Studying User-Developer Interactions Through the Updating and Reviewing Mechanisms of the Google Play Store

by

Safwat Mohamed Ibrahim Hassan

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

September 2018

Abstract

MOBILE app stores (such as the Apple App Store and the Google Play Store) provide a unique updating mechanism that helps app developers to distribute their updates efficiently. In addition, after downloading a new update, users are able to review the latest update so other users could benefit from the posted reviews. Such unique reviewing and updating mechanisms enable users and developers to interact with each other through the store.

In this thesis, we study user-developer interactions through the updating and reviewing mechanisms of app stores. Our studies can help store owners to acquire a global view about user-developer interactions. Such a global view about user-developer interactions can help store owners to improve the quality of the offered apps in their stores. For example, store owners can leverage our findings to identify the common release mistakes that are made by app developers and improve the app updating mechanism to prevent developers from making such mistakes in their updates. In particular,

we study the user-developer interactions along three perspectives: (1) study the common developer mistakes that lead to emergency updates, (2) study how the reviewing mechanism can help spot good and bad updates, and (3) study the dialogue between users and developers to help design next-generation app reviewing mechanisms.

In this thesis, we identify eight patterns of emergency updates in two categories "Updates due to deployment issues" and "Updates due to source code changes". Our studies show that it can be worthwhile for app owners to respond to reviews, as responding may lead to an increase in the given rating. In addition, we identify four patterns of developer responses. Our work demonstrates the necessity of an update-level analysis of reviews to capture the impressions of an app's user-base about a particular update. An app-level analysis is not sufficient to capture these transient impressions.

Acknowledgments

I wish to express my gratitude to Allah (God) for his blessings and his help to complete this work. I am incredibly thankful to my supervisor Prof. Ahmed E. Hassan who gave me the opportunity to do my Ph.D. at Queen's University. I appreciate his great support and continuous help throughout my Ph.D. study. At the beginning of my study, I picked up a point that looked extremely difficult and infeasible, but he continues to believe in my skills, he helped me a lot, and he guided me throughout my Ph.D. study until I finished this work. I consider myself lucky to work under his supervision, and I appreciate his tremendous effort and support.

I would like to thank my supervising committee members, Prof. Hossam Hassanein and Prof. Patrick Martin for their help and their insightful guidance and discussions. I also would like to thank my examination committee members for their valuable feedback and their useful comments.

I am very honored to have the chance to work and collaborate with the brightest

researchers during my Ph.D. study. I would like to thank all of my collaborators, Prof. Abram Hindle, Prof. Weiyi Shang, Prof. Chakkrit Tantithamthavorn, and Prof. Cor-Paul Bezemer for their great help. I also would like to thank my labmates, Hammam AlGhamdi, Heng Li, Dr. Mohamed Sami Rakha, Suhas Kabinna, Ravjot Singh, Sumit Sourav, Dr. Daniel Da Costa, Prof. Yasutaka Kamei, Dr. Gustavo Ansaldi Oliva, Dr. Wang Shaowei, Stuart McIlroy, Md Ahasanuzzaman, and Filipe Côgo for their great support. I also thank my friends, Dr. Shadi Khalifa, Wenyan Wu, Tarek Mamdouh, Dr. Ahmed Youssef, and Mahmoud Ragab for their great help.

I appreciate Microsoft Azure and Compute Canada for providing me with the needed resources and infrastructure to perform the data collection and analysis for my studies.

A special thanks to my mother, my father, my brothers and my family for their endless support, love, help, and appreciation at every moment in my Ph.D. study. I appreciate their exceptional care and their assistance primarily in the hardest moments of my Ph.D. study. I believe that this work could not be done without their great help, their endless love and their continuous praying for me.

Dedication

To my beloved parents who supported me at every moment throughout my Ph.D. journey.

Co-authorship

Earlier versions of the work in the thesis were published as listed below:

1. An Empirical Study of Emergency Updates for Top Android Mobile Apps (Chapter 4)

   Safwat Hassan, Weiyi Shang, and Ahmed E. Hassan. Empirical Software Engineering Journal (EMSE), 2017.

2. Studying Bad Updates of Top Free-to-Download Apps in the Google Play Store (Chapter 5)

   Safwat Hassan, Cor-Paul Bezemer, and Ahmed E. Hassan. IEEE Transactions on Software Engineering (TSE), 2018.

3. Studying the Dialogue Between Users and Developers of Free Apps in the Google Play Store (Chapter 6)

   Safwat Hassan, Chakkrit Tantithamthavorn, Cor-Paul Bezemer, and Ahmed E.

Hassan. Empirical Software Engineering Journal (EMSE), 2018.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

MOBILE app stores (such as the Google Play Store and the Apple App Store) provide a unique updating mechanism that enables app developers to easily publish and distribute their updates. In addition, after downloading a new update, users can post their feedback (i.e., review) so other users could benefit from the posted reviews. Such unique updating and reviewing mechanisms enable users and developers to interact with each other through the store. In particular, users can (1) download the latest updates and (2) post their reviews about updates. In addition, developers can (1) publish updates and (2) respond to user reviews.

The interactions between users and developers within the stores occur at a rapid pace. Every day, there are new updates being published, and users posting their reviews about these updates. Developers benefit from such reviews to improve their apps

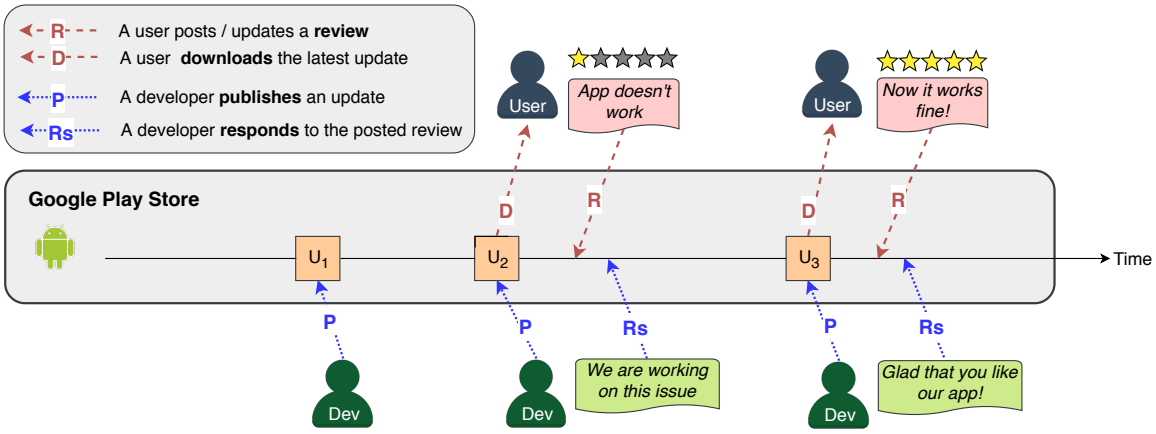

Figure 1.1: An example of the user-developer interactions through the Google Play Store

(or associated documentation), or fix any issues in their new update. Figure 1.1 illustrates the user-developer interactions through the Google Play Store. As shown in Figure 1.1, a developer publishes a new update. Then, the Google Play Store distributes the update to the current app users. After downloading the latest update of an app, users can post their feedback (e.g., *"App does not work!"*). This feedback offers insights into how users perceive a new update. For example, app developers could benefit from the reviews and quickly publish an emergency update to fix any widely raised issues.

1.1 Thesis Statement

App stores provide us with a unique opportunity to explore user-developer interactions and the impact of such interactions on the evolution of mobile apps. In this thesis, we study user-developer interactions through the updating and reviewing mechanisms of app stores. Our studies can help store owners improve the overall quality of the offered apps in their stores by better understanding the interactions between

app users and developers. Providing high-quality apps is essential not only for app developers but also for store owners. For example, in September 2016, the Apple App Store started a continuous app store cleaning process that removed outdated or non-functional apps (Apple, 2018a). This cleaning process resulted in removing hundreds of thousands of apps from the Apple App Store (Miller, 2017).

In particular, we study the user-developer interactions along three perspectives: (1) studying the common developer mistakes that lead to emergency updates, so that store owner can develop mechanisms to prevent app developers from repeating such mistakes, (2) studying how the reviewing mechanism can help spot good and bad updates, so store owners can leverage user reviews to proactively limit the distribution of bad updates, and (3) studying the dialogue between users and developers to help design next-generation app reviewing mechanisms.

Thesis Statement: *Studying user-developer interactions through the updating and reviewing mechanisms of app stores can help store owners improve the overall perceived quality of the provided apps in their stores and enhance the overall experience of app users.*

1.2 Thesis Overview

In this section, we provide an outline of our thesis.

1.2.1 Chapter 2: Background

This chapter gives background information about the user-developer dialogue in the Google Play Store.

1.2.2 Chapter 3: Literature Survey

In this chapter, we provide an overview of prior research that is related to our work. In particular, we focus on prior research in the following four areas:

1. **Bugs in mobile apps.** We summarize prior research that studies issues (e.g., user interface issues) in mobile apps. The importance of analyzing issues in mobile apps motivates us in studying the common developer mistakes that lead to emergency updates.

2. **User reviews of mobile apps.** We summarize prior research that analyzes user reviews to extract useful information such as bug reports and feature requests. From our survey, we observed that prior research incorrectly assumes that reviews are static in nature and users never update their reviews. In this thesis, we examine changes in the posted reviews to understand how users perceive every update and to leverage users' feedback about every update.

3. **Characteristics of successful apps.** We summarize prior research that studies the characteristics of successful apps (i.e., apps with high ratings). From our survey, we observed that prior research analyzed the characteristics of successful apps by taking an app-centric view. Recent research noted the importance of understanding apps on an update-level (i.e., capturing the impressions of an app's user-base about a particular update). This motivates us to study how the reviewing mechanism can help in spotting good and bad updates.

4. **User-developer dialogue in app stores.** We summarize prior research that analyzes the dialogue between users and developers. From our survey, we observed that prior research noted that responding to a review often has a positive effect

on the rating that is given by the user to an app. This motivates us to revisit prior work by conducting a more in-depth study on a larger dataset.

1.2.3 Chapter 4: Studying the Common Developer Mistakes that Lead to Emergency Updates

In this chapter, we study emergency updates for top Android mobile apps as follows. First, we propose an approach for identifying emergency updates. Second, we study the characteristics of emergency updates. We find that the emergency updates often have a long lifetime (i.e., they are rarely followed by another emergency update). We also observe that updates preceding emergency updates often receive a higher ratio of negative reviews than the emergency updates. Finally, we identify eight patterns of emergency updates. We categorize these eight patterns along two categories *"Updates due to deployment issues"* and *"Updates due to source code changes"*. Our study can help store owners continuously identify the common developer mistakes that lead to emergency updates and prevent the distribution of updates with such mistakes.

1.2.4 Chapter 5: Studying How the Reviewing Mechanism Can Help Spot Good and Bad Updates

In this chapter, we propose an approach for leveraging user reviews to spot good and bad updates. Then, we study the characteristics of the top 250 bad updates (i.e., updates with the highest increase in the percentage of negative reviews relative to the prior updates of the app) to learn how users perceive a bad update and how developers could recover from bad updates. We find that feature removal and UI issues have

the highest increase in the percentage of negative reviews. We also observe that bad updates with crashes and functional issues are the most likely to be fixed by a later update. Our study can help store owners improve the quality of the apps in their stores by spotting good and bad updates.

1.2.5 Chapter 6: Studying the Dialogue Between Users and Developers

In this chapter, we perform an in-depth analysis of user-developer dialogue. First, we analyze the impact of responding to user reviews and the common reasons for a rating increase/decrease after a developer responds to a user review. Second, we perform a quantitative analysis of the likelihood of a developer responding. We observe that the likelihood of a developer responding to a review increases as the review rating gets lower or as the review content gets longer. Finally, we perform a qualitative analysis of what drives a developer to respond. We identify seven drivers that make a developer respond to a review, of which the most important ones are to thank the users for using the app and to ask the user for more details about the reported issue.

Understanding how user and developers leverage the reviewing mechanism through user-developer dialogue will help in building better next-generation reviewing mechanisms. For example, next-generation reviewing systems should somehow consider the changes in users' long-term impressions while calculating the rating of an app. Moreover, store owners should notify app developers about such changes so developers can track the changes in users' impressions about their app.

1.3 Thesis Contributions

In this thesis, we show the importance of analyzing user-developer interactions through the updating and reviewing mechanisms of app stores to gain a better understanding of how users and developers leverage these mechanisms. We demonstrate that this in-depth analysis of user-developer interactions can help store owners improve the overall quality of the provided apps in their stores and enhance the overall experience of app users. In particular, our main contributions are as follows:

1. This thesis is the first work to demonstrate the dynamic nature of reviews. In addition, our work is the first work that leverage this dynamic nature to (1) spot good/bad updates and analyze the characteristics of these updates and (2) analyze user-developer dialogues.

2. We show that developers share common mistakes and such common mistakes can be identified using the global view of emergency updates in the Google Play Store. We identify eight patterns of emergency updates. Store owners can benefit from our studies to develop mechanisms to prevent app developers from repeating such mistakes.

3. Our work is the first work to deeply explore developer responses to user reviews. Furthermore, we are the first to demonstrate an interesting use of the app-review mechanism as a user support medium. Finally, our identification of similar developer-responses highlights the importance of providing automated responses in next-generation app-review mechanisms.

4. We demonstrate the importance of update-level analysis compared to the traditional app-level view that is done by most prior work. Furthermore, we propose

an approach for identifying good/bad updates. In addition, we perform an in-depth analysis of the characteristics of bad updates and how developers recovered from bad updates.

CHAPTER 2

Background

In this chapter, we give background information for our study by explaining how the user-developer dialogue in the Google Play Store works.

Figure 2.1 shows an abstraction of the user-developer dialogue in the Google Play Store. A user initiates the dialogue by posting a review, including a rating, for an app (S_1). User reviews can contain valuable information such as bug reports, feature requests, complaints or praise about the app (Maalej and Nabil, 2015). In turn, the app developer engages in the dialogue by responding to the review (S_2). The Google Play Store notifies a user when a developer responds to the user review. Both the user review and the developer responses can be updated at any time. Hence, by considering the changes made to the review and the response in a chronological order, we can see the dialogue between the user and developer. Note that the app store shows only the

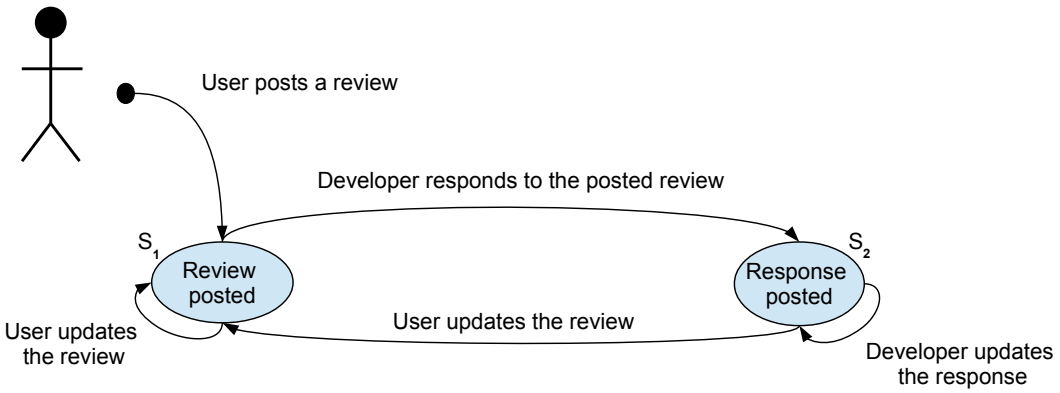

Figure 2.1: User-developer dialogue in the Google Play Store

latest versions of the review and the response. Hence, to study such a dialogue we must continuously crawl the app store reviews to spot any changes to user reviews or developer responses. At the start of our research, the Google Play Store is the only store with support for developer responses. Apple's App Store is expected to add support for responding to reviews soon (Perez, 2017).

The Google Play Store sends an email notification to an app user when a developer responds to their review (i.e., when the user-developer dialogue changes from state S_1 to state S_2) (Google, 2018c). In addition, the Google Play Store sends an email notification to an app developer whenever a user changes their posted review after a developer response (i.e., when the user-developer dialogue changes from state S_2 to state S_1). However, the store does not notify developers when users keep updating their review (i.e., remain in state S_1). In addition, the store does not notify users when developers keep changing their response (i.e., remain in state S_2). In summary, developers and users are not always notified when a review or response changes.

Table 2.1: User-developer dialogue for the *AppLock* app

Dialogue	State	Rating
User: *"Applock stopped locking apps suddenly.... help"*	S_1	★★☆☆☆
Developer: *"Hi Vikhyath, thanks for using AppLock. To solve this problem, please open phone settings, then select security, then select apps with usage access, enable AppLock."*	S_2	-
User: *"Applock stopped locking apps suddenly.... help . edit: why is it asking for usage access all of a sudden ? Reply please."*	S_1	★★☆☆☆
Developer: *"Hi Vikhyath, since Lollipop system, there are some changes in Android system. In order to make AppLock work, please open phone settings, then select security, then select apps with usage access, enable AppLock."*	S_2	-
User: *"Thank you .thank you for response"*	S_1	★★★★★
Developer: *"Hi Vikhyath, thanks for your support all along. :) We will keep working to provide the best user experience. Have a nice day!"*	S_2	-

Table 2.1 shows an example of how such a dialogue emerges when a user or a developer update their review or response for the *AppLock* app. The dialogue shows how a user raises an issue in his review and how the developer helps the user resolve that issue. After resolving the issue, the user increases the given rating from 2 stars to 5 stars out of 5 stars.

CHAPTER 3

Literature Survey

In this chapter, we give an overview of prior research that is related to our work. In particular, we focus on prior research in the area of (1) bugs in mobile apps, (2) user reviews of mobile apps, (3) characteristics of successful apps and (4) user-developer dialogue in app stores.

3.1 Bugs in Mobile Apps

In order to minimize the bugs in mobile apps, prior research studies crashes and bugs in mobile apps. Guana et al. (2012) analyzed data for 20,169 bugs in the Android platform repository. They found that the framework layer contains a higher number of bugs than the kernel layer. Han et al. (2012) studied the Android platform bugs that

are reported for the HTC and Motorola devices. They manually labeled the bugs and applied Latent Dirichlet Allocation (LDA) and Labeled LDA techniques in order to generate topics from the bug reports. Han et al. identified 57 topics and 72 topics for the Motorola and HTC devices respectively. Han et al. found 14 common topics between the Motorola and HTC devices.

Syer et al. (2013) studied the differences between mobile apps, desktop/server applications. Syer et al. studied 15 open-source mobile apps (from the Google Play Store and F-Droid apps repositories) and five desktop/server applications. Syer et al. found that mobile apps have a smaller code base than desktop/server applications. Syer et al. found that the reported bugs of mobile apps are fixed faster than in desktop/server applications. Syer et al. (2015) studied the relation between the use of Android APIs and the probability of bugs in mobile apps. Syer et al. found that source code files that have a higher dependence on the Android APIs are more bug-prone than other files. Thus, Syer et al. recommended the prioritization of code review efforts on source code files that heavily depend on the Android APIs.

Ravindranath et al. (2012) proposed an *AppInsight* tool that analyzes a mobile app and identifies performance bottlenecks. Their study included the analysis of the usage of 30 users of 30 apps over a four-months period and leveraged the AppInsight tool to identify the performance bottlenecks in these studied apps. Vásquez et al. (2013) studied the fault-proneness of the used APIs on an app's rating. They observed a high correlation between an app's rating and the change and fault-proneness of the APIs that are used by the app (Vásquez et al., 2013; Bavota et al., 2015).

The limited battery power of mobile devices may impact the user experience if some app features (e.g., the mobile camera) have a high battery consumption (Wan

et al., 2015). Researchers studied energy bugs and energy hotspots in mobile apps (Pathak et al., 2011; Banerjee et al., 2014; Pathak et al., 2012; Wan et al., 2015). According to Banerjee et al. (2014) energy bugs are defined as the cases when the mobile device resources are still used although the app is no longer active, while energy hotspots are defined as the cases when an app causes high energy consumption although the resource utilization is small. Pathak et al. (2011) proposed a taxonomy of smartphone energy bugs. Pathak et al. found that there is a variation in the types and causes of energy bugs. Pathak et al. provided a roadmap for developing a framework that identifies the root-cause of energy bugs. Banerjee et al. (2014) developed a framework that generates test data to detect energy bugs and energy hotspots in mobile apps. Wan et al. (2015) identified UI screens that have more energy consumption than optimized screens. For example, optimized colors may reduce energy consumption. Wan et al. rank UI screens with respect to the difference in the energy consumption between the original UI screens and the energy-optimized screens, so developers can focus on top energy consuming screens to reduce the consumption of energy.

