143. IEEE, March 2013. → pages 2, 35, 47, 50

[64] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source Android apps. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 133–143. IEEE Computer Society, 2013. → pages 2, 80

[65] Categories and Extensions. https://developer.apple.com/library/ios/navigation/. Accessed: 2015-12-15. → pages 125, 142

[66] R. Chandra, B. F. Karlsson, N. D. Lane, C.-J. M. Liang, S. Nath, J. Padhye, L. Ravindranath, and F. Zhao. How to smash the next billion mobile app bugs? *GetMobile: Mobile Computing and Communications*, 19(1), 2015. → pages 47, 49

[67] M. Chandramohan and H. B. K. Tan. Detection of mobile malware in the wild. *Computer*, 99, 2012. → pages 35, 47, 116

[68] R. Chandy and H. Gu. Identifying spam in the iOS app store. In *Proceedings of the Joint WICOW/AIRWeb Workshop on Web Quality*, WebQuality '12, pages 56–59. ACM, 2012. → pages 50, 112

[69] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 1535–1544. ACM, 2010. → pages 118, 126, 161, 172

[70] L. Chen, M. AliBabar, and B. Nuseibeh. Characterizing architecturally significant requirements. *IEEE Softw.*, 30(2):38–45, 2013. → pages 12, 54

[71] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang. Ar-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 767–778. ACM, 2014. → pages 2, 4, 47, 50, 85, 92, 103, 112

[72] Y. Chen, H. Xu, Y. Zhou, and S. Zhu. Is this app safe for children?: A comparison study of maturity ratings on android and iOS applications. In *Proceedings of the International Conference on World Wide Web*, WWW '13, pages 201–212. ACM, 2013. → pages 112, 113

[73] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10): 623–640, 2013. → pages 47, 164, 173

[74] S. R. Choudhary, M. Prasad, and A. Orso. Cross-platform feature matching for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014. ACM, 2014. → pages 4, 48, 163

[75] CNET. Researcher posts Facebook bug report to Mark Zuckerberg's wall. http://news.cnet.com/8301-1023_3-57599043-93/researcher-posts-facebook-bug-report-to-mark-zuckerbergs-wall/. Accessed: 2015-12-15. → pages 58

[76] Cocoa developer community. Method Swizzling. https://developer.apple.com/library/ios/navigation/. Accessed: 2015-12-15. → pages 125, 142

[77] G. Coleman and R. O'Connor. Using grounded theory to understand software process improvement: A study of Irish software product companies. *Inf. Softw. Technol.*, 49(6):654–667, 2007. → pages 13, 54, 55

[78] G. Coleman and R. O'Connor. Investigating software process in practice: A grounded theory perspective. *J. Syst. Softw.*, 81(5):772–784, 2008. → pages 12, 54, 55

[79] J. W. Creswell. *Qualitative inquiry and research design : choosing among five approaches (2nd edition)*. Thousand Oaks, CA: SAGE, 2007. → pages 12

[80] J. W. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Publications, Incorporated, 2013. → pages 60, 86

[81] I. Dalmasso, S. Datta, C. Bonnet, and N. Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 323–328. IEEE, July 2013. → pages 53

[82] J. Dehlinger and J. Dixon. Mobile application software engineering: Challenges and research directions. In *Proceedings of the Workshop on Mobile Software Engineering*, pages 29–32. Springer, 2011. → pages 2, 11, 50, 51, 115

[83] S. Diewald, L. Roalter, A. Möller, and M. Kranz. Towards a holistic approach for mobile application development in intelligent environments. In *Proceedings of the 10th International Conference on Mobile and Ubiquitous Multimedia*, MUM '11, pages 73–80. ACM, 2011. → pages 53

[84] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *18th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011. → pages 35, 47, 116

[85] M. Erfani Joorabchi and A. Mesbah. Reverse engineering iOS mobile applications. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 177–186. IEEE Computer Society, 2012. → pages iii, 9, 47, 48, 114, 164, 170, 173