Prior research provides best practices and tools that help avoid crashes and bugs in mobile apps. The importance of analyzing issues in mobile apps motivates us in studying the common developer mistakes that lead to emergency updates, so that store owner can develop mechanisms to prevent app developers from repeating such mistakes.

3.2 User Reviews of Mobile Apps

Researchers often analyze user reviews to extract useful information such as complaints and feature requests (Oh et al., 2013; Iacob and Harrison, 2013; Iacob et al., 2013b,a;

Maalej and Nabil, 2015; Villarroel et al., 2016; Keertipati et al., 2016; Khalid, 2013; Khalid et al., 2015; McIlroy et al., 2016b). Table 3.1 summarizes prior studies which analyze user reviews. For example, Iacob and Harrison (2013); Iacob et al. (2013a) proposed MARA (Mobile App Review Analyzer) which uses linguistic rules to identify reviews that contain bug reports or feature requests.

Maalej and Nabil (2015) used different techniques to extract features from user reviews (such as review rating). Then Maalej and Nabil used different algorithms (such as Naive Bayes and decision tree) to label reviews into four categories: (1) feature request, (2) bug report, (3) user experience, and (4) app rating based on the extracted features. Maalej and Nabil's evaluated their approach using 4,400 manually-labeled reviews. Their approach achieves a precision that ranges from 70% to 95% and a recall that ranges from 80% to 90% based on the technique that is used to classify reviews.

Khalid et al. (2015) studied user complaints in mobile apps and identified 13 issue types (e.g., crashes and bug reports) that were raised in user reviews. McIlroy et al. (2016b) improved the taxonomy of issue types that was identified by Khalid et al. and proposed an approach to automatically classify reviews into the corresponding issue type. McIlroy et al. manually labeled 7,456 reviews of 24 apps in the Google Play Store and the Apple App Store to evaluate their approach. McIlroy et al. reported that their approach achieves a precision of 66% and a recall of 65% in classifying reviews into the corresponding issue types.

Panichella et al. (2015, 2016) proposed ARdoc (App Reviews Development Oriented Classifier) which classifies user reviews into five categories: (1) feature request, (2) bug report, (3) providing information, (4) requesting information, and (5) others. Similar to Maalej and Nabil's approach, Panichella et al. used different techniques (such as

Natural Language Processing (NLP) and sentiment analysis) to extract features from user reviews. Then, Panichella et al. built models that classify reviews to the aforementioned five categories based on the extracted features. Finally, Panichella et al. evaluated their approach by manually labeling 1,421 sentences from the reviews of seven apps (Panichella et al., 2015). Panichella et al. observed that combining machine learning techniques (e.g., sentiment analysis and text analysis) achieves higher accuracy than using a single technique. Panichella et al. evaluated ARdoc on reviews of different mobile apps and observed that ARdoc could achieve a precision that ranges from 84% to 89% and a recall that ranges from 84% to 89%.

Sorbo et al. (2017, 2016) proposed SURF (Summarizer of User Reviews Feedback) which is based on Panichella et al.'s approach (Panichella et al., 2015, 2016). First, the SURF technique labels reviews into the five categories that were proposed by Panichella et al. (2015, 2016). Then, Sorbo et al. manually analyzed 1,390 reviews and identified 12 topics that are mentioned in these reviews. Finally, SURF identifies which topic of these 12 topics (e.g., pricing) is mentioned in every review. Di Sorbo et al.'s approach is useful for app developers to automatically filter reviews that are related to a certain category (e.g., feature request) and a certain topic (e.g., pricing).

Villarroel et al. (2016) proposed CLAP (Crowd Listener for releAse Planning) which classifies reviews into three issue types: (1) bug report, (2) feature request, and (3) others. Later, Scalabrino et al. (2017) improved CLAP to classify reviews into seven types: (1) bug report, (2) feature request, (3) performance issues, (4) energy issues, (5) security issues, (6) usability issues, and (7) others. CLAP groups similar reviews in every issue type (e.g., group all reviews that raise the same energy complaint). Finally, CLAP prioritizes the identified groups based on different factors (such as the average rating of all

reviews in every group). CLAP is useful for app developers to plan for the next release by selecting the most important raised complaints in a particular issue type (e.g., most important performance issue that was raised in the reviews).

Chen et al. (2014) proposed AR-Miner (Automatic Review Miner) that filters out non-informative reviews and groups similar reviews based on topic extraction. The AR-Miner approach ranks topics based on different criteria such as the number of reviews containing this topic or the average rating of the topic. Finally, AR-Miner displays the identified topics. The proposed approach is useful for app developers to identify raised topics over time and easily select reviews that are related to a certain topic.

Later, Palomba et al. (2015) leveraged Chen et al.'s technique to filter non-informative reviews and proposed the CRISTAL technique. CRISTAL links user reviews to the corresponding code changes (i.e., code commits and bug reports) using text similarity. Palomba et al. applied CRISTAL to 100 apps and observed that implementing features that are requested in user reviews leads to a rating increase. Palomba et al. results suggest that developers should leverage reviews analysis tools to continuously highlight the requested features and implement these features to improve their app rating. Palomba et al. (2018) extended their study by surveying app developers whether they consider the requested features in user reviews. Palomba et al. observed that at least 75% of the surveyed developers mentioned that they frequently consider the features that are requested by users. Later, Palomba et al. (2017) proposed the CHANGEADVISOR approach to group user reviews that request similar features and map these reviews to the corresponding source code. The CHANGEADVISOR approach was applied to ten apps and the CHANGEADVISOR approach achieves high accuracy (81% precision and 70% recall) in mapping the requested features from user reviews to the corresponding

Table 3.1: Summary of the prior studies which analyze user reviews (ordered by the publication year)

Study	Approach name	**Venue-Year**
Iacob and Harrison (2013); Iacob et al. (2013a)	MARA	MSR-2013, MobiCASE-2013
Khalid et al. (2015); Khalid (2013)	-	ICSE-2013, IEEE Soft.-2015
Chen et al. (2014)	AR-Miner	ICSE-2014
Maalej and Nabil (2015)	-	RE-2015
Panichella et al. (2015, 2016)	ARdoc	ICSME-2015, FSE-2016
Palomba et al. (2015, 2018)	CRISTAL	ICSME-2015, JSS-2018
McIlroy et al. (2016b)	-	EMSE-20016
Sorbo et al. (2017, 2016)	SURF	FSE-2016, ICSE-2017
Villarroel et al. (2016) & Scalabrino et al. (2017)	CLAP	ICSE-2016, TSE-2017
Palomba et al. (2017)	CHANGEADVISOR	ICSE-2017
Gao et al. (2018)	IDEA	ICSE-2018

source code elements.

Gao et al. (2018) proposed IDEA (IDentify Emerging App issues) that identifies the emerging topics in every update of an app. IDEA automatically identifies the representative sentence for each topic and displays the topic's evolution over time. The main difference between AR-Miner and IDEA is that IDEA introduced the AOLDA (Adaptively Online Latent Dirichlet Allocation) technique to adaptively detect the topics of an update U_i based on the topics of the previous updates. Gao et al. evaluated IDEA by comparing the identified topics to the release notes of six apps. Gao et al. observed that IDEA detects the topics with 60% precision and 60% recall.

Prior research incorrectly assumes that reviews are static in nature and users never update their reviews. In addition, prior research mainly focused on the app-level analysis of reviews. Such app-level analysis does not identify how users perceive every update. In this thesis, we examine changes in the posted reviews to understand how users perceive every update and to learn from users' feedback about every update.

3.3 Characteristics of Successful Apps

Researchers studied the characteristics of successful apps (i.e., apps with high ratings).

Harman et al. (2012) studied the relationship between the app rating, the rank for the number of users who downloaded the app, and the price of the app using data from 32,108 apps on the Blackberry App Store. Their study showed that there is a strong correlation between the app rating and the app downloads and that there is no correlation between the app price, and the app rating or the app downloads.

Prior work primarily examined the characteristics (e.g., deployed APK file size and app category) of successful apps based on the latest update and the overall rating of an app (Vásquez et al., 2013; McIlroy et al., 2016a; Tian et al., 2015; Noei et al., 2017). Ruiz et al. (2016) observed that the overall rating of an app is not impacted much by individual updates. Hence, by taking an app-centric view when studying mobile apps, important information about updates may be lost. Therefore, in this thesis, we focus on studying mobile apps using an update-centric view.

Later, Martin (2016); Martin et al. (2016) performed causal impact analysis to study the impact of the deployed updates of an app on the success of the app. The success of an app is quantified using three metrics: (1) the average app rating, (2) the number

of ratings of the app, and (3) the number of weekly ratings of the app. Martin et al. found that 33% of the studied updates have an impact on the success of their apps. In addition, Martin et al. observed that updates that positively impact the success of an app have more descriptive release notes which mention bug fixes and new features. Finally, Martin et al. reported that 39 out of 45 surveyed app developers wish to know the characteristics of their impactful updates.

Prior research analyzed the characteristics of successful apps by taking an app-centric view. Recent research showed the importance of understanding apps on update-level. This motivates us to study how the reviewing mechanism can help spot good and bad updates. In addition, we carefully examined the characteristics of bad updates and how developers recovered from such updates.

3.4 User-Developer Dialogue in App Stores

Researchers studied the characteristics of user-developer dialogues. In this section, we present prior research that is related to analyzing user-developer dialogue in app stores.

Oh et al. (2013) surveyed 100 users to study the different approaches for user-developer interactions. Oh et al. found that users prefer posting reviews on the mobile stores over giving feedback through other communication channels (such as email or telephone).

McIlroy et al. (2017) studied the impact of a developer response on the rating given by a user. They found that users often increase the given rating after a developer responds to the posted review. McIlroy et al. showed that the majority of developer responses provide assistance to the user or ask the user to contact the developer through another communication channel.

Prior research showed that responding to a review often has a positive effect on the rating that is given by the user to an app. In this thesis, we revisited prior work by conducting a more in-depth study on a larger dataset (as well tracking such user-developer dialogues in detail over an extended period of time).

CHAPTER 4

Studying the Common Developer Mistakes that Lead to Emergency Updates

THE mobile app market continues to grow at a tremendous rate. The market provides a convenient and efficient updating mechanism for updating apps. App developers continuously leverage such mechanism to update their apps at a rapid pace. The mechanism is ideal for publishing emergency updates (i.e., updates that are published soon after the previous update). In this chapter, we study such emergency updates in the Google Play Store. Examining more than 44,000 updates of over 10,000 mobile apps in the Google Play Store, we identify 1,000 emergency updates. By studying the characteristics of such emergency updates, we find that the emergency updates often have a long lifetime (i.e., they are rarely followed by another emergency update). Updates preceding emergency updates often receive

a higher ratio of negative reviews than the emergency updates. However, the release notes of emergency updates rarely indicate the rationale for such updates. Hence, we manually investigate the binary changes of several of these emergency updates. We find eight patterns of emergency updates. We categorize these eight patterns along two categories "Updates due to deployment issues" and "Updates due to source code changes". We find that these identified patterns of emergency updates are often associated with simple mistakes, such as using a wrong resource folder (e.g., images or sounds) for an app. We manually examine each pattern and document its causes and impact on the user experience. App developers should carefully avoid these patterns in order to improve the user experience. In addition, store owners can develop mechanisms to prevent app developers from repeating such mistakes. Our findings are based on a study of the top 1,000 emergency updates. The characteristics and patterns of the emergency updates that are not in the top 1,000 may be different from our findings. Our work is the first step towards analyzing emergency updates and our study may be improved by examining more emergency updates (e.g., analyzing randomly selected updates instead of the top 1,000 emergency updates).

4.1 Introduction

The mobile app market is continuously growing and evolving. Research on mobile app markets reports that in 2018 more than 7.1 million mobile apps are available for users across the different mobile app stores (Statista, 2018). There exist more than 1.2 billion mobile app users worldwide and the number of users continues to grow at a very fast pace (mobiThinking, 2013). ABI research estimates that mobile users downloaded 70 billion apps in 2013 (ABI Research, 2013).

Mobile app stores, such as the Google Play Store, provide a unique updating mechanism to facilitate the release and deployment of app updates. When a developer publishes an update for their app, all the current users of the app can automatically receive the update within the same day (Google, 2018a). Developers extensively leverage this low cost updating mechanism in order to rapidly publish updates. Such an updating mechanism eases the shift towards faster update cycles (Khomh et al., 2012; Mäntylä et al., 2013). However, frequent updates may disturb users, such that some corporations (like Microsoft (Mic, 2018)) opted to reduce the frequency of their updates based on user feedback.

The updating mechanism enables the rapid release and deployment of emergency updates for mobile apps. Emergency updates are updates that are released soon after the previous update. For example, the *"OPM Alert"*[1] app has an update on February 4^{th} 2014 and an emergency update on the following day (February 5^{th} 2014).

However, to the best of our knowledge, there exist no studies that explore such emergency updates. In this chapter, we perform an empirical study on the emergency updates for the top apps in the Google Play Store. Our study focuses on the top 12,000 free-to-download apps (according to the App Annie's top popular apps report in 2013 (AppAnnie, 2018), these top free-to-download apps are distributed among 25 different categories). We choose to study these apps, since updates to these top apps impact a large number of users. Moreover, such mature apps are less likely to exhibit very frequent updates. Hence, we can easily identify emergency updates. We rank the emergency nature of each update by measuring the ratio of the lifetime of its preceding update versus the median lifetime of an update for that particular app (we call this metric, the emergency ratio of an update).

[1] https://play.google.com/store/apps/details?id=gov.opm.status

We study the top 1,000 emergency updates (according to our aforementioned emergency ratio metric). Our study of the characteristics of the top 1,000 emergency updates, shows that emergency updates often have a long lifetime (i.e., they are rarely followed by another emergency update). The lifetime of an emergency update is on average 2.25 times longer than the median lifetime of all non-emergency updates of an app. Hence, users should install these emergency updates and not worry about another emergency update following the emergency update. In addition, we find that developers rarely mention the reasons of emergency updates in their release notes. 63.4% of the emergency updates do not include any useful information about the rationale for the updates in their release notes. We find that the ratio of negative reviews for the update preceding an emergency update is often higher than the ratio of negative reviews for the emergency update.

To further understand emergency updates, we manually examine the decompiled code and files in the binaries (APK files) associated with such emergency updates and their preceding updates. Our manual analysis of 361 emergency updates leads us to identify several common patterns for emergency updates. We document eight patterns of emergency updates. These patterns of common developer mistakes that lead to emergency updates belong to two categories: 1) Updates due to deployment issues and 2) Updates due to source code changes. We document the details of each pattern with its root-causes, example updates, speed of repair, examples of user complaints and lessons learned from the pattern.

The contributions of this chapter are as follows:

1. This chapter is the first study to empirically study the characteristics and patterns of emergency updates for mobile apps.

2. Our detailed documentation of emergency updates can help mobile app developers avoid these patterns before releasing updates for their mobile apps.

Our work is a first step towards documenting such patterns and we expect that future studies will extend these patterns and uncover new ones.

The rest of this chapter is organized as follows. Section 4.2 describes the studied apps and illustrates our data collection process and study methodology. Section 4.3 discusses the characteristics of emergency updates and Section 4.4 defines our approach for identifying patterns of emergency updates. Section 4.5 describes the identified patterns for emergency updates. Section 4.6 outlines the limitations and threats to the validity of our study. Section 4.7 describes the related work. Finally, Section 4.8 concludes our study.

4.2 Methodology

In this section, we describe the methodology of our study. First, we collect apps from the Google Play Store. Then, we identify emergency updates from the collected apps. Figure 4.1 illustrates an overview of the methodology of our study.

4.2.1 Collecting the Studied Apps

We study the top free-to-download mobile apps in the Google Play Store. Google Play Store is one of the world's largest mobile app stores with millions of apps and billions of downloads (Statista, 2018, 2016). We study free-to-download apps because the majority of the apps in the Google Play Store are free-to-download (AppBrain, 2018). In

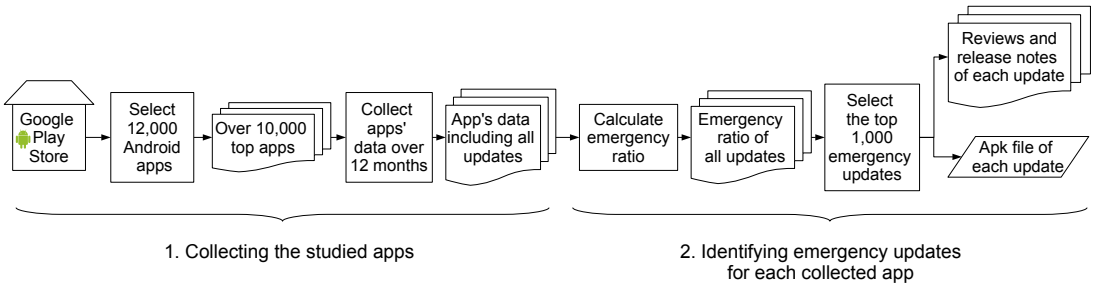

Figure 4.1: An overview of the methodology of our study.

addition, we can only download binary files (APK file) from such apps due to our limited budget. We focus on the most popular free-to-download apps since these apps have many users and contain more updates than the less popular apps. We select the top free-to-download mobile apps using the App Annie's top popular apps report in 2013 (AppAnnie, 2018). App Annie's report provides a list of the 400 top apps for each of the 24 non-Game categories. In addition, App Annie's report provides a list of 400 top apps for each of the 6 Game categories. In total, we collect 12,000 top free-to-download apps. We use the App Annie's published popular apps ranking in 2013 (one year before we start our data collection) in order to ensure that the apps are more stable and that the collected updates are not early updates of an app (such early updates are likely to be rapid in nature).

In order to collect daily data about the studied apps, we crawl the Google Play Store using a specialized store crawling library (Akdeniz, 2013). We collect the general information of an app on a daily basis, such as its name, category, uploaded APK file, and user reviews. We select the Samsung S3 as the mobile device to download apps from the Google Play Store, as Samsung S3 is a very popular mobile device (at the time of our study). If the crawler interacts with the Google Play Store frequently, the crawler may

be blocked by the Store due to too many requests. Therefore, we use a timer to pause the crawler periodically and visit the store page of a particular app only once a day.

The crawler runs during the study period of around 12 months starting from November 26^{th} 2013 to November 18^{th} 2014. During the study period, some apps were removed from the Google Play Store. Hence, we only collect data for the 10,747 available top apps. During the study period, we collect the APK files and user reviews for 44,113 updates. On average, each app has 4.11 updates during the study period and 6,894 apps published at least one update during the study period.

4.2.2 Identifying Emergency Updates

In order to identify emergency updates, for each update U_i, we calculate the *lifetime of the update (U_i)*, as the time difference (in days) between the update date of the update U_i and the update date of the next update U_{i+1}.

In order to identify emergency updates, intuitively we consider an update U_i as an emergency update, if the lifetime of the update U_{i-1} is less than one day. However, some apps have a more frequent update cycle than other apps. For example, The *"Hola Free VPN"*[2] app has 69 updates during the study period (the median update lifetime for this app is three days), while the *"Stock Watch: BSE / NSE"*[3] app has only three updates during the study period (the median update lifetime for this app is 202 days). There are 200 apps that have a median update lifetime of less than a week.

Therefore, we cannot use one value as a threshold to determine whether an update is an emergency update. Instead, we quantify the emergency nature of an update using a metric named the *Emergency ratio* of an update. We define the *Emergency ratio* of an

[2] https://play.google.com/store/apps/details?id=org.hola
[3] https://play.google.com/store/apps/details?id=com.snapwork.finance

Table 4.1: Mean and five-number summary of emergency ratio of the top 1,000 emergency updates. The larger the emergency ratio, the longer the lifetime of the update preceding the emergency update.

Update category	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
The emergency update (U_i)	0.03	0.00	0.02	0.03	0.04	0.05

update U_i as the ratio between the lifetime of the U_{i-1} update and the median lifetime of all the updates for that app.

$$Emergency\ ratio(U_i) = \frac{lifetime(U_{i-1})}{Median\ lifetime\ of\ the\ updates\ for\ that\ app} \tag{4.1}$$

The lower the emergency ratio, the higher the emergency nature of a particular update. We rank the updates by their emergency ratio and we focus on the top 1,000 emergency updates with the lowest emergency ratios. Table 4.1 represents the mean and five-number summary of emergency ratio of the top 1,000 emergency updates. For example, the *"Tweakker APN INTERNET MMS"*[4] app has a median update lifetime of 124 days and the lifetime of one update (*"version 1.8.1"*) is only one day (i.e., the emergency ratio for the following update is 0.008). Such a low emergency ratio indicates that the following update is very likely an emergency update.