[86] M. Erfani Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'13, pages 15–24. IEEE, 2013. → pages iii, 3, 4, 5, 6, 7, 8, 10, 85, 109, 138, 139, 165, 168, 171

[87] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! characterizing non-reproducible bug reports. In *The 11th Working Conference on Mining Software Repositories*, MSR'14, pages 62–71. ACM, 2014. → pages iii, 8, 57, 167

[88] M. Erfani Joorabchi, M. Ali, and A. Mesbah. Detecting inconsistencies in multi-platform mobile apps. In *Proceedings of the 26th International Symposium on Software Reliability Engineering*, ISSRE, pages 450–460. IEEE Computer Society, 2015. → pages iv, 9, 47, 48, 85, 113, 137, 171

[89] D. Falessi, M. A. Babar, G. Cantone, and P. Kruchten. Applying empirical software engineering to software architecture: challenges and lessons learned. *Empirical Softw. Engg.*, 15(3):250–276, 2010. → pages 54

[90] A. C. C. Franca, D. E. S. Carneiro, and F. Q. B. da Silva. Towards an explanatory theory of motivation in software engineering: A qualitative case study of a small software company. *2012 26th Brazilian Symposium on Software Engineering*, 0:61–70, 2012. → pages 54

[91] D. Franke and C. Weise. Providing a Software Quality Framework for Testing of Mobile Applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 431–434. IEEE Computer Society, 2011. → pages 115

[92] D. Franke, C. Elsemann, S. Kowalewski, and C. Weise. Reverse engineering of mobile application lifecycles. In *18th Working Conference on Reverse Engineering (WCRE)*, 2011. → pages 11, 50, 51

[93] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining, KDD '13, pages 1276–1284. ACM, 2013. → pages 112

[94] R. Gallo, P. Hongo, R. Dahab, L. C. Navarro, H. Kawakami, K. Galvão, G. Junqueira, and L. Ribeiro. Security and system architecture: Comparison of Android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 12:1–12:6. ACM, 2015. → pages 109

[95] L. Galvis Carreno and K. Winbladh. Analysis of user comments: An approach for software requirements evolution. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 582–591. IEEE Computer Society, 2013. → pages 50, 112

[96] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 171–180. ACM, 2012. → pages 80

[97] A. Gimblett and H. Thimbleby. User interface model discovery: towards a generic approach. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '10, pages 145–154. ACM, 2010. → pages 4, 117

[98] B. Glaser. *Doing Grounded Theory: Issues and Discussions*. Sociology Press, 1998. → pages 13

[99] B. Glaser and A. Strauss. *The discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 1967. → pages 3, 11, 12, 13, 45, 166

[100] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 82–91. IEEE Press, 2013. → pages 4, 47, 163

[101] P. Gokhale and S. Singh. Multi-platform strategies, approaches and challenges for developing mobile applications. In *Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014 International Conference on*, pages 289–293. IEEE, April 2014. → pages 47, 53, 54

[102] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81. IEEE Press, 2013. → pages 47, 164, 173

[103] Google Play Store Crawler. https://github.com/MarcelloLins/GooglePlayAppsCrawler. Accessed: 2015-12-15. → pages 87

[104] Goole Play Store Review scraper. https://github.com/jkao/GooglePlayScraper. Accessed: 2015-12-15. → pages 92

[105] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035. ACM, 2014. → pages 2, 47, 50, 52, 85, 112

[106] M. Greiler, A. van Deursen, and M.-A. Storey. Test confessions: a study of testing practices for plug-in systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 244–254. IEEE Computer Society, 2012. → pages 2, 12, 54

[107] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 100–110. IEEE Press, 2015. → pages 47

[108] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 495–504. ACM, 2010. → pages 58, 80

[109] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. 'not my bug!' and other reasons for software bug report reassignments. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW, pages 395–404. ACM, 2011. → pages 58, 80, 81

[110] E. Guzman and W. Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 153–162, 2014. → pages 47, 112

[111] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *19th Working Conference on Reverse Engineering (WCRE), 2012*, pages 83–92. IEEE, 2012. → pages 109