Table 4.2 represents the mean and five-number summary for lifetime (in days) of the updates that precede the 1,000 top most updates according to their emergency ratio. We notice that 688 of the updates are followed by an emergency update within one day and a large portion of updates are followed with an emergency update within two days.

[4] `http://tweakker.com`, the app was available during the study period but the Google Play Store no longer hosts this app at the time of the writing of this chapter.

Table 4.2: Mean and five-number summary for lifetime (in days) of the top 1,000 updates (according to the emergency ratio of their following update).

Update category	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
The lifetime of the update preceding the emergency update (U_{i-1})	1.79	1.00	1.00	1.00	2.00	21.00

4.3 Characteristics of Emergency Updates

In this section, we study the characteristics of the top 1,000 emergency updates. In particular, we focus on five aspects of emergency updates: their lifetime, the content of their release notes, their rating and their version numbering. Table 4.3 summarizes the major findings and implications of our findings.

4.3.1 Lifetime of Emergency Updates

We would like to examine whether emergency updates last for a long time, and whether emergency updates are likely to be followed by additional emergency updates. First, we compare the lifetime of emergency updates and the median lifetime of an update of an app. We then study whether there is a statistically significant difference between the emergency ratios of the update that follows an emergency update and the update that follows a non-emergency update. We use the *MannWhitney U test* (Wilcoxon rank-sum test) (Gehan, 1965), since the emergency ratio is highly skewed. The *MannWhitney U test* is a non-parametric test which does not have any assumptions about the distribution of the sample population. A p-value of ≤ 0.05 means that the difference between the emergency ratios of the updates that follow an emergency update and the emergency ratios of the updates that follow a non-emergency update is statistically significant and we may reject the null hypothesis. By rejecting the null hypothesis, we

Table 4.3: Our major findings (and their implications) on the characteristics of the top 1,000 emergency updates for top Android apps.

Lifetime of emergency updates	**Implications**
Emergency updates have a long lifetime.	Users should update mobile apps with the emergency updates without being concerned about another update showing up soon afterwards.
Release notes of emergency updates	**Implications**
Release notes of emergency updates rarely provide a clear description about the rationale for the update.	Developers should highlight the emergency nature of an update and encourage users to download it.
Version numbering of an emergency update	**Implications**
There is no fixed numbering convention for the emergency updates.	Developers may consider using version numbers to indicate whether an update is a major update or an emergency update.
Ratings of emergency updates	**Implications**
The updates preceding the emergency updates have more negative reviews than the emergency updates.	It would be beneficial if users are told about the recent rating of a newly available update (relative to the apps rating for its prior update) when users are informed of the availability of the update.

can accept the alternative hypothesis, which shows that there is a statistically significant difference between the emergency ratios of the updates that follow an emergency update and the emergency ratios of the updates that follow a non-emergency update.

Emergency updates have a long lifetime. We find that that on average the lifetime of an emergency update is 2.4 times the median lifetime of all non-emergency updates of an app. The p-value of the *MannWhitney U test* Test is 0.01271, (i.e., less than 0.05) which shows that there is a statistically significant difference between the emergency ratios of the updates that follow an emergency update and the emergency ratios of the

updates that follow a non-emergency update.

Nevertheless, we find 40 emergency updates (out of the 1,000 studied emergency updates) that are followed by another emergency update. We find that the minimum value of the emergency ratio of the update after the emergency updates is only 0.01. For example, *"Horoscope HD Free"*[5] app has an update on March 30^{th} 2014, the development team published an emergency update on the next day (March 31^{st} 2014). However, the development team forgot to add the needed permissions for the modified code. Therefore, they publish yet another emergency update on the next day on April 1^{st} 2014 to add the missing permissions.

Our results suggest that users should update their mobile apps if an emergency update is published as developers rarely follow an emergency update with yet another one.

4.3.2 Release Notes of Emergency Updates

We would like to better understand the rationale for emergency updates and whether developers do inform their users about the rationale for such emergency updates. We read the release notes for all 1,000 identified emergency updates (U_i) and the updates preceding the emergency updates (U_{i-1}). Then we identify the differences in the release notes between an emergency update (U_i) and the preceding update (U_{i-1}). Based on the differences between the two release notes, we determine whether the release note provides useful information about the rationale for the emergency update.

Around a third of the updates explain the rationale for the emergency update in the release notes. We find that 36.6% of the release notes explain what is fixed in

[5] https://play.google.com/store/apps/details?id=ch.smalltech.horoscope.free

the emergency update. For example, the *"Body Fat Calculator"*[6] app has an update on October 10^{th} 2014 and an emergency update on the next day. The emergency update added text in the release notes that describes the update as follows: *"Version 3.2.1: Revised on-line guide did not load properly on some old devices. Issue Fixed"*.

Two third of the updates do not have a clear description regarding the rationale for the emergency update. We find that 63.4% of the release notes do not include any useful information about the rationale for the emergency update. 59.8% of the emergency updates use the same release note as the previous update and 3.7% of the emergency updates changed their release notes, however with very general descriptions, such as *"Fix bugs and add new features"*.

We find that both apps with long or short update cycles rarely include the rationale for the emergency update in their release notes. For example, the *"Emojidom: Chat Smileys and Emoji"*[7] app has a median update lifetime of 20 days. An update of the *"Emojidom: Chat Smileys and Emoji"* app was published on September 16^{th} 2014 and an emergency update was published on the next day. Both updates have the exact same release notes. Similarly, the *"Sushi Bar"*[8] app with a long update lifetime of 409 days has one emergency update without updating the release notes that are associated with that emergency update.

We also investigate the release notes that are from the websites of the 361 studied apps with emergency updates. We find that 279 apps provide a link to their website on their store page in the Google Play Store. We manually checked the websites for these 279 apps and we find that only 18 apps provide release notes on their web site. We find that such online release notes provide no additional information over the release notes

[6] https://play.google.com/store/apps/details?id=com.fullquieting.android.FatCalc
[7] https://play.google.com/store/apps/details?id=com.plantpurple.emojidom
[8] https://play.google.com/store/apps/details?id=com.roidgame.sushichain.activity

that are posted on the app's store page.

Our analysis suggests that developers should consider highlighting emergency updates in their release notes to encourage users to download these emergency updates. Besides, store owners should encourage app developers to highlight the fixed issues in their release notes. For example, store owners could leverage the changed files (e.g., changed image files or added permissions) in the emergency updates and provide initial suggestions for the release notes. For example, if an emergency update added an app permission, the store owner can give an initial recommendation to app developers to write a fix about the added permission.

4.3.3 Version Numbering of an Emergency Update

We would like to understand whether the update version numbers of emergency updates follow a certain format, in order to ease the further investigation of emergency updates. We study the difference between the version numbers for the emergency update (U_i) and the update preceding the emergency update (U_{i-1}).

There is no fixed numbering convention for emergency updates. We define *Version level* as the number of digits that are separated by dots (".") in a version number. In some cases, developers change the version numbers while keeping the same number of version levels (e.g., from version 2.32 to version 2.33). In other cases, developers use an additional version level for their emergency updates relative to the version number of the preceding update (e.g., from version 2.3 to version 2.3.1). Table 4.4 summarizes the different changes to version numbers that we find in the emergency updates.

From Table 4.4, we summarize the results as follows:

- In 11% of the emergency updates, the update preceding the emergency update

Table 4.4: Study of version numbers in the top 1,000 emergency updates.

Changes in the version numbers of emergency updates	% of emergency updates
The update preceding the emergency update and the emergency update have the same exact version number.	11%
The emergency update has the same version level as the update preceding the emergency update (e.g., version 5.77 and 5.78).	71%
The emergency update has an additional version level than the update preceding the emergency update (e.g., version 7.3 and 7.3.1).	18%

and the emergency update both have the **same exact version number**.

- In 89% of the emergency updates, the update preceding the emergency update and emergency update have **different version numbers**. In such cases:

- 71% of the emergency updates have the **same version level** as the update preceding the emergency update (e.g., version 5.77 and 5.78).
- 18% of the emergency updates have an **additional version level** than the update preceding the emergency update (e.g., version 7.3 and 7.3.1).

Based on the obtained results, we find that users cannot know whether an update is an emergency update just based on changes to version numbers. However, developers may consider making better use of version numbering patterns (and best practices) to indicate whether an update is a major update or an emergency update.

4.3.4 Ratings of Emergency Updates

In order to study changes in the ratings to the updates preceding the emergency updates. We focus on negative reviews (reviews with one or two stars rating) as these reviews mainly contain user complaints (Khalid et al., 2015; Martin, 2015). Among the total 1,000 emergency updates, 363 updates have no reviews for the update preceding the emergency update and 280 updates have too few reviews (with a median of two received reviews for the update preceding the emergency update). Therefore, we focus on the rest 357 updates to study the negative reviews.

We define the ratio of negative reviews (RNR) for an app A at date d as follows:

$$RNR(A,\ d) = \frac{N(A,\ d)}{T(A,\ d)} \tag{4.2}$$

where $N(A,\ d)$ is the number of negative reviews that are received at date d for an app A, and $T(A,\ d)$ is the total number of reviews that are received at date d for an app A.

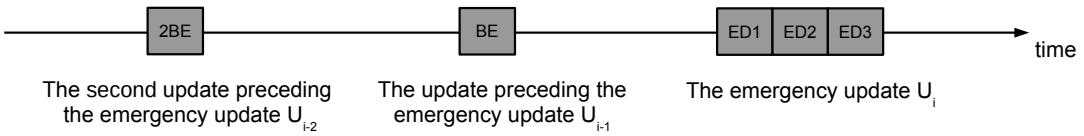

Figure 4.2: An overview of the studied dates for an app A.

For each app A that has an emergency update, as illustrated in Figure 4.2, we calculate the ratio of negative reviews for the app during the following five dates:

- Date d_{2BE} is the deployment date of the second update preceding the emergency update.
- Date d_{BE} is the deployment date of the update preceding the emergency update.
- Date d_{ED1} is the deployment date of the emergency update.
- Date d_{ED2} is the following day to the deployment date of the emergency update.
- Date d_{ED3} is the second following day to the deployment date of the emergency update.

In order to compare the ratio of negative reviews, we use the $RNR(A,\ d_{BE})$ i.e., the ratio of negative reviews on the deployment date of the update preceding the emergency update as a baseline. We compare the ratios of negative reviews as follows:

$$Comparison_{2BE}(A) = \frac{RNR(A,\ d_{2BE})}{RNR(A,\ d_{BE})} \tag{4.3}$$

$$Comparison_{ED1}(A) = \frac{RNR(A,\ d_{ED1})}{RNR(A,\ d_{BE})} \tag{4.4}$$

$$Comparison_{ED2}(A) = \frac{RNR(A,\, d_{ED2})}{RNR(A,\, d_{BE})} \tag{4.5}$$

$$Comparison_{ED3}(A) = \frac{RNR(A,\, d_{ED3})}{RNR(A,\, d_{BE})} \tag{4.6}$$

where $Comparison_{2BE}(A)$ compares the ratio of negative reviews between dates d_{2BE} and d_{BE}. The rest of the equations follow similar comparisons.

The updates preceding the emergency updates have more negative reviews than the emergency updates. We compare the ratio of negative reviews in different days as illustrated in equations (4.3, 4.4, 4.5 and 4.6) for all emergency updates.

Table 4.5: Mean and five-number summary of comparison metrics for the top 1,000 emergency updates. A value higher than one means that the ratio of negative reviews in the corresponding date is higher than the date of the deployment of the update preceding the emergency update.

Metric name	Mean	Min.	1st Qu.	Median	3rd Qu.	Max
$Comparison_{2BE}$	0.89	0.00	0.00	0.56	1.29	8.83
$Comparison_{ED1}$	1.02	0.00	0.00	0.78	1.30	7.48
$Comparison_{ED2}$	0.91	0.00	0.00	0.67	1.22	10.74
$Comparison_{ED3}$	0.97	0.00	0.00	0.67	1.31	8.48

As shown in Table 4.5, the update preceding the emergency update (U_{i-1}) has a higher ratio of negative reviews than the 2^{nd} update preceding the emergency update (U_{i-2}). Also we find that the update preceding the emergency update (U_{i-1}) has a higher ratio of negative reviews than the emergency update (U_i) in the 1^{st}, 2^{nd}, and 3^{rd} deployment days. We notice that users still complain during the days after issuing the emergency update since users may have not downloaded the new update (the emergency

update) yet. Interestingly, for some emergency updates, users may prefer the issues not being fixed, such as the emergency updates that are published to ensure that the app is able to continue displaying advertisement (based on our manual reading of user reviews).

Based on the observed results, it would be beneficial if users are told about the recent rating of a newly available update (relative to the rating of the prior update of the app) when users are informed of the availability of the update. This would permit users to decide on whether they wish to update their apps or not. For example, users might even configure their automated updaters to only install updates with an improved rating.

4.4 Our Approach for Identifying the Patterns of Emergency Updates

In this section, we present our approach for identifying the patterns of emergency updates. In order to identify the patterns of emergency updates, as illustrated in Figure 4.4, we leverage four data sources as follows:

1. The APK file for each update.
2. The release notes for each update.
3. The reviews associated with each update.
4. The F-Droid apps repositories data (F-Droid, 2018).

We explain the content of each source in the rest of this section.

4.4.1 The APK File for Each Update

First we unarchive the APK files of the emergency update (U_i), the two updates preceding the emergency update (U_{i-2} and U_{i-1}) and the update following the emergency update (U_{i+1}). We unarchive each APK file using the Android-Apktool (Apktool, 2018). Each unarchived APK file contains four folders and one AndroidManifest.xml file as follows (Apk, 2013):

- **Smali:** This folder contains the byte code files (Apk, 2013; Hendysoft, 2013). In order to obtain readable source code, we converted the APK to jar using dex2jar tool (dex2jar, 2016). We decompiled the generated jar into java source code using the Class File Reader (CFR) tool (CFR, 2018).

- **Libs:** This folder contains third-party libraries that are used by the mobile app (Apk, 2013).

- **Res:** This folder stores the different resources that are needed by the app, such as images (stored in "drawable" folders), layout, style, colors, and configuration files for the various customizations that are used across the app (e.g., the displayed text) (Apk, 2013).

- **Assets:** This folder contains raw files that a developer may want to use in the app, such as texture, audio, text, fonts, and game data files. The raw data files can be stored either in the sub folder raw in the Res folder (res/raw) or in the assets folder: Files in the raw folder need to be accessed via the resource identifier, while files in the assets folder are accessed using the Java filesystem API without constraints on the file names (Apk, 2013; Pete Houston, 2011; stackoverflow, 2011, 2013; Google, 2018d).

- **AndroidManifest.xml:** The manifest file specifies the required configuration for the Android platform for the proper execution of the app (Google, 2018a). For example, the manifest file specifies the required Android SDK version for the mobile app to work properly (Google, 2018c). Developers also configure the needed permissions (Google, 2018e) and the needed software or hardware features (such as camera, Bluetooth, app widgets) (Google, 2018b) that are required by the app.

In total, we only encounter 56 APKs where the unarchiving or decompilation failed in our experiment.

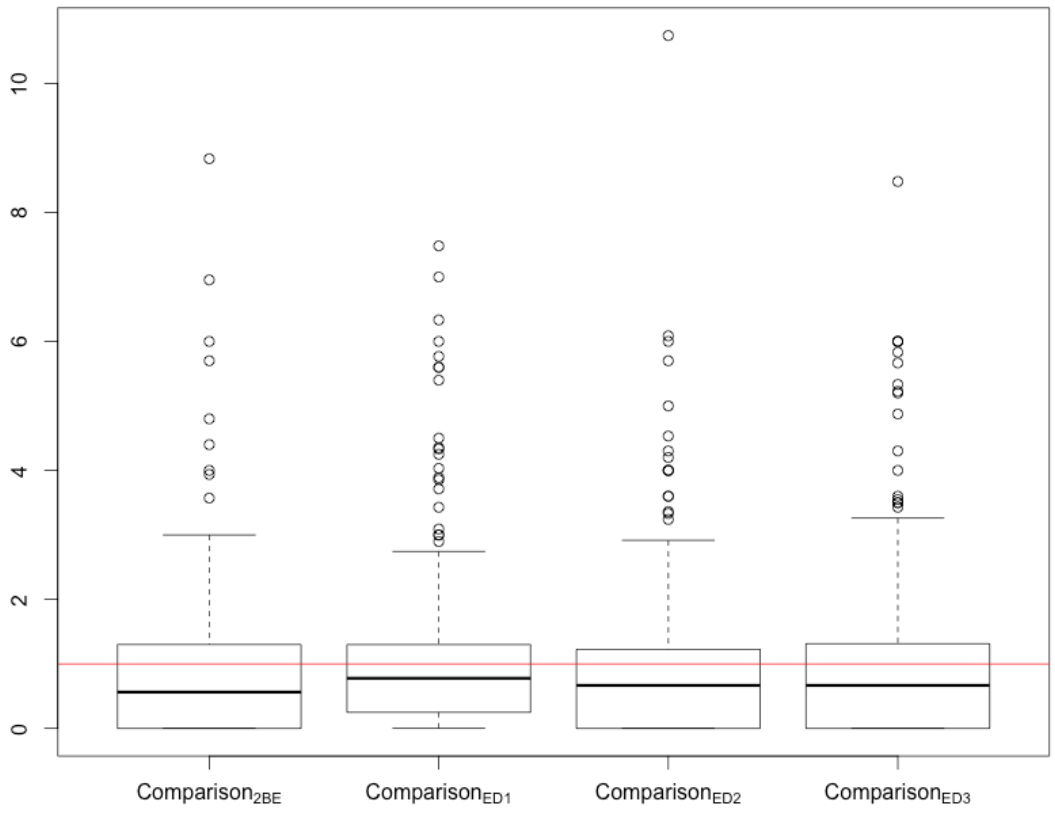

Figure 4.3: Boxplot for the calculated comparison metrics for the top 1,000 emergency updates. The red line in the figure shows the metric value 1. Metric values lower than 1 means that the update preceding the emergency update has a higher ratio of negative reviews than this emergency update.

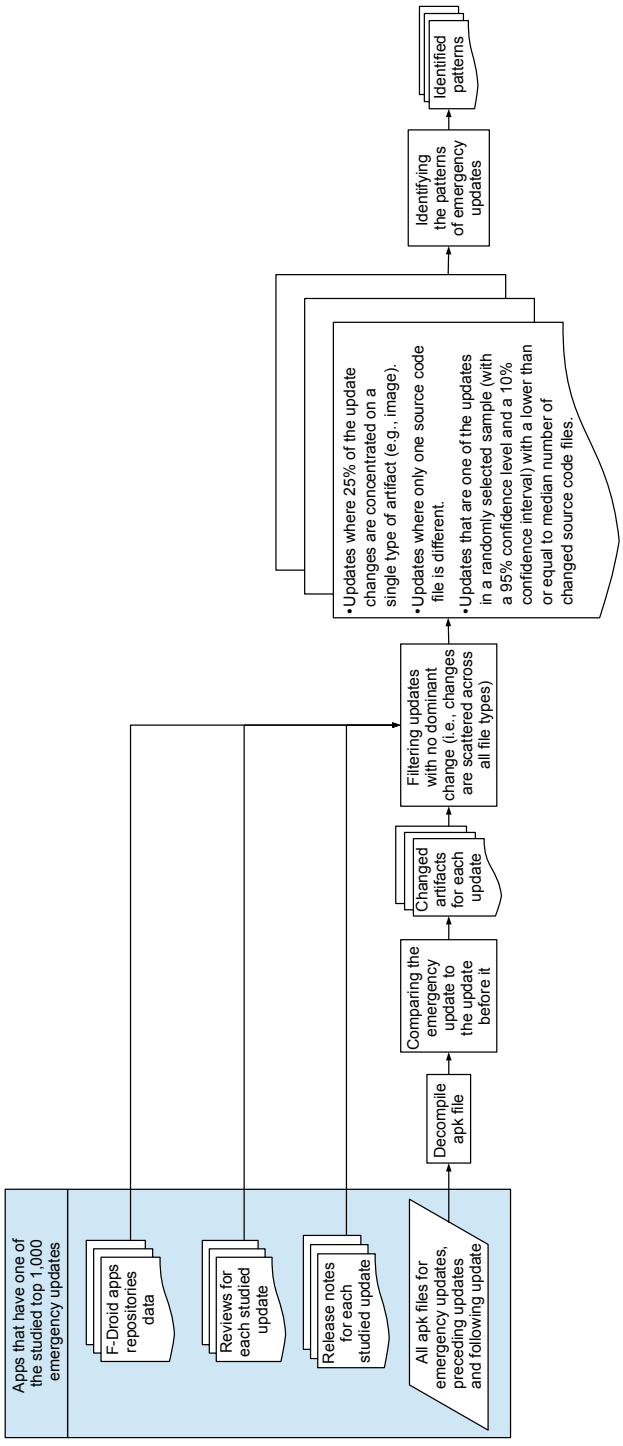

Figure 4.4: Process for identifying patterns of emergency updates.

We compare the decompiled binaries in the emergency update (U_i) and the preceding update (U_{i-1}) to find what is changed in the emergency updates. We define ten types of files: code, third-party libraries, images, layout, style, colors, audio and video, displayed text, application configurations, and changes in the AndroidManifest.xml file. Then we categorize the similarly changed files together into one type, e.g., the images files are grouped together into image type, all source code files are grouped into the code type and changed XML elements in AndroidManifest.xml file as the AndroidManifest.xml change type. After that, we count the number of changes in each type. For the artifacts that are typically at the file level, e.g., figures and source-code files, we count the number of files that are changed. For the artifacts that are not at the file level, e.g., permissions and SDK version, we count the number of individual items of the artifact. In particular, changes to the AndroidManifest.xml are not counted as only one change. Then we calculate the total number of changes in the APK file as the total number of items changed in the AndroidManifest.xml file and the total number of changed files (other than the AndroidManifest.xml file). Finally, we calculate the percentage of changes in each type relative to the total number of changes in the APK file.

We examine the decompiled source code files that are different between the emergency update (U_i) and the preceding update (U_{i-1}). We find that the median percentage of files that are the same between the two updates is 99.7%, and the minimum percentage is 1.2%. We manually investigated the updates with a very large number of source code files that are different to the files in the proceeding updates (i.e., low percentage of unchanged files). In particular, we rank the updates with the percentage of unchanged files across the two updates. We find that the updates in the first quartile have up to

98.3% of the files that are the same between the update and the previous one. We manually investigate several of the updates in the first quartile. We find that there are two reasons for the low percentage of unchanged code files across updates:

1. The low percentage of unchanged source code files is due to code obfuscation, instead of actual code changes.

2. The app has too few app-specific code files and the rest of the code files are library files (e.g., advertisement library files). Hence, the adoption of a different library would lead to a very large percentage of changed source code files in that particular update. For instance, the update of the *"Fingerprint Lock Free"*[9] app on May 3^{rd} 2014 contains only 1.2% of the code files that are unchanged relative to its prior update. This update involves changes to four out of nine app-specific source code files and 651 out of 654 third party advertisement libraries files.

Since around two thirds of the emergency updates do not specify the rationale for their update (see Section 4.3.2), we need to manually analyze some of these emergency updates. In order to ease our understanding of the rationale for an emergency update, we manually examine an emergency update if one of the following three selection criteria hold:

- At least 25% of the update changes are concentrated on a single type of artifact. For example, all of the files (100%) in the images type is changed in the *"Pediatrics"*[10] app on January 8^{th} 2014. Therefore, we manually examine this emergency update. If there are no major changes to the artifacts, it would be very difficult for us to deduce the root-cause of the emergency update.

[9] https://play.google.com/store/apps/details?id=com.nb.fingerprint.lock.free
[10] https://play.google.com/store/apps/details?id=com.texterity.android.Pediatrics

- There is only one code file that differs (i.e., a new file, a removed old file or a changed old file) between the emergency update (U_i) and its preceding update (U_{i-1}), as any large amounts of source code differences between the two updates are likely due to code obfuscation. Furthermore, it is very challenging for us to deduce the rationale for such large changes given that we are using decompiled code and we have very limited knowledge of the apps and their codebase.

- The update is one of the updates in a random sample of updates (with a 95% confidence level and a 10% confidence interval) that have lower than or equal to median number of changed source code files. In order to create the random sample, we count the median number of changed source code files in the emergency updates and find that the median number of changed source code files is four. We collect all 376 updates that have less than or equal to four changed source code files and randomly select a sample of 77 updates for manual inspection. The size of our random sample achieves a 95% confidence level and a 10% confidence interval.

In our study, we examine the updates for which we can identify the rationale for the changes without the need for the release notes. E.g., 1) updates where at least 25% of the update changes are concentrated on a single type of artifact (such as images, requested permissions, layout, colors, Android SDK versions), 2) updates with only one source code file being different, or 3) the random sample of updates for manual inspection.

In total, we select 361 emergency updates to manually examine based on our aforementioned selection criteria.

4.4.2 The Reviews Associated with Each Update

Google Play Store enables users to provide their reviews for each update of an app. We manually examine the reviews that are associated with each studied emergency update (U_i) and the preceding update (U_{i-1}).

4.4.3 The Release Notes for Each Update

Release notes are also one of the data sources that are used to study the emergency updates. However, about 60% of the emergency updates use the same release notes as their preceding update, and 3.7% of the emergency updates only have general words for the emergency updates, such as: *Hot-fix and Various Bug Fixes* (see Section 4.3.2).

In our manual process for identifying patterns of an emergency update, first we start by reading the release notes for the emergency update and comparing the details that are mentioned in the release notes to the changed files in the emergency update. If the release notes do not mention any useful details about the rationale for the emergency update, then we manually read all the reviews associated with the emergency update (U_i) and the preceding update (U_{i-1}). We excluded reviews that contain generic user complaints and considered reviews that mention details that are related to each identified pattern.

If the emergency update does not include useful release notes or reviews that are explicitly related to the identified patterns, we manually compare the changed files in the four updates (the emergency update (U_i), the update following emergency update (U_{i+1}), and the two updates preceding the emergency update (U_{i-1}) and (U_{i-2})) in order to identify the rationale for the change. For example, in order to identify that incorrect images were included in a particular update (U_i), we compare the images content

in the two preceding updates (U_{i-1}) and (U_{i-2}) in order to see the content before the emergency update, then we compare the emergency update with the preceding update (U_{i-1}) in order to identify the changed images. Finally, we compare the content of the updates (U_i) and (U_{i+1}) in order to ensure that the updated files are the correct images that are still used in the following updates.

4.4.4 The F-Droid Apps Repositories Data

The release notes of app updates may be short and with limited details (see Section 4.3.2). In order to have more information about the update, we explore another source of data to understand the patterns of emergency updates. In particular, we find that 11 apps with emergency updates are hosted in a software repository named F-Droid (2018). F-Droid provides a public collection of different FOSS (Free and Open Source Software) apps. We collect all the available releases for the 11 apps from F-Droid. We search for the emergency update (U_i) and the preceding update (U_{i-1}) releases. Then we collect the code comment in the source code files and the commit messages, for the changed files between the two releases for the updates (U_i) and (U_{i-1}). We collect the repository release notes, which are different from the release notes shown in the Google Play Store,[11] for the emergency update (U_i).

The releases of emergency update (U_i) or the release of the preceding update (U_{i-1}) are not always explicitly tagged in the repository. In such cases, we use the update time to find the related code commits for the emergency update (U_i) and the preceding update (U_{i-1}). Then we use the commit messages and code comments that are associated with these particular code commits to understand the root-cause for the emergency

[11] We refer to the release note collected from the Google Play Store as "release notes" and the release notes collected from the apps repositories as "repository release notes".

updates.

We have the following three cases for the 11 apps that are hosted on F-Droid:

1. The repository is not available. We find two updates where either the repository URL is not working or the repository does not store the commits for the code changes during our study period.

2. The repository is available but we cannot identify the code changes to the emergency update. We find three updates in this case.

3. The repository is available and we could benefit from the releases and code changes in order to map the studied updates to the actual changed code. We find six updates in this case.

Such results show that even though F-Droid provides much detailed development information about mobile apps, data analytics on mobile apps cannot rely solely on F-Droid due to its small scale and the quality of its data.

4.4.5 Manual Inspection

We manually inspected all emergency updates that meet our aforementioned selection criteria. In our process, we go through the following two steps:

- **First step – identifying patterns of emergency updates.** Two researchers (including myself and a collaborator) work together by manually examining all differences between all the associated updates. We examine the differences between the downloaded APKs, release notes, F-Droid repositories data and app reviews in order to know the issue that is fixed in each update. If the issue is a

new issue then we add it to the list of identified issues. We iteratively examine all the updates until there are no more identified issues. If there is disagreement during the issue identification process, the two researchers together come to a consensus. In particular, we have 20 updates which there is disagreement in the identified issues. After finalizing the list of identified issues, we consider only issues that are fixed in more than one update as patterns for emergency updates. The output of the first step is the list of the identified patterns along with the updates that are related to each pattern.

- **Second step – identifying root-causes of each pattern.** Similar to the first step, the two researchers examine all the updates associated with each pattern in order to study the root-causes for each pattern. For each inspected emergency update, we use all available data in order to identify the root-cause for the update. If we identify a new root-cause for a certain pattern, we add this root-cause to the list of the root-causes that are related to this pattern. We iteratively examine all the updates in the pattern until there are no more identified root-causes.

4.5 Identified Patterns for Emergency Updates

We could map 146 updates to certain patterns and we cannot identify the patterns for the remaining 215 updates (361 in total). We identify two categories of emergency patterns including “Updates due to deployment issues” and “Updates due to source code changes”. Each category consists of different patterns. Tables 4.6 and 4.7 summarize the identified patterns for emergency updates. For each pattern, we discuss the description of the pattern and the root-cause of the pattern with some real-life examples.

We also discuss how fast developers address the root-cause problem of the pattern (as shown in Table 4.8) and show some examples of user complaints. Finally, we discuss the lessons learned from each pattern.

4.5.1 Updates Due to Deployment Issues

We identify 68 emergency updates that are due to deployment issues: nine updates that are done to address image quality issues, 18 updates that are done to address image content issues, 13 updates that are done to address inconsistent permissions, 18 updates that are done to address inappropriate SDK versions and ten updates that are done to address incorrect debugging mode.

Including low quality images

Pattern description:

An update is published to repair image quality (e.g., image resolution or image brightness). If an emergency update (U_i) fixes the quality of the displayed images in the update preceding the emergency update (U_{i-1}), we consider that the emergency update (U_i) belongs to this pattern.

Root-causes:

We find three root-causes of this pattern:

1. Developers may not have the resources to verify the quality of images on every single device on which their apps may run. The images may not be of a suitable

Table 4.6: Patterns of deployment issues emergency updates.

Updates due to deployment issues				
Pattern Name	**Description**	**Root-cause**	**Identified updates**	**App names**
Including low quality images	An update is published to repair image quality issues (image resolution or image brightness).	1) Developers do not test the quality of images on devices with varying screen resolution. 2) Developers forget to test newly added images.	9	"*Read Unlimitedly! Kids'n Books*","*Chomp SMS theme add-on*", "*Learn Portuguese with Babbel*", "*Learn French with Babbel*", "*Learn German with Babbel*", "*Our Groceries Shopping List*", "*Curso de Ingles Gratis*", "*Naver*" and "*Baby FlashCards for Kids*"
Including incorrect images	An update is published to replace the incorrect image with the appropriate one.	1) Developers of multiple mobile apps use the wrong images from their other apps. 2) Developers miss updating old images.	18	"*Camera ZOOM FX Buddy Pack*","*Pediatrics*", "*Camera ZOOM FX Halloween Pack*", "*Navmii GPS USA (Navfree)*", "*Davis's Drug Guide*", "*10K Runner Trainer FREE*", "*C25K ©- 5K Runner Trainer FREE*", "*OnLive*", "*Camera ZOOM FX Composites*", "*Camera ZOOM FX New Composites*", "*Camera ZOOM FX Extra Props*", "*Camera ZOOM FX Props Pack*", "*Camera ZOOM FX More Composites*", "*Camera ZOOM FX Picture Frames*", "*Camera ZOOM FX Cool Borders*", "*Brain Age Test Free*", "*Police Lights and Sirens Pt.*" and "*Safeway*"
Having inconsistent permissions	An update is published to remove the request for not needed permissions or to request the needed permissions that are missing.	1) Developers use some permissions during development but some of these permissions are no longer needed for the most recent app update. 2) Developers forget to add some of the needed permissions. 3) Developers make mistakes when defining customized permissions.	13	"*Easy Uninstaller App Uninstall*", "*Free Sports Radio*", "*Mobiletag QR & product Scanner*", "*Higher One Mobile Banking App*", "*Mixology ™Drink Recipes*", "*Bingo Fever - Free Bingo Game*", "*Spanish Translator*" '*One More Clock Widget Free*", "*OPM Alert*", "*Evernote Widget*", "*Horoscope HD Free*", "*AroundMe*" and "*AnyTimer Pill Reminder*"
Having inappropriate SDK versions	An update is published to fix the needed SDK version of the app.	1) Developers mistakenly leverage new features from a new version of the SDK without specifying the need of the new SDK version in the most recent app update. 2) Developers make a code change and downgrade or upgrade the target SDK version, which is discovered to lead to problems in the field.	18	"*Guitar Lessons Free*", "*Horoscope and Tarot*", "*Convert video to mp3*", "*Mp3 Converter Free*", "*Chinook Book*", "*Stock Watcher*", "*Voxer Walkie Talkie Messenger*", "*Pyramid Spirits 3 - Slots*" "*Simple Notepad*", "*Useful Knots*", "*DJ*", "*FVD - Free Video Downloader*", "*BoothStache*", "*UglyBooth*", "*MixBooth*", "*AgingBooth*", "*FatBooth*" and "*BaldBooth*"
Incorrect debugging mode	An update is published to enable or to disable the debugging mode.	Developers need to enable or disable the debugging mode.	10	"*Office Calculator Free*", "*Mixology^{TM} Drink Recipes*", "*Offroad Legends*", "*Foods That Burn Fat*", "*Real Time CPR Guide*", "*Prenatal Ultrasound Lite*", "*TAGstagram - IG TAG searcher*" and "*VAT CALCULATOR*"

Table 4.7: Patterns of source code changes emergency updates.

Updates due to source code changes				
Pattern Name	**Description**	**Root-cause**	**Identified updates**	**App names**
Invoking unavailable APIs	An update is published to check the availability of APIs before calling the APIs.	Developers find that their app crashes due to calling certain features in Android APIs that may not exist in some SDK versions on user devices.	5	*"justWink Greeting Cards"*, *"Couple Tracker - Mobile monitor"*, *"DJ Control"*, *"Free Golf GPS APP - FreeCaddie"* and *"Word Learner Vocab Builder GRE"*
Advertisement issues	An update is published to ensure the correct display of advertisements.	1) Developers do not set the correct identifier for the advertisement. 2) Developers do not validate how advertisements are loaded and displayed in the app.	19	*"Tractor Pull"*, *"KWCH 12"*, *"Pocket Tanks"*, *"Baja Trophy Truck Racing"*, *"Make Up Salon!"*, *"Love-Cycles - Period Tracker"*, *"Drift Mania Championship Lite"*, *"Update Samsung Android Version"* and *"Crazy Grandpa"*
Un-handled exceptions	An update is published to handle exceptions.	Developers do not handle all possible exceptions that can occur when users run the app.	54	*"Photosphere Free Wallpaper"*, *"SlenderMan LIVE"*, *"FrostWire - Torrent Downloader"*, *""*, *"GROWLr: Gay Bears Near You"*, *"Fast Burst Camera Lite"*, *"Zen Pinball"*, *"Hersheypark"*, *"Harvest Time & Expense Tracker"*, *"First Aid Emergency & Home"*, *"Flashlight + Clock"*, *"FeaturePoints: Free Gift Cards"*, *"GPS Phone Tracker Pro"*, *"Followers+ for Twitter"*, *"Buycott - Barcode Scanner Vote"*, *"Highway Crash Derby"*, *"Next Music Widget"*, *"GO Cleaner & Task Manager"* and *"Speed Card Free"*

Table 4.8: The median speed of repair for all patterns of emergency updates.

Category	Pattern name	Median speed of repair (days)
Updates due to deployment issues	Including low quality images	1
	Including incorrect images	3
	Having inconsistent permissions	1
	Having inappropriate SDK versions	1
	Incorrect debugging mode	2
Updates due to source code changes	Invoking unavailable APIs	2
	Advertisement issues	1
	Un-handled exceptions	1

quality (e.g., resolution) for all mobile devices (especially for mobile devices with high screen resolution).

2. Developers may forget that they included new images in an app and they forget to test the quality of such new images.

3. Developers add images and icons without knowing whether users would like the new images and icons.

Example updates:

- An update of the "*Chomp SMS theme add-on*"[12] app on August 17^{th} 2014 has icons that are not comfortable for users (very bright). An emergency update adjusts the images to be less fluorescent.

- An update of the "*Baby FlashCards for Kids FREE*"[13] app on November 25^{th} 2013

[12] https://play.google.com/store/apps/details?id=com.p1.chompsms

[13] https://play.google.com/store/apps/details?id=au.com.alexooi.android.flashcards.alphabets

has images with sizes that are not suitable for all devices. An emergency update adjusts the images sizes to be suitable for all devices, especially for the devices with high-resolution screens.

Examples of user complaints:

We find user complaints (in the app reviews) for only one out of the nine updates that are related to this pattern. The users become frustrated with the image quality issues and complain about this pattern. The update of "*Chomp SMS theme add-on*" app on August 17^{th} 2014 has an issue in the displayed icons. Users complain about the icons in the app reviews, such as "*Love the app ever since I purchased it a while back ! The icon not so much.*" and "*Love the update but not a big fan of the new icon*". The development team repairs the icon images and publishes a new update with release notes "*Thanks for valuable user feedback, we appreciate it and have updated the icon to be less fl(u)orescent!*".

Speed of repair:

As shown in Table 4.8, this pattern requires a median of one day to repair (i.e., the median lifetime for the update that precedes the emergency update is one day). The short time to repair indicates that it is not hard for developers to replace the image with ones of higher quality.

Lessons learned for developers:

Developers should examine the quality of the images when new images are added to the mobile app or when the mobile app starts to support new devices with different screen resolutions. Having an image of low resolution may not be suitable for some devices with higher resolution screens. Developers should always consider making

different versions of images with different quality to suit the different devices. A recent study of user reviews have proposed approaches to prioritize the devices that need to be tested based on the reviews (Khalid et al., 2014). Developers may leverage such an approach to prioritize their effort in order to test the quality of images on various devices.

Prior research studies the relationship between the colors that are used in products and the successfulness of the products (Hannah Alvarez, 2014; Neil Patel, 2017; Icons Mind, 2017; Apple, 2018b). Tools have been proposed to assist in evaluating whether the used colors in a product are comfortable for end users (Color-Oracle, 2018; Coblis, 2011). Developers may leverage results and tools of the prior studies to select the most suitable colors for their apps.

Lessons learned for store owners:

Mobile app store owners can enhance the current review mechanism by enabling users to upload screenshots for their complaints. Moreover, app store owners can augment their existing tooling to notify app developers about images that may not appear well on some devices so developers are aware of the issue earlier (e.g., as part of the automated verification of an app that is published by the store owners for each new update of an app before making the update available on the store).

Including incorrect images

Pattern description:

An app has incorrect images. An update is published to replace the incorrect images with the appropriate ones. If an emergency update (U_i) fixes the content of the displayed images in the update preceding the emergency update (U_{i-1}), we consider that the emergency update (U_i) belongs to this pattern.

Root-causes:

We find two root-causes of this pattern:

1. Developers of multiple mobile apps mistakenly package incorrect images across their other apps.

2. Developers forget to include updated images in a new update.

Example updates:

- The *"Camera ZOOM FX Buddy Pack"*, *"Camera ZOOM FX Halloween Pack"*, *"Camera ZOOM FX Composites"*, *"Camera ZOOM FX New Composites"*, *"Camera ZOOM FX Extra Props"*, *"Camera ZOOM FX Props Pack"*, *"Camera ZOOM FX More Composites"*, *"Camera ZOOM FX Picture Frames"*, and *"Camera ZOOM FX CoolBorders"* apps have issues in their updates on June 10^{th} 2014 when all these apps use the same set of images. Developers had mistakenly replaced the images with Halloween photos from the *"Camera ZOOM FX Halloween Pack"* app. The developers repair this issue by updating the images for each of their apps.

- Developers forgot to update the main introductory image for the app portfolio of the *"Pediatrics"* app on January 6^{th} 2014. Developers replace the wrong image with the correct image in an emergency update.

- The "Brain Age Test Free"[14] app stopped using the Hyzap (2018) advertisement network but the developers forgot to remove the Heyzap advertisement in their image. They published an emergency update on July 31^{st} 2014 that removes all images that represent Heyzap. The release notes for this update are *"No more Heyzap"*.

Examples of user complaints:

We find user complaints (in the app reviews) for only one out of the 18 updates that are related to this pattern. The *"Camera ZOOM FX Extra Props"*[15] app has a user complaining that the different *"Androidslide"* apps have the same content. The small number of reviews may be because not all users realize the wrong image content.