[112] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 83–92. IEEE, Oct 2012. → pages 2, 47, 50, 53

[113] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: MSR for app stores. In *9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111. IEEE, June 2012. → pages 50, 112

[114] Z. Hemel and E. Visser. Declaratively programming the mobile web with Mobl. In *Proceedings of Intl. Conf. on Object oriented programming systems languages and applications (OOPSLA)*, pages 695–712. ACM, 2011. → pages 48

[115] S. Herbold, J. Grabowski, S. Waack, and U. Bünting. Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 232–241. IEEE Computer Society, 2011. → pages 79, 82

[116] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 392–401. IEEE Computer Society, 2013. → pages 58, 66, 79, 80, 81

[117] E. Holder, E. Shah, M. Davoodi, and E. Tilevich. Cloud twin: Native execution of Android applications on the Windows Phone. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 598–603. IEEE, 2013. → pages 47, 163

[118] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 34–43. ACM, 2007. → pages 80

[119] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83. ACM, 2011. → pages 4, 47, 48, 118

[120] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 18:1–18:15. ACM, 2014. → pages 47, 49

[121] N. P. Huy and D. vanThanh. Evaluation of mobile app paradigms. In *Proceedings of the International Conference on Advances in Mobile Computing and Multimedia*, MoMM, pages 25–30. ACM, 2012. → pages 53, 54

[122] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 41–44. IEEE Press, 2013. → pages 112

[123] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 41–44. IEEE Press, 2013. → pages 47

[124] C. Iacob, V. Veerappa, and R. Harrison. What are you complaining about?: A study of online reviews of mobile applications. In *Proceedings of the 27th International BCS Human Computer Interaction Conference*, BCS-HCI '13, pages 29:1–29:6. British Computer Society, 2013. → pages 2, 4, 85, 112

[125] iOS Developer Library. Apple's developer guides. https://developer.apple.com/library/ios/navigation/. Accessed: 2015-12-15. → pages xii, 117, 118, 122

[126] iTunes App Store Review scraper. https://github.com/grych/AppStoreReviews. Accessed: 2015-12-15. → pages 92

[127] iTunes RSS Feed Generator. https://rss.itunes.apple.com/ca/. Accessed: 2015-12-15. → pages 91

[128] M. Janicki, M. Katara, and T. Paakkonen. Obstacles and opportunities in deploying model-based gui testing of mobile software: a survey. *Software Testing, Verification and Reliability*, 22(5):313–341, 2012. → pages 47, 115

[129] Y.-W. Kao, C.-F. Lin, K.-A. Yang, and S.-M. Yuan. A cross-platform runtime environment for mobile widget-based application. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 68 –71, 2011. → pages 53

[130] K. Karhu, T. Repo, O. Taipale, and K. Smolander. Empirical observations on software testing automation. In *Proceedings of the International Conference on Software Testing Verification and Validation (ICST)*, pages 201–209. IEEE Computer Society, 2009. → pages 12, 54, 55

[131] J. Kasurinen, O. Taipale, and K. Smolander. Software test automation in practice: empirical observations. *Adv. Soft. Eng.*, 2010:4:1–4:13, 2010. → pages 54

[132] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander. A study on agility and testing processes in software organizations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 231–240. ACM, 2010. → pages 12, 45, 54, 166

[133] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 36–47. ACM, 2014. → pages 2, 3, 11, 12, 54

[134] H. Khalid. On identifying user complaints of iOS apps. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1474–1476. IEEE Press, 2013. → pages 50, 52, 112

[135] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the devices to test your app on: a case study of Android game apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 610–620. ACM, 2014. → pages 47, 52

[136] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan. What do mobile app users complain about? a study on free iOS apps. *IEEE Software*, 99, 2014. → pages 50, 52, 112

[137] H. Kim, B. Choi, and W. Wong. Performance testing of mobile applications at the unit test level. In *Proceedings of the 3rd International Conference on Secure Software Integration and Reliability Improvement*, pages 171–180. IEEE Computer Society, 2009. → pages 115