Speed of repair:

The median time to repair the updates with this pattern is three days (shown in Table 4.8). In comparison to the pattern of "Including low quality images", this pattern takes a longer time to repair. We believe that the slower repair pace may be due to the users not complaining about this pattern as often as the pattern of including low quality image. The fewer complaints may cause the developers to not realize the issue right after the update, or the developers realize the issue but are not as hard pressed to repair it, given the low number of complaints.

[14] https://play.google.com/store/apps/details?id=brain.age.analyzer
[15] https://play.google.com/store/apps/details?id=slide.cameraZoom.extraprops

Lessons learned for developers:

There exist automated GUI testing tools for mobile apps, such as Robotium (2016). However, automated GUI testing on mobile apps is effort consuming and challenging (Joorabchi et al., 2013). In order to avoid this pattern, developers need better automated testing tools that can reduce their testing efforts. Developers might wish to consider using different build environments for their different apps to ensure that each app has separate resources.

Developers need better code analysis tools that track the dependency between two changed elements in the app (e.g., which parts of the code that are related to which resources in the app) (Hassan and Holt, 2004; Zimmermann et al., 2004). For example, If developers change source code, tools can notify developers with a list of non-code resources (e.g., images) that should be updated.

Lessons learned for store owners:

Mobile app store owners should augment their automated verification process of new updates so that the process would examine all recent updates across a single organization not just for the current app update. The app store may then warn app developers about the repeated content (e.g., resources and configurations) across their different apps. Such automated analysis might also help flag spam apps (i.e., apps with similar content and very slight variations) (Ruiz et al., 2012, 2014).

Having inconsistent permissions

Pattern description:

A mobile app requests a list of permissions. Some of these permissions may not be actually needed or, on the other hand, some needed permissions are not requested. An update is published to remove the request for the not needed permissions or to request the needed permissions that are missing. If an emergency update (U_i) fixes the requested permissions in the AndroidManifest.xml file, we consider that the emergency update (U_i) belongs to this pattern.

Root-causes:

We find four root-causes for this pattern:

1. Developers do not examine whether all the requested permissions by the app are actually needed. For example, developers sometimes use tools to assist during development but these tools may require permissions to function correctly, however such permissions are not needed once the app is published (Telerik, 2014).

2. Developers may use some permissions during development but forget to remove them before issuing their update.

3. Developers forget to add some needed permissions.

4. Developers restrict the permissions for certain SDK versions and discover after issuing an update that the permission needs to be requested for all SDK versions.

Example updates:

- An update of the *"Free Sports Radio"*[16] app on March 8^{th} 2014 requests several not needed permissions. The developers removed the not needed permissions "Read External Storage", "Write External Storage", "Read User Directory", and "Write User Directory" in an emergency update.

- To repair an issue in an update of the *"Higher One Mobile Banking App"*[17] app on April 29^{th} 2014, developers needed the "Read Phone State" permission, which requests read access to the phone state. This permission is not needed any more after the fix. However, developers of the *"Higher One Mobile Banking App"* app forgot to remove the permission for this update. An emergency update removes the permission since it is no longer needed.

- The *"Spanish Translator"*[18] app has an update on November 11^{th} 2014 and the update has unused permissions *Read Phone State* and *Access Network State*. On the next day, the development team publishes a repair that removes these permissions that are not needed.

- An update of the *"One More Clock Widget Free"*[19] app on November 4^{th} 2014 was restricting the "Write External Storage" permission to certain SDK versions. Developers discover that they need to request the "Write External Storage" permission for all SDK versions. To repair this issue, developers correctly reconfigure the permission request in an emergency update by removing the "maxSdkVersion" attribute so the "Write External Storage" permission is requested for all SDK

[16] https://play.google.com/store/apps/details?id=com.MyIndieApp.FreeSportsRadio
[17] https://play.google.com/store/apps/details?id=com.higherone.mobile.android
[18] https://play.google.com/store/apps/details?id=pl.pleng.spanish
[19] https://play.google.com/store/apps/details?id=com.sunnykwong.freeomc

versions.

- An update of the *"Evernote Widget"*[20] app on May 16^{th} 2014 misses to request the "Read Data" and "Write Data" permissions. Developers publish an emergency update to request these missing permissions.

Examples of user complaints:

We find that two out of 13 updates that are related to this pattern (the updates for the *"One More Clock Widget Free"* and the *"Evernote Widget"* apps) have user complaints.

The missed permissions introduces high severity issues as the app may not work because of these missed permissions. Below are examples of user complaints:

- The update of the *"Evernote Widget"* app on May 16^{th} 2014 misses to request the "Read Data" and "Write Data" permissions. Users start to complain that the updated app is not working. Users leave the following reviews on the same day of the update, such as a review *"Application No longer working correctly since evernote update"*, and another review with title "Stopped working" and content *"The new update does not work"*. Developers publish the emergency update that requests the missing "Read Data" and "Write Data" permissions.

- The update for the *"One More Clock Widget Free"* app on November 4^{th} 2014 restricts the request for the "Write External Storage" permission to some SDK versions. Users start complaining that they cannot open the app. For example, user wrote a review on November 4^{th} 2014 with title *"Was good"* and comment text *"App was great till latest update, then wouldn't open. Will try again later"*. Developers publish an emergency update on the next day to fix this permission issue.

[20] https://play.google.com/store/apps/details?id=com.evernote.widget

Speed of repair:

As shown in Table 4.8, the median time to repair the updates related to this pattern is one day. The reason for this fast repair can be explained as follows:

- In the case where the emergency update removes the unneeded permissions, we think the fast turnaround may be due to developers taking permission requests very seriously. For example, according to a recent survey issued on 2,272 participants, 49% of mobile app users do not download at least one app due to privacy issues (Stuart Dredge, 2013).

- In the case where the emergency update adds the missing permissions, we find that reviews often report app crashes for missing to request the needed permissions. The app crashing may be the reason for this fast speed of repair.

Lessons learned for developers:

Developers should better track the requested permissions and their corresponding source code or third party libraries. With strong and up to date traceability links between requested permissions and the source code of an app or third party libraries, developers can ensure that all requested permissions are needed and all needed permissions are requested correctly (stackoverflow, 2013, 2011, 2014).

In order to track the inconsistency in the requested permissions, many researchers introduced tools to verify the needed permissions (Felt et al., 2011; Xu et al., 2013; Pandita et al., 2013; Gorla et al., 2014). Developers should leverage the existing permissions tracking tools to identify any inconsistency between the requested permissions and the app behavior.

Lessons learned for store owners:

Mobile app store owners should leverage the existing permission tools in order to prevent app developers from issuing updates with unneeded or missing permissions.

Having inappropriate SDK versions

Developers define the minimum and target SDK versions in the Android Manifest file. The minimum SDK version represents the minimum SDK version that needs to be installed on the mobile device in order to assure that the app runs properly (Google, 2018c). Users, who installed a later version of the Android platform, can run the updated app. For example, if a user has SDK version 5.0 installed on his mobile device and the app has a minimum SDK version 7.0, this update will not appear to this user (Google, 2018c; Simon Vig Therkildsen, 2012). Developers change (usually increase) the minimum SDK version because they use new features that are introduced in a certain SDK version. By upgrading the minimum SDK version to a higher value, developers prevent the update from being installed on devices with lower SDK versions. If the app runs on an older SDK version, the app may crash because the app may call features that are not supported by the older SDK version (Simon Vig Therkildsen, 2012).

The target SDK version represents the SDK version that is targeted by the development team (Google, 2018c). Adjusting the target SDK version to a certain value means that this SDK version is the version that development team use to test the app. For example, if an app has a target SDK version lower than the user's installed SDK version, the Android platform may apply compatibility behavior in order to make the app run in the same way as expected by app developers (Google, 2018c). For example, The

"Honeycomb" version of Android provides a set of themes called "Holo" themes, by identifying the target SDK version to a lower version than Android Honeycomb, the Android platform will not enable the "Holo" themes for these apps in order to prevent issues like drawing a black text on a black background (Google, 2018c; Simon Vig Therkildsen, 2012).

We find 18 emergency updates that are due to issues related to SDK versions (either minimum or target SDK versions). Eight updates are done to address the issue of missing to update an old SDK version and ten updates are done to address the issue of using a wrong SDK version.

Pattern description:

A mobile app requires a certain SDK version to run, however the app fails to specify the need for this SDK version in its AndroidManifest.xml file. An update is published to fix the needed SDK version of the app. If an emergency update (U_i) fixes the required SDK version of the app, we consider that the emergency update (U_i) belongs to this pattern.

Root-causes:

We find two root-causes for this pattern:

1. A new version of the SDK often provides new features, such as additional APIs. Developers may leverage such new features from a new version of SDK. However, an update might miss specifying the need for a new SDK version in the AndroidManifest.xml file.

2. Developers make a code change and downgrade or upgrade the SDK version.

After the update, developers discover that the SDK version change introduces issues (e.g., menus do not appear) on some devices. Developers perform an emergency update to correct the SDK version to the version that is suitable for the new changes, or to revert back their new changes and to continue using the original SDK version.

Example updates:

- An update of the *"Stock Watcher"*[21] app on September 25^{th} 2014 misses to change the target SDK version. Developers release an emergency update on the next day in order to update the SDK version with release notes *"Fix the disappearing menu button on some devices"*.

- An update of the *"DJ"*[22] app on November 5^{th} 2014 increases the target SDK version from 10 to 21. The developers discover an issue that the menu key does not appear, such that users cannot shut down the app. The developers repair the issue by reverting the *SDK* version back to 10.

Examples of user complaints:

We find that four out of the 18 updates that are related to this pattern (for the *"Voxer Walkie Talkie Messenger"*[23], *"DJ"*, *"AgingBooth"*[24] and the *"FatBooth"*[25] apps) have users complaining about symptoms that are related to this pattern:

[21] https://play.google.com/store/apps/details?id=com.mobileappsresearch.stockwatcher
[22] https://play.google.com/store/apps/details?id=com.spartacusrex.prodjlite
[23] https://play.google.com/store/apps/details?id=com.rebelvox.voxer
[24] https://play.google.com/store/apps/details?id=com.piviandco.agingbooth
[25] https://play.google.com/store/apps/details?id=com.piviandco.fatbooth

- The *"Voxer Walkie Talkie Messenger"* app on September 15^{th} 2014 does not update the minimum SDK version. A user posts a review with the title *"Voxer freezing"* and the following review content *"After new update voter is freezing and not letting me listen to full vox messages"*. Developers publish an emergency update on the following day to increase the minimum SDK version from version 8 to version 10.

- The *"AgingBooth"* app has an update on June 20^{th} 2014 and users report issues about this update. For example, a user leaves a review with title *"Crashed on first use"* and content *"Cannot use it. Let's me take pic and then adjust markers. Then it closes unexpectedly for no apparent reason. Uninstalling"*, another user leaves a review with content *"Force closes after I take a picture, as I can see it()s doing the same for everyone else too"*. Because of the large impact on the user experience, the development team publishes an emergency update to repair this pattern in two days.

- Before updating the *"DJ"* app with the emergency update, we observe that the users complain that the app cannot shutdown. For example, a user leaves the review *"Since it()s for free but with add - you cannot shut down anymore this app. Craps!!"*. After the emergency update, the same user confirms that the app is working well *"NOW it works PERFECT! GREAT technical support and best DJ app. THANKS"*.

- The app *"FatBooth"* has an update on June 20^{th} 2014 and users complain about this update. For example, a user writes a comment with the title *"Force close"* and the following text: *"Since last update it automatically force closes"*. Another

user left a comment *"I (t)take a picture, then go to do it and it force close every single time. Stupidity"*. The development team published an emergency update to upgrade the minimum SDK version from version 9 to version 10 and the target SDK version from version 20 to version 21.

Speed of repair:

The median time to repair the SDK version issues pattern is one day (as shown in Table 4.8). The reason of this pattern being repaired fast is that this pattern is easy to fix. Moreover, this pattern has a large impact on users since every user with the inappropriate SDK version is impacted.

Lessons learned for developers:

To ensure the correct running environment of apps, developers must correctly specify the minimum requirement of SDK version in the AndroidManifest.xml file. However, there exist no automated techniques to ensure that the specified minimum requirement of SDK version is the correct one. Therefore, for every update of an app, developers need to verify the correctness of the specified minimum SDK version. Automated techniques are needed to analyze the app source code and identify the minimum SDK version that is needed for the current code, then to notify the developers with any inconsistency between the needed and the requested minimum SDK versions.

Prior research finds that updating to a new SDK version may be harmful since the API change may lead to defects in mobile apps, leading to a negative impact on the user ratings (Bavota et al., 2015; Vásquez et al., 2013). Before updating the SDK version, developers need to understand the impact of the update. A thorough regression testing process is needed to compare the behavior of an app with the old and the new SDK versions.

Lessons learned for store owners:

Mobile app store owners need automated tools that warn developers if they change the source code without updating the needed SDK version, or if the new source code is not compatible with the specified minimum SDK version.

Incorrect debugging mode

Pattern description:

A mobile app uses debugging functionality to store information about the app behavior. Developers find a need to enable or disable the debugging mode. An update is published to change the debugging mode. If the emergency update (U_i) changes app source code in order to enable or disable the debugging mode, we consider that the emergency update (U_i) belongs to this pattern.

Root-causes:

We find two root-causes of this pattern:

- Developers forgot to disable the debugging mode. Developers publish an emergency update to disable the debugging mode.
- Developers find that there is a need to store the debugging information in order to track the app behavior, so an emergency update is published to enable the debugging mode.

Example updates:

- Developers of the *"Office Calculator Free"*[26] app notice that the app was published with the debugging mode mistakenly enabled. An emergency update is published on June 30^{th} 2014 to disable the debugging mode.
- Developers of the *"Mixology^{TM} Drink Recipes"*[27] app need to track debugging information about the app. An update is published on July 2^{nd} 2014 to enable the debugging mode.

Examples of user complaints:

We did not find user complaints about this pattern. The lack of user complaints is likely due to users not being aware of the debugging and with debugging not having a large impact on the user experience with the app (e.g., performance).

Speed of repair:

As shown in Table 4.8, this pattern requires a median of two days to repair. The short time to repair is most likely due to enabling or disabling debugging information not requiring much effort from developers.

Lessons learned for developers:

Developers should have a checklist to review the app configurations (such as enabling or disabling the debugging mode) before issuing the updates, in order to avoid the need for an emergency update.

[26] https://play.google.com/store/apps/details?id=net.taobits.officecalculator.android

[27] https://play.google.com/store/apps/details?id=com.digitaloutcrop.mixology

4.5.2 Updates Due to Source Code Changes

We identify 78 emergency updates that are due to source code changes: five updates are done to address invoking unavailable APIs, 19 updates are done to address advertisement issues and 54 updates are done to address un-handled exceptions.

Invoking unavailable APIs

Pattern description:

A mobile app does not consider the users' installed SDK versions while calling certain API features. App developers publish an emergency update that enables different behaviors based on the availability of API. If an emergency update (U_i) changes the source code to address the invocation of an unavailable API issue, we consider that the emergency update (U_i) belongs to this pattern.

Root-cause:

The root-cause for this pattern is that developers fail to consider the availability of an API before invoking it in their code.

Example updates:

An update of the *"Word Learner Vocab Builder GRE"*[28] app on June 19^{th} 2014 has an issue in getting the app data using the Android "Context" object. The issue exists because the app invokes APIs that are not available in some versions of the Android SDK . An emergency update with version *"2.3"* is published to handle the described issue with release notes *"Version 2.3: Fixed crash on Android 4.2 and above"*.

Examples of user complaints:

We find complaints about the app crashing but we cannot be sure that the complaints about app crashing are related to the code changes in the emergency update.

Speed of repair:

As shown in Table 4.8, the "Invoking unavailable APIs" pattern requires a median of two days to repair. The "Invoking unavailable APIs" pattern needs longer time to fix than the other code related patterns (e.g., "Un-handled exceptions"). The reason may be that the "Invoking unavailable APIs" pattern needs more investigation and development effort than the other source code related patterns.

Lessons learned for developers:

We notice that developers handle this pattern by adapting the code to behave in a different ways depending on the availability of APIs on the user device. Development tools (e.g., development IDEs) can warn developers when their code calls certain methods that are not provided for some SDK versions that are configured by the app. Similarly, apps store owners can easily warn developers about such issues as part of their automated verification of new updates.

[28] https://play.google.com/store/apps/details?id=com.wordLearner.Free

Advertisement issues

Pattern description:

A mobile app uses third party libraries to display advertisement. Developers find an issue that is related to displaying advertisement. An update is published to ensure the correct display of advertisement. If an emergency update (U_i) changes app source code to fix issue in the calls to the ad libraries, we consider that the emergency update (U_i) belongs to this pattern.

Root-causes:

We find two root-causes of this pattern:

1. Developers use a pseudo value for the ad identifier and forget to update the ad identifier with the correct value.

2. Developers do not validate how the ad will be loaded and displayed in the app.

Example updates:

- An update of the "*Tractor Pull*"[29] app on May 11^{st} 2014 has an issue in setting the ad identifier for the displayed advertisement. An emergency update is published on the next day to fixe the ad identifier value.

- An update of the "*KWCH 12*"[30] app on June 5^{th} 2014 does not handle issues related to the rendering of advertisement. An emergency update is published on the next day to handle such issues. The emergency update adds retry mechanisms to load advertisement.

[29] https://play.google.com/store/apps/details?id=com.anddgn.tp.main
[30] https://play.google.com/store/apps/details?id=com.newssynergy.kwch

Examples of user complaints:

We did not find user complaints about advertisement in the updates that precede the emergency update. On the other hand, we find 128 recent reviews of the emergency updates that are related to this pattern where users complain about the intensive existence of the advertisement in the app. Some users threaten to uninstall the app. For example, on the same day of the emergency update of the "*Tractor Pull*" app, users posted negative reviews such as "*Fun game but since last update get killed with ads! Hard to run gas peddle when a stupid ad covers right half of screen! Uninstall!!!*".

Speed of repair:

As shown in Table 4.8, this pattern requires a median of one day to repair. The short time to repair may be because displaying advertisement is one of the main sources of revenue for the free-to-download app developers.

Lessons learned for researchers:

More research needs to be done in order to provide recommendation about the common usability issues (e.g., best practices and common pitfalls) surrounding the integration of advertisements in mobile apps.

Un-handled exceptions

Pattern description:

Mobile app code does not handle all possible scenarios and app crashes in certain scenario. An update is published to handle exceptions. If an emergency update (U_i)

changes app source code to handle some previously unhandled exceptions that may be thrown by the code, we consider that the emergency update (U_i) belongs to this pattern.

Root-cause:

The root-cause for this pattern that developers do not handle all possible raised exceptions that may occur when users run the apps.

Example updates:

- An update of the *"Photosphere Free Wallpaper"*[31] app on June 27^{th} 2014 has an issue in setting the scroll speed for users if the mode is auto scroll. The app code does not handle the case if the user does not provide scroll speed value. An emergency update is published on the next day (on June 28^{th} 2014) to handle the exception by setting the scroll speed to a default value with release notes *"Fixed issue with auto scroll"*.

- An update of the *"SlenderMan LIVE"*[32] app on April 3^{rd} 2014 has an issue in the code that opens the camera and sets the orientation of the displayed preview of the camera images. The app code does not handle the case of an exception occurring while opening and setting the camera preview. An emergency update is published on the next day (on April 4^{th} 2014) to handle the exception of the un-handled cases with release notes *"Fixed upside down camera !"*.

- An update of the *"First Aid Emergency & Home"*[33] app on December 8^{th} 2013 has an issue in loading the In Case of Emergency (ICE) profile data. The issue occurs

[31] https://play.google.com/store/apps/details?id=fishnoodle.photospherewp_free
[32] https://play.google.com/store/apps/details?id=www.agathasmaze.com.slendermanlive
[33] https://play.google.com/store/apps/details?id=com.nikolay.arfa

because the code does not handle the case of the user not providing all the requested information. An update is published on the next day to handle the null cases with release notes *"Fixed crash related ICE Profile"*.

- An update of the *"Flashlight + Clock"*[34] app on June 9^{th} 2014 has an issue in setting the visibility of component as the code does not handle case if component is null. An update with version *"1.1.2"* is published on the next day to handle the null cases with release notes *"Version 1.1.2 Crash bug fixed"*.

- An update of the *"FeaturePoints: Free Gift Cards"*[35] app on April 30^{th} 2014 has an issue in connecting to Google+ as the code does not handle the case of the user data being null. An update is published to handle the null cases with release notes *"Fixed crash when connecting to Google+"*.

Examples of user complaints:

We find some user complaints about the app crashing but similar to the "Invoking unavailable APIs" pattern, it is difficult to be sure that these complaints are related to exceptions.

Speed of repair:

As shown in Table 4.8, this pattern requires a median of one day to repair. The short time to repair can be explained as this pattern causes the app to crash with the unhandled exceptions.

[34] https://play.google.com/store/apps/details?id=flashlight.led.clock
[35] https://play.google.com/store/apps/details?id=com.tapgen.featurepoints

Lessons learned for developers:

Although un-handled exceptions may produce critical issues, research illustrates that un-handled exceptions are not often identified during code review. For example, Bacchelli and Bird (2013) study code review comments for Microsoft code. Bacchelli et al. find that only 4% of code review comments are about handling exceptions. Developers need to perform more efficient code review mechanism and possibly automated tools in order to avoid the occurrence of this pattern.

We find 35 of the 54 updates that are related to this pattern are due to unhandled null pointer exceptions. There exists a slew of tools that can identify possibly unhandled null pointer exceptions (FindBugs, 2015). Developers and store owners should make use of static analysis tools to avoid the need for emergency updates.

4.6 Limitations and Threats to Validity

In this section, we discuss the limitation and threats to validity of our findings.

4.6.1 Construct Validity

We select 1,000 emergency updates based on the emergency ratio that is defined in this chapter. The emergency ratio depends on the number of days after the last update and the median update lifetime. As illustrated in Table 4.1, the lifetime for the updates preceding the emergency updates is less than or equals to 5% of the median lifetime of the app. Such a low emergency ratio indicates that such updates are most likely emergency updates. However, some mobile apps may have an unstable update cycle. An update with a low emergency ratio may not be an actual emergency update. Including other

factors in defining the emergency updates such as considering the standard deviation of the release cycle is another possible alternative definition of emergency updates.

We leverage a heuristic to identify emergency updates. The heuristic may not be 100% accurate. There is a chance that a developer might release two updates back to back even though the second update is not urgently needed. However, we feel the chances of such rapid updating is very low. For example, as illustrated in Table 4.2 we find one app (for the *"AutoZone"*[36] app) where the lifetime for the update preceding the emergency update is 21 days, while its median lifetime is 568 days. However, for this particular update, many issues are fixed. The release notes are as follows: *"Version 2.0.1 Multiple bug fixes, including: * Accurate product fitment notes. * AutoZone Rewards login issue. * Free Repair Guide images issue. * Improved accuracy with barcode scanning"*. Therefore, such an update is not included in our manual investigation of patterns of emergency updates.

We manually identify the patterns of emergency updates. Although we examine the decompiled APK file artifacts of the updates, the release notes and the user reviews, we are not experts in the development of these mobile apps. Our observation can be biased by our knowledge. To minimize the bias, two researchers (including myself and a collaborator) work together during the manual investigation. However, to further address this threat, interviews and users studies of such mobile apps are required to better understand the rationale for these emergency updates.

[36] https://play.google.com/store/apps/details?id=com.autozone.mobile

4.6.2 Internal Validity

Although we find eight patterns of emergency updates, not all studied emergency updates follow one of our eight patterns. In short, we do not consider our patterns as a comprehensive set of emergency updates patterns. Instead, our work is a first step in creating a richer and more complete set of such patterns. As more updates are examined, we expect that more patterns will emerge.

The key contribution of our study is to raise awareness about the fact that many of these emergency updates share common reasons and by documenting such patterns we hope to assist in improving the quality assurance processes for app updates.

4.6.3 External Validity

Our study is based on the 10,747 top free-to-download apps from the Google Play Store. App Annie lists the top apps from each category of apps for the US market. There might be other top apps in other areas of the world. Including other mobile app stores, such as iOS store, and including the top popular apps in different countries (not only the US) would complement our study. Our results are based on running a Google Play Store crawler for 12 months. Our empirical study may be improved by including the mobile apps' data from a longer run period of our crawler. As any work that identifies patterns, our work is a first step towards creating a catalogue of such patterns. We expect that more patterns will emerge over the years. Although we find other issues that are fixed in emergency releases (e.g., we find one update that fixes the displayed text of a field), we did not formally document them as patterns since we find too few instances of them to claim a recurring pattern. Future studies should explore the generality of these patterns.

The Google Play Store crawler acts as a Samsung S3 device in order to download APK files. Some apps may have multiple APK files for different mobile devices. Therefore, other mobile devices may download different APK files from such apps. Crawling Google Play store with another device may complement our study.

The Google Play Store limits the number of retrieved user reviews for an app. Such a limitation is also noted by recent research by Martin et al. (2015). In our study we try to overcome this issue by running the Google Play Store crawler on a daily basis to ensure that we get as many reviews as possible (since the store will not return more than 500 new reviews since our last crawl).

Our study focuses on the top free-to-download apps since free-to-download apps are the majority of the apps in the Google Play Store. Moreover, free-to-download apps may not be totally free as some free-to-download apps include paid features through in-app purchases or subscriptions. Our study may benefit from including the top non-free apps and comparing the difference in the emergency updates of both free-to-download and non-free apps. However, one would need access to the APK files of these non-free apps by purchasing such apps, requiring a substantial amount of funds.

Our findings are based on a study of the top 1,000 emergency updates. The characteristics and patterns of the emergency updates that are not in the top 1,000 may be different from our findings. Our study may be improved by examining more emergency updates (e.g., analyzing randomly selected updates instead of the top 1,000 emergency updates).

4.7 Related work

In this section, we present prior research that is related to our work. In particular, we focus on prior research in the area of rapid releases.

4.7.1 Rapid Releases

Many organizations are moving from a traditional slower release cycle to a rapid release cycle (Souza et al., 2015; Mäntylä et al., 2015; Khomh et al., 2015). For example, Mozilla Firefox has shifted from releasing every 12 to 18 months to releasing every six weeks. (Souza et al., 2015). Rapid releases enable the delivery of software in a shorter time and enables reacting rapidly to customer feedback (Mäntylä et al., 2015; Khomh et al., 2015). However, there is little knowledge about the impact of a faster release cycle on the quality of software. Thus, researchers have explored the impact of rapid releases on quality (Souza et al., 2015; Mäntylä et al., 2015; Khomh et al., 2015, 2012; Souza et al., 2014).

Khomh et al. (2012, 2015) study the impact of changing from a traditional long release cycle to a rapid release cycle in the Mozilla Firefox by comparing the number of post-release bugs, median uptime, and crash rates. Khomh et al. (2012, 2015) illustrate that the number of reported bugs per day in rapid releases does not change significantly in comparison to traditional long releases, while defects are fixed faster in rapid releases. Khomh et al. also find that users discover bugs faster because the program crashes more quickly than in traditional releases.

Mäntylä et al. (2015) study the testing of rapid releases for the Mozilla Firefox browser. The study collects different metrics that are related to the testing activities; such as the count of tests that are executed per day, the count of testers who are working in the

project per day; they compare the testing activities within traditional releases to the testing activities within rapid releases. The study finds that with rapid releases developers test less compared to traditional releases.

Souza et al. (2014) study the impact of rapid releases on the software quality by comparing the bug reopening rate in traditional and rapid releases. They study traditional and rapid releases of Mozilla Firefox and find that rapid releases have a 7% increase in the bug reopening rate than traditional releases. Hemmati et al. (2015) find that a risk-driven prioritization technique has a higher accuracy in prioritizing the test cases in rapid releases than other prioritization techniques.

All prior work on rapid releases is done on Mozilla Firefox, while we study the mobile apps in the Google Play Store. Moreover, we look at another type of release, which is very rapid by definition, i.e., emergency updates.

4.8 Chapter Summary

Mobile app stores provide an update mechanism that enables app developers to rapidly publish new updates to their users in a cost effective manner. Developers leverage this mechanism to publish emergency updates that are published soon after the previous update.

In this chapter, we study emergency updates in the Google Play Store by analyzing more than 44,000 updates based on around a year of monitoring the update activities of over 10,000 of the top free-to-download apps in the store.

By analyzing the top 1,000 emergency updates, we find that:

1. The emergency updates are often updates with a long lifetime (i.e., they are rarely

followed by another emergency update). Users should update their apps when there is an emergency update without being concerned about another update showing up soon afterwards.

2. Emergency updates rarely include a description in their release notes about the rationale for such an update.

3. The updates preceding the emergency updates receive a higher ratio of negative reviews than the emergency updates.

We identify eight patterns of emergency updates in two categories, updates due to deployment issues and updates due to source code changes. For each pattern, we document the description, root-causes, example updates, examples of user complaints, speed of repair, and the takeaway from this pattern for users, developers, researchers and app store owners. Our findings can help developers and app store owners avoid emergency updates in order to improve the quality and user satisfaction of their apps.

Our study is a first step in creating a rich catalogue of patterns of emergency updates. Future studies should explore additional emergency updates in order to augment our identified patterns.

In the next chapter, we study how the reviewing mechanism can help in spotting good and bad updates. Hence, store owners can leverage user reviews to proactively limit the distribution of bad updates

CHAPTER 5

Studying How the Reviewing Mechanism Can Help Spot Good and Bad Updates

DEVELOPERS always focus on delivering high-quality updates to improve, or maintain the rating of their apps. Prior work has studied user reviews by analyzing all reviews of an app. However, this app-level analysis misses the point that users post reviews to provide their feedback on a certain update. For example, two bad updates of an app with a history of good updates would not be spotted using app-level analysis. In this chapter, we examine reviews at the update-level to better understand how users perceive bad updates. We focus our study on the top 250 bad updates (i.e., updates with the highest increase in the percentage of negative

reviews relative to the prior updates of the app) from 26,726 updates of 2,526 top free-to-download apps in the Google Play Store. We find that feature removal and UI issues have the highest increase in the percentage of negative reviews. Bad updates with crashes and functional issues are the most likely to be fixed by a later update. However, developers often do not mention these fixes in the release notes. Our work demonstrates the necessity of an update-level analysis of reviews to capture the impressions of an app's user-base about a particular update.

5.1 Introduction

App developers focus on publishing high-quality updates to improve or at least to maintain the rating of their apps. A 2015 survey shows that 77% of the users would not download an app with a rating that is less than three stars (Martin, 2015).

Mobile app stores, such as the Google Play Store and the Apple App Store, enable app developers to rapidly deploy new updates of their apps. In turn, users are able to provide update-level feedback to developers. A recent survey of 138 app developers highlights app developers' need for understanding the characteristics of impactful updates (i.e., updates that have an impact on the rating of an app) (Martin et al., 2016). App stores are starting to show the update rating (Google, 2018). Such update-level ratings are likely to impact whether users download an app or not, highlighting the importance of the update rating for app developers.

However, prior work (including our own prior work) mostly focused on studying reviews at the app-level instead of taking an update-centric view to capture the dynamic nature of the response of the user-base for a particular update. Recently, Gao et al. (2018) studied topics that are raised in reviews of each update of an app. Gao

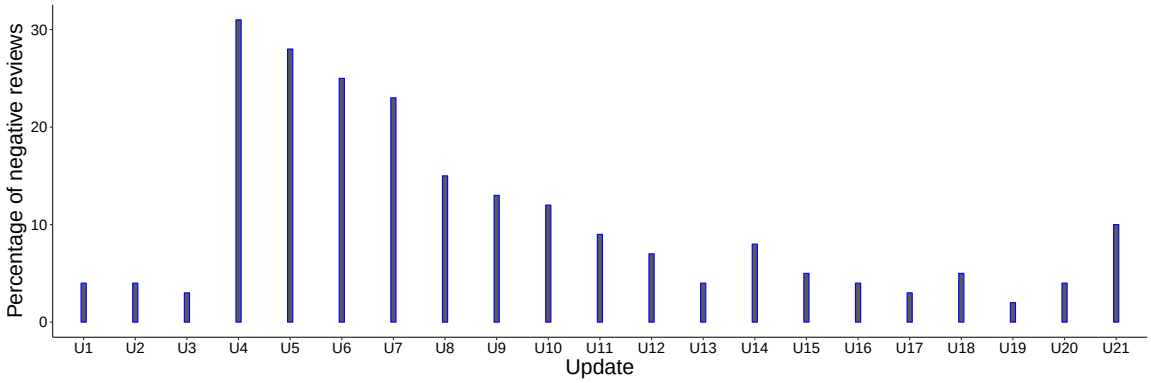

Figure 5.1: The percentage of negative reviews for the *"GasBuddy: Find Cheap Gas"* app.

et al. observed that the distribution of the raised topics for an app changes with each update. In addition, Martin et al. (2016) showed that taking an update-centric view when studying mobile apps is important. For example, an update may lead to many crashes about which users complain. The following update may address the crashes, thereby introducing performance issues. An app-level analysis of reviews will observe that reviews complain about crashes and performance issues without identifying the reality that user complaints were about two different updates. In addition, the crash is already addressed and the app currently has a performance issue. Our work performs an in-depth analysis of mobile app reviews through an update-centric view. Our main target audience consists of researchers. For example, researchers could benefit from our proposed approach by studying the reviews of each update to analyze how the user-perceived quality of an app changes over time.

The necessity of studying reviews at the update-level is demonstrated by the following real-life example. Figure 5.1 shows the percentage of negative reviews per update of the "GasBuddy: Find Cheap Gas" app. As shown in Figure 5.1, the percentage of negative reviews increased after the U_4 update (September 13^{th} 2016) of the app. The

update forces users to enable the GPS location to locate the nearest station instead of searching for the already saved favorite stations. Users complained about sharing their GPS location. On the following day (September 14^{th} 2016), developers deployed the U_5 update that restored the favorites search. As shown in Figure 5.1, the percentage of negative reviews started to decrease after deploying the fix for the raised issue. An app-level analysis of reviews would fail to identify that user complaints are about a GPS issue that is already addressed in the following updates.

In this chapter, we present an in-depth analysis of bad updates (i.e., updates with an increased burst of the percentage of negative reviews) of mobile apps. In particular, we analyzed 26,726 updates and 26,192,781 reviews of 2,526 top free-to-download apps in the Google Play Store. We observed that (1) the update-level analysis is useful for identifying how users perceive the updates of an app over time and (2) the negative reviews of bad updates are different from the negative reviews of regular updates. In particular, the negative reviews of bad updates are more descriptive and contain more update-related information than the negative reviews of regular updates. These results motivated us to further analyze such bad updates to learn how users perceive a bad update and how developers could recover from bad updates. In particular, we addressed the following research questions:

RQ1: *What do users complain about after a bad update?*

A manual analysis of the release notes and negative reviews of the top 250 bad updates in our dataset shows that functional complaints, crashes, additional cost and user interface issues are the most frequently raised issues in bad updates. We observed that apps in the financial and social categories have the highest percentage of bad updates. In addition, we measured the negativity ratio of an

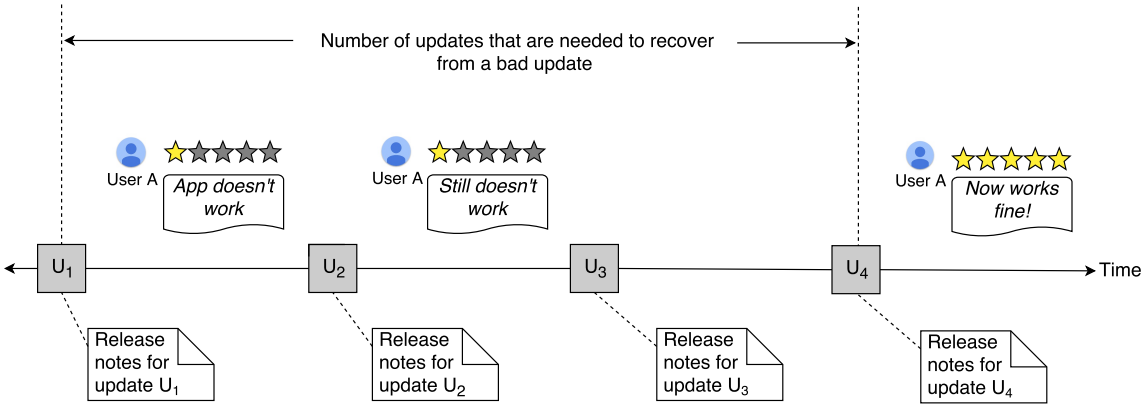

Figure 5.2: An overview of our approach for identifying how many updates are needed to recover from a bad update U_i

update U_i as the ratio of the percentage of negative reviews before update U_i to the percentage of negative reviews of update U_i. We observed that updates where feature removal and user interface issues are raised have the highest negativity ratio.

Our findings show that bad updates are not only perceived as bad because of functional issues as previously observed in prior app-centric studies (McIlroy et al., 2016b). Instead, we observed that crashes, additional cost and user interface issues are the second most-often raised issues in bad updates.

RQ2: *How do developers recover from a bad update?*

Figure 5.2 shows an example of a bad update U_1 that makes the app crash. We determined if an issue was addressed by looking at updated reviews. For example, as shown in Figure 5.2, a user complained about the crash after update U_1. The same user changed their review of the following update U_2 to say that the app *"Still does not work"*. Finally, after update U_4 the user reported that the crash was addressed.

Out of 250 manually investigated bad updates, we located evidence that developers could recover from 105 bad updates. For these updates, recovery was most likely when response time, crashes, network problems, and functional issues were raised (100%, 68%, 60%, and 59% respectively). In the release notes of 44% of the updates that address the raised issues in bad updates, developers explicitly mentioned that they addressed an issue. We measured the differences in the negativity ratio (Neg_{Diff}) as the negativity ratio of a bad update - the negativity ratio of the fixing update. We measured the Neg_{Diff} in both cases when developers mentioned explicitly that an update addresses the raised issue and when developers mentioned general release notes (e.g., *"bug fixes"*). We observed that the cases where developers mentioned explicitly that they addressed the issues of a bad update have a higher difference in the negativity ratio (the median Neg_{Diff}= 1.9) than the cases where developers do not mention that the issues were addressed (the median Neg_{Diff}= 1.7). Hence, we recommend that developers mention explicitly in their release notes that an issue was addressed to encourage users to download the update and eventually update their rating leading to an improved overall app rating.

The purpose of our study is twofold. First, we demonstrate the importance of update-level analysis compared to the traditional app-level view that most prior work (including our own work) on app review analysis takes. Hereby, we propose an approach which can be used by researchers. Second, we use the update-level analysis to understand the characteristics of bad/good updates and to analyze how developers recover from bad updates, hereby allowing researchers to understand the opinion of users about an app with a much finer granularity.

The rest of this chapter is organized as follows. Section 5.2 describes our methodology for identifying bad updates. Section 5.3 describes our motivational study of bad updates. Section 5.4 analyzes the characteristics of bad updates. Section 5.5 describes the implications of our work. In addition to the analysis of bad updates, to complete our analysis, we study why users perceive an update as good in Section 5.6. Section 5.7 describes threats to the validity of our findings. Section 5.8 concludes this chapter.

5.2 Methodology

In this section, we describe our approach for studying bad updates from the Google Play Store. Figure 5.3 gives an overview of the steps of our approach. We detail each step below.

5.2.1 Collecting Data

In this section, we describe our selection criteria and data collection process.

Select Top Android Apps

We selected apps for our study based on the following criteria:

- **App popularity:** We focused on popular apps as we expect that these apps are maintained by developers who care about the rating of their app, and have a large enough user-base that has an opinion about the app.
- **App diversity:** We selected top popular apps across all categories in the Google Play Store to ensure that the app categories do not impact our observations.

We selected the top free-to-download apps in 2016 using App Annie's report on popular apps (AppAnnie, 2018). We focused on free apps to avoid the impact of app price on our analysis. The price of an app may have a significant impact on how users perceive an update (Noei et al., 2017). App Annie's report of popular apps contains 28 app categories (e.g., games and finance categories). We selected the top 100 apps in each category. In total, we selected 2,800 apps for our study. We found that 214 apps were repeated across categories and 60 apps were already removed from the store when we started our study. Hence, we conducted our study on 2,526 top apps. We did not use the same dataset for the study in Chapter 4 as we study recent apps (i.e., popular apps in 2016) and we need to collect more data than the collected data in the previous study (e.g., we collect the changes in user reviews to understand how developers could recover from bad updates).

Crawl App Data Over 12 Months

We used a Google Play crawler (Akdeniz, 2013) to collect data from the Google Play Store. For each studied app, we collected the following data:

1. **General data:** app description, app title, the current number of downloads and current rating of the app.

2. **Updates:** release notes of each update.

3. **User reviews:** review title, review contents, rating, review time.