[138] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society, 2015. → pages 2, 11, 50, 51

[139] T. A. Kroeger, N. J. Davidson, and S. C. Cook. Understanding the characteristics of quality for software engineering processes: A grounded theory investigation. *Inf. Softw. Technol.*, 56(2):252–271, Feb. 2014. → pages 12, 54

[140] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with Android and symbian. In *Software Reliability Engineering (ISSRE), IEEE International Symposium on*, pages 249–258. IEEE, Nov 2010. → pages 50, 112

[141] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile OSes: A case study with Android and Symbian. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 249–258. IEEE Computer Society, 2010. → pages 80

[142] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 297–308. USENIX Association, 2013. → pages 47

[143] D. Lavid Ben Lulu and T. Kuflik. Functionality-based clustering using short textual description: Helping users to find apps installed on their mobile device. In *Proceedings of the 2013 International Conference on Intelligent User Interfaces*, IUI '13, pages 297–306. ACM, 2013. → pages 2, 50, 85, 112

[144] V. L. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707–710, 1996. → pages 149

[145] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a Google case study. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 372–381. IEEE Computer Society, 2013. → pages 80

[146] C.-J. M. 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 *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 519–530. ACM, 2014. → pages 47, 49

[147] 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 *Proceedings of the International Symposium on the Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487. ACM, 2013. → pages 47, 50, 53, 112

[148] M. Linares-Vasquez, B. Dit, and D. Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 93–96. IEEE, May 2013. → pages 2, 50, 52

[149] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the Android SDK. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 83–94. ACM, 2014. → pages 2, 50, 52

[150] 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 *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122. IEEE Press, 2015. → pages 51

[151] M. Linares-Vsquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *Proceedings of 31st IEEE International Conference on Software Maintenance and Evolution*, ICSME'15. IEEE, 2015. → pages 47

[152] W. Maalej and H. Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *IEEE 23rd International Requirements Engineering Conference (RE), 2015*, pages 116–125. IEEE, 2015. → pages 47, 50

[153] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234. ACM, 2013. → pages 47

[154] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609. ACM, 2014. → pages 47

[155] L. Martie, V. Palepu, H. Sajnani, and C. Lopes. Trendy bugs: Topic trends in the Android bug reports. In *9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 120–123, June 2012. → pages 2, 47, 50, 53

[156] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. The app sampling problem for app store mining. In *12th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015. → pages 50, 112

[157] E. Masi, G. Cantone, M. Mastrofini, G. Calavaro, and P. Subiaco. Mobile apps development: A framework for technology decision making. In *Proceedings of International Conference on Mobile Computing, Applications, and Services.*, MobiCASE'4, pages 64–79, 2012. → pages 1, 47, 53, 54

[158] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the Android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79. IEEE Computer Society, 2013. → pages 47, 53

[159] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, pages 260–269. IEEE, 2003. → pages 4, 117

[160] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE'12)*, pages 408–418. IEEE Computer Society, 2012. → pages 117

[161] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11)*, pages 561–570. ACM, 2011. → pages 4, 136, 163

[162] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 561–570. ACM, 2011. → pages 48

[163] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. In *ACM Transactions on the Web (TWEB)*, volume 6, pages 3:1–3:30. ACM, 2012. → pages 4, 117, 129, 163

[164] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35 –53, 2012. → pages 117

[165] Mining iOS and Android mobile app-pairs: Toolset and dataset. https://github.com/saltlab/Minning-App-Stores. Accessed: 2015-12-15. → pages 85, 87, 96, 111, 112

[166] 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 *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2014, pages 50–53. ACM, 2014. → pages 2, 11, 50, 51

[167] I. Mojica Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan. Impact of ad libraries on ratings of android mobile apps. *Software, IEEE*, 31(6):86–92, 2014. → pages 47

[168] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 673–686. ACM, 2015. → pages 47, 50

[169] H. Muccini, A. D. Francesco, and P. Esposito. Software Testing of Mobile Applications: Challenges and Future Research Directions. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*. IEEE Computer Society, 2012. → pages 2, 11, 50, 51, 115