The crawler connects to the Google Play Store using the Samsung S3 device model (as the Samsung S3 device was one of the most popular models at the time that we started to crawl (AppBrain, 2018)). Each time the crawler collects app data, the crawler

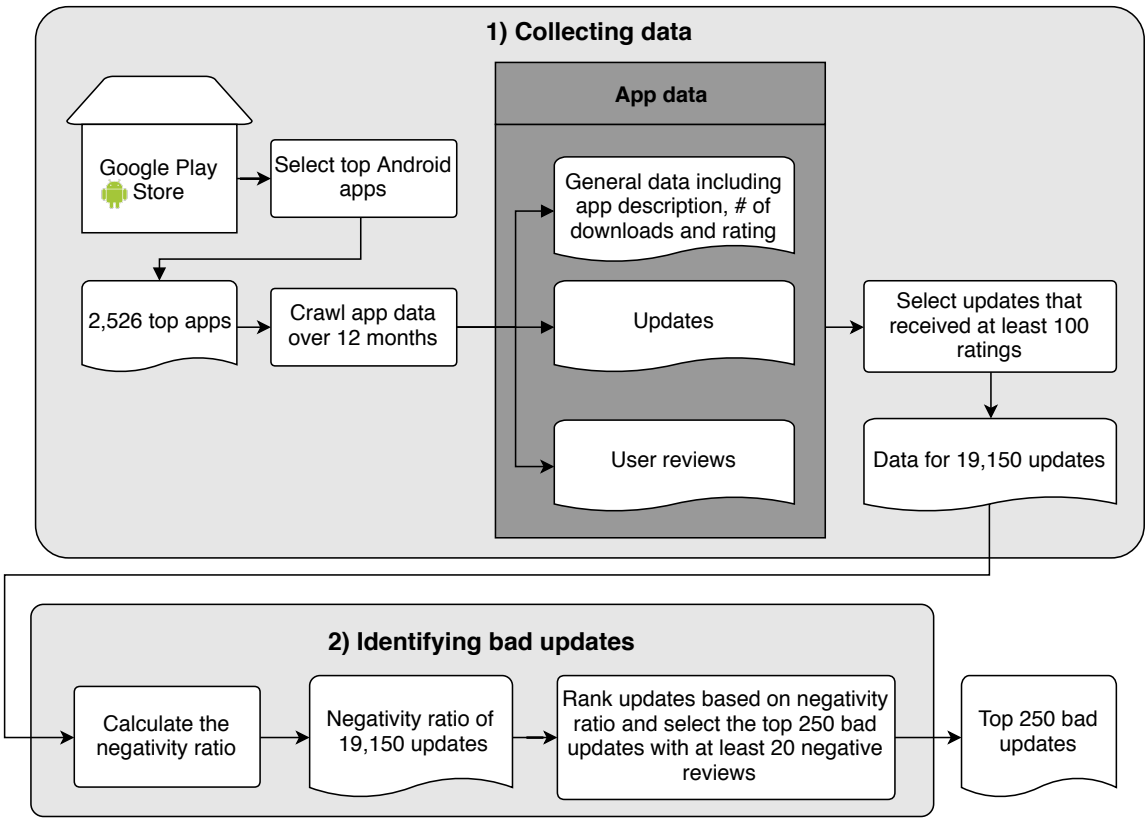

Figure 5.3: An overview of our approach for studying bad updates

stores the current app data (e.g., the current rating) and the latest 500 reviews of that app.

During our study, we observed that apps differ in the amount of posted reviews per day. For example, some apps receive thousands of new reviews per day (e.g., Facebook and Instagram) while other apps receive a small number of new reviews per day. To avoid overloading the Google Play Store while still crawling as much data as possible, our crawler automatically adjusts its crawling frequency per app based on the number of newly posted reviews after each crawl.

The Google Play Store allows users to post only one review per app. Users can modify the contents or rating of their posted review. Our crawler receives a chronological

Table 5.1: Dataset description.

Number of studied apps	2,526
Number of collected updates	26,726
Number of collected reviews	26,192,781
Number of collected changes in reviews	3,470,113

overview of the review changes. We used changes in each review to investigate the time that it takes a developer to address a reported issue (i.e., by studying the time between a user reporting an issue and the same user updating their review to report that the issue was addressed).

We ran the crawler from April 20^{th} 2016 to April 13^{th} 2017. We crawled the store for almost a year as we need app data for a longer period to identify a bad update. During our study period, we collected 26,726 updates, and 26,192,781 reviews with 3,470,113 changes in reviews. We focused on updates with at least 100 ratings to assure that every update has sufficient data for our study. We ended up with 19,150 updates for our study. Table 5.1 describes our dataset. In the next section, we explain how we used the collected data to identify bad updates.

5.2.2 Identifying Bad Updates

In this section, we describe the steps for identifying bad updates. First, we explain how to calculate the negativity ratio which we used to identify bad updates.

Calculating the Negativity Ratio

To identify bad updates, we calculated the negativity ratio for an update U_i as follows. First, we calculated the **percentage of negative ratings** (i.e., ratings of one or

Table 5.2: Mean and five-number summary of the negativity ratio of the 19,150 studied updates.

Metric	**Mean**	**Min.**	**1st Qu.**	**Median**	**3rd Qu.**	**Max.**
Negativity ratio	1.0	0.0	0.8	1.0	1.1	26.3

two stars (Martin, 2015)) **of an update** U_i ($PNR(U_i)$) as the ratio of the number of negative ratings of update U_i to the total number of ratings of update U_i. For example, an update with ten ratings (two ratings with one star and eight ratings with four stars) has a *PNR* of 0.2.

Then, we calculated the **percentage of negative ratings before update** U_i ($PNRB(U_i)$) as the ratio of the number of negative ratings before update U_i to the total number of ratings before update U_i.

Finally, we calculated the **negativity ratio of an update** U_i as follows:

$$Negativity\ ratio(U_i) = \frac{PNR(U_i)}{PNRB(U_i)} \tag{5.1}$$

Note that we link a review to the latest update at the time that the review was posted. A negativity ratio that is lower than one means that users are less negative about the app after releasing update U_i than before. On the other hand, a negativity ratio higher than one means that users are more negative about the app after the release of update U_i than before. Table 5.2 shows the mean and five-number summary of the negativity ratio of all 19,150 studied updates.

Table 5.3: Descriptive summary of the Top 250 bad updates.

Number of studied apps	211
Number of studied updates	250
Number of collected negative reviews	81,273
Number of collected changes in reviews	12,987

Identifying the Top 250 Bad Updates

To identify top bad updates, we focused on updates with the highest negativity ratio. We applied the following approach. First, we ranked all updates based on their negativity ratio. Then for the top 1,000 updates with the highest negativity ratio, if an app has consecutive updates in the list of the top 1,000 updates, we include only the first update in our study. The reason is that we cannot verify whether the negative ratings that are posted for a consecutive bad update are due to an issue with the consecutive update or because users are still complaining about an issue from the previous update.

Figure 5.4 shows an example of an app with nine updates. The three updates U_2, U_3 and U_8 are in the top 1,000 bad updates. We included both updates U_2 and U_8 as users clearly started to complain about these updates, while we excluded update U_3 since we cannot verify whether the negative ratings that were posted for update U_3 were due to a new issue in update U_3, or because users were still complaining about update U_2.

We selected the top 250 bad updates with at least 20 negative reviews to have enough data for our manual analysis to help us understand why users perceive the update as a bad update.

Table 5.3 shows the number of apps, the number of collected reviews and the number of collected review revisions of the studied bad updates.

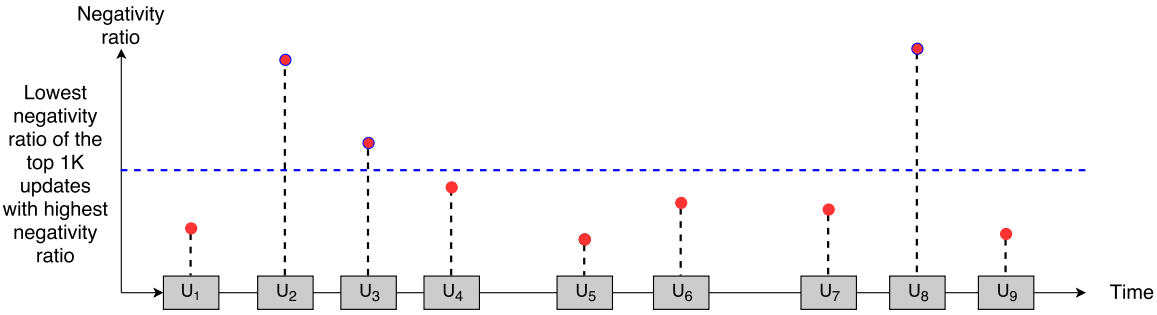

Figure 5.4: An example of an app with nine updates U_1 to U_9. The blue dotted line in the figure shows the lowest negativity ratio of the top 1,000 updates with the highest negativity ratio. In our study, we included both updates U_2 and U_8 as they are separated by other updates while we excluded update U_3 since it follows the bad update U_2

5.2.3 Approach for Identifying the Types of the Raised Issues in a Review

In our analysis of raised issues about bad updates, we need to investigate which issues do users raise in a bad update and what is the difference between the raised issues in the negative reviews of bad updates and those issues of negative reviews of regular updates. To answer these questions, we need to manually read user reviews and identify the issue type (e.g., crash) that is raised in every review. Our approach for the manual analysis is as follows. Two researchers (including myself and a collaborator) manually read the reviews and identified the raised issues and the corresponding issue type of each raised issue. McIlroy et al. manually analyzed the complaints in user reviews of mobile apps and identified 14 issue types (e.g., crashing and user interface issue) (McIlroy et al., 2016b). We used the same issue types as McIlroy et al. (2016b). Note that Maalej and Nabil used several techniques (e.g., bag of words) to automatically classify user reviews into four high-level categories: bug report, feature request, user opinion or rating (Maalej and Nabil, 2015). In our analysis of negative reviews,

Table 5.4: The identified issue types.

Issue type	Description (*D*) - Example (*E*)
Functional Complaint	***D:*** The user complains about a functional issue in the app. ***E:*** *"It does not update cart. Also keeps login me off"*
Crashing	***D:*** The user complains that the app crashes or does not work. ***E:*** *"Always crashes. It says it's updating and then just closes out. Stupid useless app I just get on the Browser to pay my card."*
User Interface	***D:*** The user complains about user interface issues (e.g., layout, icons, colors and style issues). ***E:*** *"Please revert back to old icon. Changing the icon took away the true identity of Instagram."*
Feature Request	***D:*** The user requests from developers to add a certain feature. ***E:*** *"The new update got rid of tabs.... one of the best features of the browser."*
Additional Cost	***D:*** The user complains about the additional cost of the app (e.g., an app has advertisements or asks for additional payment). ***E:*** *"Too much advertisements are ruining the experience."*
Privacy and Ethical Issue	***D:*** The user complains about the private information that is requested by the app or the user complains about ethical issues in the app. ***E:*** *"I would like an explanation of the need for the phone permission. Clearly app permissions in Android have become useless."*
Network Problem	***D:*** The user complains about network or connectivity issues. ***E:*** *"New update running slowly and keeps prompting me to connect to wifi despite already being connected."*
Compatibility Issue	***D:*** The user complains about an issue for a certain device model or a certain Android version. ***E:*** *"Unable to sync lyrics on samsung j5"*
Feature Removal	***D:*** The user requests from developers to remove a certain feature. ***E:*** *"Please remove the news section." or "Mandates to rate. Hate the mandatory ratings after every ride."*
Response Time	***D:*** The user complains about the slow performance or the delay of the app. ***E:*** *"Very slow."*
Uninteresting Content	***D:*** The user complains that the app content is not useful or uninteresting. ***E:*** *"CNN has become too one sided and biased as a news organization. Will uninstall the app."*
Update Issue	***D:*** The user complains that the issue is related to the new update. ***E:*** *"New update is terrible"*
Resource Heavy	***D:*** The user complains that the app consumes too many resources, such as battery, memory, CPU or storage. ***E:*** *"ESPN made it so if you want to listen to podcasts or live radio you are now required to use this app. My battery usage has now jumped to 41% just for this app alone. Uninstalled and will not be using until this issue is fixed."*
Unspecified	***D:*** The review does not contain detailed information about a raised issue. ***E:*** *"Bad very bad"*

we did not use Maalej and Nabil's high-level categories as the proposed categories are too generic and do not provide issue types at the level of detail (e.g., user interface issue type or privacy and ethical issue type) that is necessary for our analysis. Table 5.4 shows the list of McIlroy et al.'s issue types, together with a description and an example of each issue type. If there is a conflict between the two researchers, then both researchers discuss how they interpreted the reviews until both researchers agree on the identified issue types of the manually analyzed reviews. Finally, we calculated the agreement between both researchers using Cohen's Kappa interrater agreement (Cohen, 1960). Cohen's Kappa measures the agreement between the two researchers and provides value ranges from -1 to +1 (Cohen, 1960). The highest Cohen's Kappa measurement value (i.e., +1) means that both researchers identified the same issue types in all examined reviews.

In the following sections, we describe the motivation, approach and the results of our study.

5.3 Motivational Study

In this section, we discuss our motivational study of bad updates. Our motivational study has two parts. First, we demonstrate the importance of update-level analysis of user reviews. Second, we studied the difference between the negative reviews of bad updates and the negative reviews of regular updates. The motivation, approach and the results of our motivational study are described in the following sub-sections.

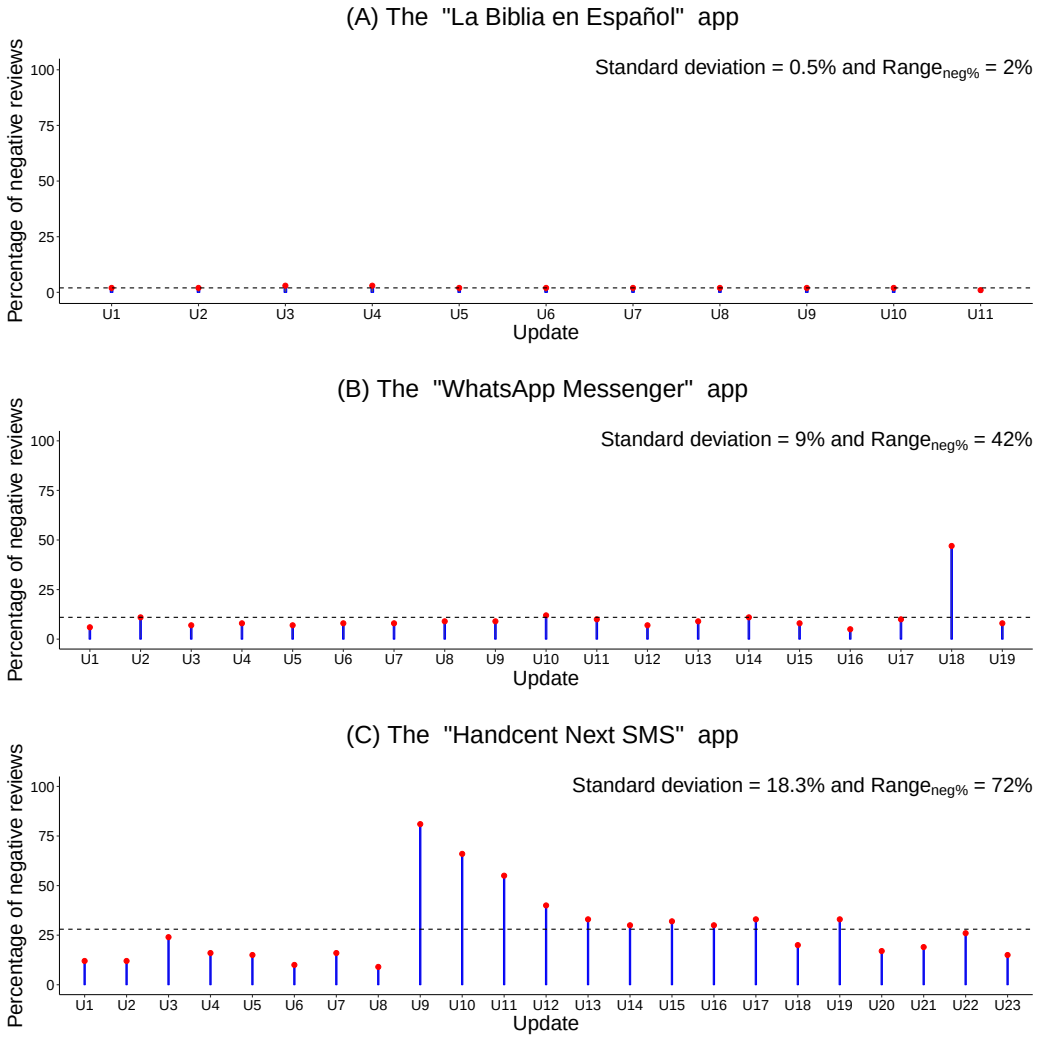

Figure 5.5: An example of: (A) the "La Biblia en Español" app with a low standard deviation (0.5%) and a small $Range_{neg\%}$ (2%), (B) the "WhatsApp Messenger" app with a burst in the percentage of negative reviews and a fast recovery of the percentage of negative reviews (the standard deviation value is 9% and the $Range_{neg\%}$ is 42%), and (C) the "Handcent Next SMS" app with a burst in the percentage of negative reviews and a slow recovery of the percentage of negative reviews (the standard deviation value is 18.3% and the $Range_{neg\%}$ is 72%). The black dotted line in the figure shows the average percentage of negative reviews of every app.

5.3.1 Demonstrating the Need for of Update-Level Analysis of User Reviews

Motivation: We want to demonstrate the importance of update-level analysis over the app-level analysis of reviews. The app-level analysis shows the average percentage of negative reviews of an app across all updates, while the update-level analysis shows the percentage of negative reviews for each update. By comparing how the app-level and update-level views change over time, we could determine whether the update-level view is necessary.

We calculated the percentage of negative reviews instead of using the average rating as the latter does not indicate the percentage of the overall user-base who posted negative reviews about the update. For example, if an app has two updates. The first update has two ratings, one rating with 1-star and one rating with 5-stars. The second update has six ratings, four ratings with 2-stars and two ratings with 5-stars. Both updates have the same average 3-stars, but the percentage of users who posted negative reviews about the update is different (50% for the first update and 67% for the second update).

Approach: We measured the percentage of negative reviews for each update for each studied app. Figure 5.5 shows an example of changes of the percentage of negative reviews for three different apps. We observe from Figure 5.5 that, (1) the percentage of negative reviews may change from update to another (e.g., the "WhatsApp Messenger" and "Handcent Next SMS" apps in cases B and C), and (2) apps vary in how fast they recover from a bad update (i.e., the "Handcent Next SMS" app in case C).

We measured the standard deviation of the percentage of negative reviews for all

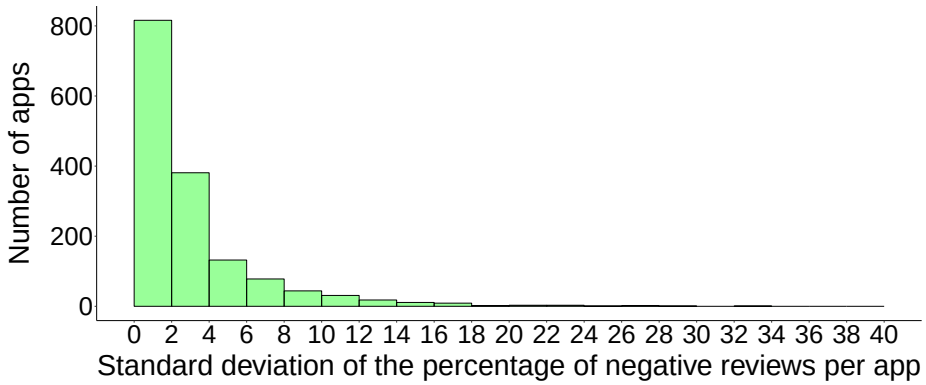

Figure 5.6: A histogram of the standard deviation of the percentage of negative reviews per app

the studied updates. A low standard deviation value means that the percentage of negative reviews is almost stable for that app during the study period. As shown in Figure 5.5 for the "WhatsApp Messenger" app, the standard deviation of the app will not be impacted if there is a bad update and the percentage of the negative reviews recovered quickly in the following updates. Hence, in addition to the standard deviation, we measured the range of the percentage of negative reviews ($Range_{neg\%}$) per app (i.e., the *maximum* percentage of negative reviews per app - the *minimum* percentage of negative reviews per app). A high $Range_{neg\%}$ means that there are cases when there is a burst of negative reviews that may point to a bad update. The $Range_{neg\%}$ alone does not show how fast an app recovers from a bad update. Thus, we measured both the standard deviation and the $Range_{neg\%}$ to identify cases when there is a burst of negative reviews and how fast an app recovers from such cases.

Findings: Figure 5.6 shows the histogram of the standard deviation of the percentage of negative reviews per app. Table 5.5 shows the mean and the five-number summary of the $Range_{neg\%}$ per app.

Table 5.5: Mean and five-number summary of the $Range_{neg\%}$ per app.