[170] B. A. Myers and M. B. Rosson. Survey On User Interface Programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI'92, pages 195–202. ACM, 1992. → pages 115, 162, 169

[171] S. N. Nader Boushehrinejadmoradi, Vinod Ganapathy and L. Iftode. Testing cross-platform mobile app development frameworks. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2015. ACM, 2015. → pages 47

[172] M. Nagappan and E. Shihab. Future trends in software engineering research for mobile apps. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering, FoSE*, 2016. → pages 111, 113

[173] S. Nath. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 59–73. ACM, 2015. → pages 47

[174] S. Nath, F. X. Lin, L. Ravindranath, and J. Padhye. Smartads: Bringing contextual ads to mobile apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 111–124. ACM, 2013. → pages 47

[175] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen. A preliminary survey on software testing practices in Australia. In *Proceedings of the Australian Software Engineering Conference*, pages 116–125. IEEE Computer Society, 2004. → pages 54

[176] N. Nikzad, O. Chipara, and W. G. Griswold. Ape: An annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 515–526. ACM, 2014. → pages 47

[177] A. Nistor and L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 282–292. ACM, 2014. → pages 47

[178] NPD DisplaySearch. Smartphones to pass 80% of global mobile phone shipments by 2017. http://en.ofweek.com/news/Smartphones-to-pass-80-of-global-mobile-phone-shipments-by-2017-3354. Accessed: 2015-12-15. → pages 114

[179] A. Onwuegbuzie and N. Leech. Validity and qualitative research: An oxymoron? *Quality and Quantity*, 41:233–249, 2007. → pages 45, 166

[180] T. Paananen. Smartphone Cross-Platform Frameworks. Bachelor's Thesis., 2011. → pages 53

[181] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 125–134. IEEE, July 2013. → pages 47, 50, 53, 112

[182] M. Palmieri, I. Singh, and A. Cicchetti. Comparison of cross-platform mobile development tools. In *Intelligence in Next Generation Networks (ICIN), 2012 16th International Conference on*, pages 179 –186, 2012. → pages 53

[183] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015. → pages 112

[184] F. Palomba, M. Linares-Vsquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *Proc. ICSME*, pages 291–300. IEEE, 2015. → pages 47, 50

[185] S. Panichella, A. D. 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 *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015. → pages 2, 4, 85, 92, 103, 112

[186] G. P. Picco, C. Julien, A. L. Murphy, M. Musolesi, and G.-C. Roman. Software engineering for mobility: Reflecting on the past, peering into the future. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 13–28. ACM, 2014. → pages 2

[187] A. Puder and O. Antebi. Cross-compiling Android applications to iOS and Windows phone 7. *Mob. Netw. Appl.*, 18(1):3–21, Feb. 2013. → pages 47, 53, 54

[188] K. Rasmussen, A. Wilson, and A. Hindle. Green mining: Energy consumption of advertisement blocking methods. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 38–45. ACM, 2014. → pages 47

[189] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220. ACM, 2013. → pages 35

[190] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th*

Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14, pages 190–203. ACM, 2014. → pages 49

[191] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550. ACM, 2011. → pages 12, 54

[192] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE Computer Society, 2012. → pages 4, 7, 115, 162, 169

[193] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej. Monitoring user interactions for supporting failure reproduction. In *International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE, 2013. → pages 79, 82

[194] I. Ruiz, M. Nagappan, B. Adams, and A. Hassan. Understanding reuse in the Android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122. IEEE, June 2012. → pages 2, 47, 50, 85, 112

[195] I. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan. On the relationship between the number of ad libraries in an android app and its rating. *IEEE Software*, 99, 2014. → pages 2, 47, 50, 85, 112

[196] A. Sadeghi, N. Esfahani, and S. Malek. Mining the categorized software repositories to improve the analysis of security vulnerabilities. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 155–169. Springer-Verlag, 2014. → pages 47

[197] F. Sarro, A. Al-Subaihin, M. Harman, Y. Jia, W. Martin, and Y. Zhang. Feature lifecycles as they spread, migrate, remain, and die in app stores. In *Proc. RE*, pages 76–85. IEEE, 2015. → pages 50

[198] Scikit Learn: Machine Learning in Python. http://scikit-learn.org/stable/index.html. Accessed: 2015-12-15. → pages 93

[199] C. Seaman. Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on*, 25(4):557–572, Jul 1999. → pages 3, 11

[200] S. Seneviratne, A. Seneviratne, M. A. Kaafar, A. Mahanti, and P. Mohapatra. Early detection of spam mobile apps. In *Proceedings of the 24th International Conference on World Wide Web*, WWW, pages 949–959. ACM, 2015. → pages 2, 85, 112

[201] H. Seo and S. Kim. Predicting recurring crash stacks. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 180–189. ACM, 2012. → pages 80

[202] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 249–258. IEEE Computer Society, 2010. → pages 80

[203] I. Steinmacher, T. Uchoa Conte, and M. Gerosa. Understanding and supporting the choice of an appropriate task to start with in open source software communities. In *48th Hawaii International Conference on System Sciences (HICSS)*, pages 5299–5308. IEEE, 2015. → pages 2, 12, 54

[204] M. Sulayman, C. Urquhart, E. Mendes, and S. Seidel. Software process improvement success factors for small and medium web companies: A qualitative study. *Inf. Softw. Technol.*, 54(5):479–500, 2012. → pages 3, 11, 12, 54

[205] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for Dynamic Analysis of iOS Applications. In *Proceedings of the Workshop on Open Research Problems in Network Security (iNetSec)*, pages 65–77, 2011. → pages 47, 116, 120, 126, 161, 172

[206] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for dynamic analysis of iOS applications. In *Proceedings of the 2011 IFIP WG 11.4 International Conference on Open Problems in Network Security*, iNetSec'11, pages 65–77. Springer-Verlag, 2012. → pages 50

[207] K. Thomas, A. K. Bandara, B. A. Price, and B. Nuseibeh. Distilling privacy requirements for mobile applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 871–882. ACM, 2014. → pages 35

[208] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. What are the characteristics of high-rated apps? a case study on free android

applications. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015. → pages 112

[209] Top Free in Android Apps. https://play.google.com/store/apps/collection/topselling_free?hl=en. Accessed: 2015-12-15. → pages 91

[210] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233. ACM, 2014. → pages 107

[211] P. M. Vu, T. T. Nguyen, H. V. Pham, and T. T. Nguyen. Mining user opinions in mobile app reviews: A keyword-based approach. *CoRR*, abs/1505.04657, 2015. → pages 112

[212] W. Wang and M. W. Godfrey. Detecting api usage obstacles: A study of iOS and Android developer questions. In *Proceedings of the10th Working Conference on Mining Software Repositories*, MSR '13, pages 61–64. IEEE Press, 2013. → pages 2, 50, 52

[213] A. I. Wasserman. Software engineering issues for mobile application development. In *FSE/SDP workshop on Future of software engineering research*, FoSER'10, pages 397–400. ACM, 2010. → pages 1, 11, 50, 51, 115

[214] M. Waterman, J. Noble, and G. Allan. How much up-front? a grounded theory of agile architecture. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), 2015*, volume 1, pages 347–357. IEEE, 2015. → pages 54

[215] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on Android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 623–634. ACM, 2013. → pages 109

[216] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99. IEEE Press, 2015. → pages 47

[217] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the*

International Conference on Fundamental Approaches to Software Engineering (FASE), pages 250–265. Springer-Verlag, 2013. → pages 47, 48, 164, 173

[218] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015. ACM, 2015. → pages 47, 50, 112

[219] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of ESEC/FSE*, pages 26–36. ACM, 2011. → pages 58, 80

[220] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 595–605. IEEE Computer Society, 2012. → pages 38

[221] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '12, pages 93–104. ACM, 2012. → pages 47

[222] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1074–1083. IEEE Computer Society, 2012. → pages 58, 80, 81