Metric	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
$Range_{neg\%}$	9.8%	0%	4%	7%	12%	88%

We observe from Figure 5.6 and Table 5.5 that: (1) The percentage of negative reviews does not vary much between updates (the median standard deviation is 2.3%). (2) There are peaks of negative reviews as the median difference between the maximum and the minimum percentage of negative reviews is 7% and the maximum difference is 88%.

To compare the app-level analysis with the update-level analysis, Figure 5.5 shows the average percentage of negative reviews for the same three cases. As shown in Figure 5.5, the update-level analysis could identify the burst of variation in the negative reviews so users and store owners could easily identify when there is a bad update and when an app recovers from such a bad update.

5.3.2 Comparing the Raised Issues in Bad Updates to the Raised Issues in Regular Updates

Motivation: Before analyzing the raised issues of every bad update, we need to examine whether the overall base of the negative reviews of bad updates differs from the overall base of negative reviews of regular updates. If there is no difference between the negative reviews of bad updates and negative reviews of regular updates, then there is no need for further analysis of bad updates. On the other hand, if there is a difference between the type of raised issues for negative reviews of bad updates versus regular updates, then it is useful to further investigate how users perceive bad updates (i.e.,

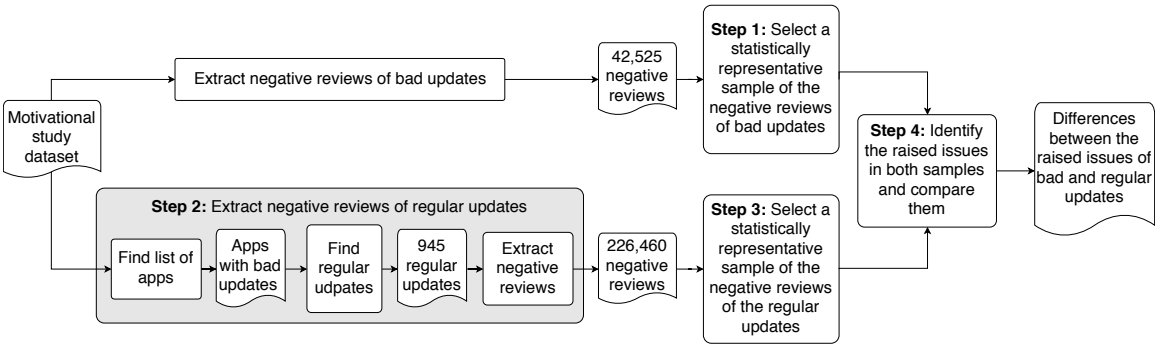

Figure 5.7: An overview of our approach for comparing the raised issues in bad updates to the raised issues in regular updates of our motivational study dataset

what do users complain about after a bad update?) and how developers often recover from bad updates.

Approach: Figure 5.7 shows an overview of our approach for comparing the raised issues in the negative reviews of bad updates versus the negative reviews of regular updates. We followed these steps:

Step 1: *Select a statistically representative sample of the negative reviews of bad updates.* As described in Section 5.2, we observed that apps differ in the number of received reviews. Hence, sampling the overall collected reviews may lead to a bias towards apps with many reviews. Thus, we first randomly selected a statistically representative sample of the negative reviews with a confidence level of 95% and a confidence interval of 5% for each bad update of the top 100 bad updates. Then, we grouped the collected random samples. We ended up with a refined sample of 11,829 negative reviews out of 42,525 negative reviews.

Finally, we randomly selected a statistically representative sample of 372 reviews (out of 11,829 reviews) with a confidence level of 95% and a confidence interval of 5%.

Step 2: *Extract negative reviews of regular updates.* To compare reviews of bad updates and regular updates, both types of updates should be related to the same apps. In this way, we eliminate any bias caused by comparing reviews of different apps. We observed that our motivational study dataset (i.e., the top 100 bad updates) spans 94 apps with 1,450 updates.

To identify regular updates, first, we filtered out bad updates (we kept 1,350 out of 1,450 updates). Then, we ranked the 1,350 updates with their negativity ratio and removed the top 30% bad updates (i.e., updates with the highest negativity ratio), so we can ensure with more confidence that the remaining updates are not bad updates. We ended up with 945 regular updates, which have 226,460 negative reviews in total.

Step 3: *Select a statistically representative sample of the negative reviews of the regular updates.* For each regular update, we randomly selected a statistically representative sample of its negative reviews with a confidence level of 95% and a confidence interval of 5%. Then, we grouped the collected random samples. We ended up with 70,643 negative reviews (out of the collected 226,460 negative reviews).

Finally, we randomly selected a statistically representative sample of 382 reviews (out of 70,643 reviews) with a confidence level of 95% and a confidence interval of 5%.

Step 4: *Identify the raised issues in both samples and compare them.* We manually read the reviews in both samples and identified the raised issues and the corresponding issue type in the negative reviews of both samples (as described in

Section 5.2.3). We measured the agreement between both researchers. In addition, we compared the statistics of the issue types across samples. Finally, we applied a statistical test to examine the statistical difference between the distribution of the issues types in bad updates and regular updates. In particular, we applied Pearson's Chi-squared test because it can be used to test distributions of categorical variables for a statistical difference (Ling, 2008; RTutorial, 2018). We defined the null-hypothesis as the hypothesis that the raised issue types for bad updates are different from those raised in regular updates.

Findings: **The frequency of update-related issues in the negative reviews of bad updates is higher than in regular updates.** Figure 5.8 shows the distribution of each issue type for regular updates and bad updates. Users mention that an issue is related to the latest update in 32% of the negative reviews of bad updates and in 20% of the negative reviews of regular updates. Hence, an important characteristic of bad updates is that users complain more specifically about an issue in the update. Thus, reading the negative reviews of every bad update might provide insight as to why an update is perceived as bad.

The frequency of unspecified issues in the negative reviews of regular updates is almost four times higher than that in bad updates. For regular updates, 26% of the negative reviews do not specify an issue (e.g., a review only says *"Bad"*). On the other hand, for bad updates, only 7% of the negative reviews are not specific. This finding suggests that negative reviews of bad updates are more descriptive than the negative reviews of regular updates.

User interface, feature request, feature removal, additional cost, crashing and

compatibility issues are raised more frequently in the negative reviews of bad updates than in regular updates. For negative reviews of bad updates that raise user interface, feature request and feature removal issues, we observed that users often ask developers to revert back to the old user interface or an old feature. For example, in reviews that request the addition of a feature, users do not ask for *new* features. Instead, users ask for the restoration of a removed feature. In the user interface reviews, users ask developers to return to the original user interface as they felt that the previous update had a better user experience than the new update. Hence, it is important that developers consult with their users before changing or removing a feature.

We measured Pearson's Chi-squared test and we found that the p-value is < 0.01. The Pearson's Chi-squared test result shows that there is a statistical difference between the issue types of regular updates and the issue types of bad updates. As described in our approach section, we measured the agreement between both researchers. We observed that the Cohen's Kappa interrater agreement between both researchers is 0.7.

The update-level analysis is useful for identifying updates with a burst of user complaints (i.e., bad updates). Analyzing a sample of negative reviews of bad updates shows that the negative reviews of bad updates are different from those of regular updates. In particular, negative reviews of bad updates are more descriptive and raise more update-related issues than those of regular updates. Hence, further analysis on what do users complain about after a bad update? and how do developers recover from a bad update? can offer insights about bad updates and how to recover from such updates.

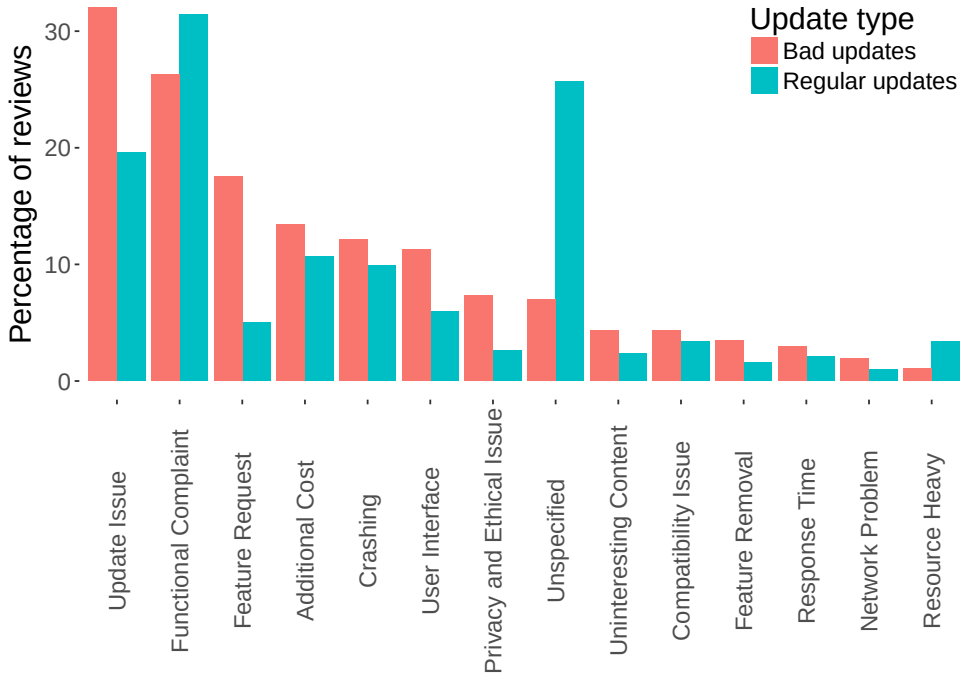

Figure 5.8: Distribution of each issue type for both regular updates and bad updates of our motivational study dataset

5.4 A Study of Bad Updates

In this section, we present our study of the top 250 bad updates. For each RQ, we present our motivation, approach, and results.

5.4.1 RQ1: What do users complain about after a bad update?

Motivation: It is hard to make every user happy. Bad reviews are inevitable. The real problem with bad reviews is when they result in an alienation of the user-base of an app. Prior work studied user complaints in reviews at the app-level (Oh et al., 2013; Iacob and Harrison, 2013; Iacob et al., 2013b,a; Maalej and Nabil, 2015; Villarroel et al.,

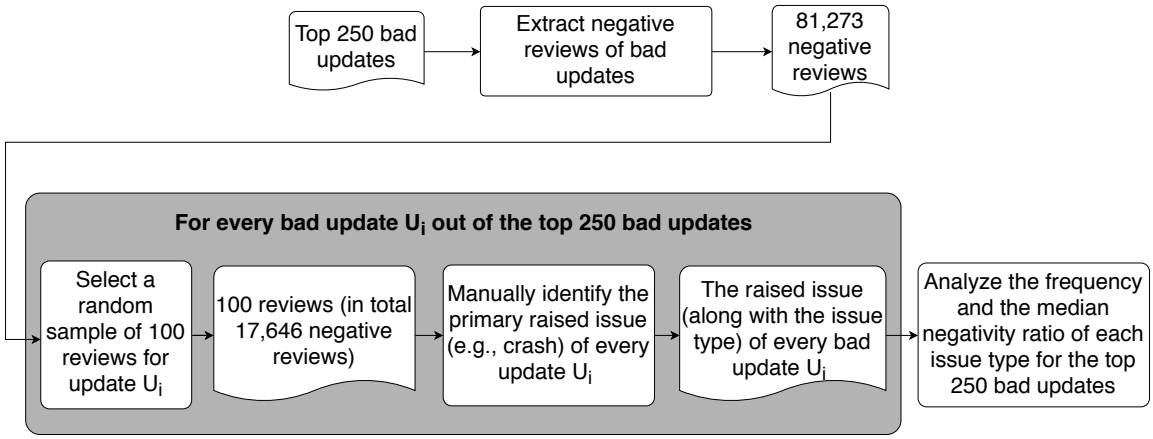

Figure 5.9: An overview of our approach for studying the raised issues in bad updates

2016; Keertipati et al., 2016; Hu et al., 2018a,b). In our motivational study, we demonstrated the need for update-level analysis. In this section, we analyze user complaints at the update-level. Our goal is to understand whether some issue types are more likely to make a particular update be perceived as bad. Our findings can help developers identify the issues that should be dealt with more caution to avoid bad updates.

Approach: We manually analyzed 17,646 negative reviews of bad updates. Figure 5.9 shows an overview of our approach for analyzing reviews at the update-level.

We grouped all reviews per update and analyzed what is the primary raised issue for every bad update. For each bad update U_i, we identified the negative reviews that are posted between the release of update U_i and U_{i+1}. We found 81,273 negative reviews that belong to the top 250 bad updates (a median of 84.5 negative reviews per bad update). Then, for each update U_i, two researchers (including myself and a collaborator) manually read a random sample of 100 negative reviews as we observed that 100 reviews were enough to identify the core issues of a bad update. Both researchers independently read the reviews to identify the raised issues in bad updates. If there

was a conflict between the two researchers, then both researchers discussed how they interpreted the reviews until both researchers agreed on the identified issues for every bad update. As described in Section 5.2.3, we used the issue types that are listed in Table 5.4. The Cohen's Kappa interrater agreement between both researchers for this classification is 0.78.

For each bad update, we observed that most of the negative reviews complained about the same issue (the **primary** issue). Hence, we documented the primary issue for each update. We counted the number of updates that refer to a certain issue type. As described in Section 5.3, If at least 20% of the reviews of an update do not complain about a specific issue (i.e., there is no primary issue), we consider the raised issue for this update as unspecified. Additionally, for each issue type, we calculated the median negativity ratio of the updates that were labeled with this issue as the primary issue. We also compared the percentage of apps with bad updates across app categories.

Findings: **Functional complaint and crashing issues are the most frequently raised issues in bad updates.** Table 5.6 shows the number of updates that were labeled with a certain issue type. We observed that functional complaints (70 updates) were the most occurring primary issue type in bad updates. For 20 out of 70 updates, users complained that they could not log in to the app. For 5 out of 20 login issues, developers mentioned in the release notes of one of the following updates that the login issue was addressed. For the other 15 updates, the release notes have generic content (e.g., *"bug fixes"*). For the two updates out of the five updates with descriptive release notes, the reason was that the login functionality did not work on all devices (e.g., *"Fixed issue on small screens where account button got hidden"*). Testing an app on all possible devices requires considerable time and resources. Developers could benefit from the existing

studies to prioritize the needed devices for testing their apps (Khalid et al., 2014; Noei et al., 2017).

We performed a comparison between the frequency of the identified issue types in our update-level analysis and the identified issue types in the app-level analysis work that was performed by McIlroy et al. (2016b). In particular, we observed that crashes, additional cost and user interface issues occur more frequently at the update-level than at the app-level. On the other hand, reviews with feature requests and network issues are more frequent at the app-level than at the update-level. A possible explanation for this difference is that the two studies were conducted on different apps and analyzed user reviews at different times which may impact the distribution of the raised issues in the two studies. However, the differences between our work and the work of McIlroy et al. indicate that our update-level analysis provides a complementary view that is not available in other prior work on the analysis of mobile app reviews.

Feature removal and user interface issues have the highest median negativity ratio. We observed that the highest median negativity ratio (3.5) occurred for bad updates where users ask for removing a new feature that was added by the latest update. For example, users complained about an additional step that mandates users to create an account to use the app or users asked for the removal of notifications. In another example, the developer improved the app's security by making the user session expire after a particular time, and users need to enter their credentials (i.e., the username and the password) to remain signed in. Although developers initially expected that the added features would be perceived as a good update, users asked developers to roll back this additional feature. Hence, app developers should consult users before adding new features to avoid such bad updates.

Table 5.6: The number of updates that were labeled with a certain issue type and the median negativity ratio of each issue type (ranked by the number of bad updates).

Issue type	**# of bad updates containing this issue as a primary issue**	**Median negativity ratio**
Functional Complaint	70	2.8
Crashing	44	2.5
Additional Cost	35	2.5
User Interface	23	3.4
Privacy and Ethical Issue	23	2.5
Other	18	2.3
Feature Request	17	2.8
Uninteresting Content	16	2.6
Network Problem	10	2.5
Feature Removal	7	3.5
Compatibility Issue	3	2.8
Response Time	2	2.4
Resource Heavy	1	2.1
Total number of bad updates	250	2.7

During our manual analysis of the 23 updates where user interface issues were raised, we observed that user interface issues could be classified into different subtypes. To identify the different subtypes of user interface issues, we used an approach similar to coding (Khandkar, 2009; Borgatti, 1996). We manually read the previously-selected random sample of 100 reviews of each of the 23 updates. Then, we identified the subtypes of the user interface issues. If a new subtype was identified, it was added to the list of identified subtypes. After reading the reviews, we identified five different subtypes of user interface issues in bad updates. Table 5.7 shows the five identified subtypes together with their description. Although the identified subtypes seem minor issues and could be addressed easily (e.g., icons or colors), clearly users care about

Table 5.7: The identified sub-types of user interface issues.

Category	Description (*D*) - Example (*E*)
Logo or icon	***D:*** The user complains about the main app logo or the displayed icons in the apps. ***E:*** *"The previous icon was much much better than this new updates icon... Didn't like this"*
Design and layout	***D:*** The user complains about the design or layout of the screens. ***E:*** *"The layout before was so easy to navigate and use. There is literally nothing I like about the new update."* or *"No more simplicity. So much wasted space and have to scroll miles for miles to look at anything. Not intuitive. Writing is so small can't read anything even tho my screen could accommodate much more. Peoples pics are tiny can't even see who's attending the events. Whoever did this should be fired."*
Colors	***D:*** The user complains about the colors of the screen (e.g., screens are too bright or too dark). ***E:*** *"This new UI is too bright n I don't find it comfortable to use it at night"* or *"I can barely even type this black on black is kinda hard to use"*
Photos/pictures quality	***D:*** The user complains about the quality of an image. ***E:*** *"New version is not good picture quality is very bad plz solved tha problem.. Discasting...'*
Not user-friendly	***D:*** The user complains that functionality is not easily accessible in the new interface (e.g., users need to perform many actions to navigate). ***E:*** *"Your changes are awful dumped all of my stocks no longer easy to look at requires multiple clicks to see portfolio.should have left the app alone"*

these user interface issues. Hence, it is recommended that developers give more attention to how their apps look because even trivial details like icons can impact their ratings.

Finance and social apps have the highest percentage of bad updates. Table 5.8 shows the number of apps with at least one bad update, the number of updates, the

Table 5.8: The number of apps with bad updates grouped by the app category.

Category	# of apps with bad updates	# of bad updates	Total # of apps	Total # of updates	% of apps with bad updates	% of bad updates	Median # of updates per app
Finance	22	30	81	544	27%	6%	5
Social	19	27	79	1,434	24%	2%	16
Shopping	18	20	88	1,042	20%	2%	9
News and Magazines	13	19	66	644	20%	3%	6
Sports	12	13	58	439	21%	3%	5
Communication	11	11	75	1,289	15%	1%	12
Productivity	11	13	79	965	14%	1%	9
Health and Fitness	10	11	70	746	14%	1%	8
Tools	10	11	92	1,729	11%	1%	12
Photography	9	10	91	1,337	10%	1%	9
Lifestyle	8	10	58	535	14%	2%	7
Weather	8	10	66	422	12%	2%	3
Games	7	8	79	757	9%	1%	7
Travel and Local	7	7	68	730	10%	1%	8
Business	6	8	76	657	8%	1%	7
Entertainment	6	7	81	656	7%	1%	7
Education	5	5	66	690	8%	1%	7
Maps and Navigation	5	5	57	495	9%	1%	3
Personalization	5	5	76	826	7%	1%	6
House and Home	4	5	11	132	36%	4%	14
Medical	3	3	45	257	7%	1%	4
Music and Audio	3	3	76	882	4%	0%	8
Books and Reference	2	2	64	512	3%	0%	6
Food and Drink	2	2	13	229	15%	1%	17
Parenting	2	2	6	50	33%	4%	8
Auto and Vehicles	1	1	11	132	9%	1%	7
Comics	1	1	29	175	3%	1%	3
Video Players	1	1	55	619	2%	0%	7

percentage of bad updates, and the median number of updates per app in each app category. As shown in Table 5.8, finance apps (e.g., the “Citi Mobile” and the “Bank of America Mobile Banking” apps) and social apps (e.g., the “Instagram” and the “Meetup” apps) have the highest number of bad updates. One possible reason for the finance category having more bad updates than others, is that users may expect higher quality updates from large financial corporations. For example, the “Citi Mobile” app released

an update that crashed, after which users posted negative reviews such as: *"I do not understand why such a big and powerful bank has this awful app"* and *"Can't believe it is a banking app by Citi... keep giving error.. worst app ever"*.

Considering the percentage of bad updates, financial apps have the highest percentage of bad updates (27% of the apps and 6% of the updates have bad updates). We observed that while social, house and home, food and drink, communication and tools apps have the highest median number of deployed updates per app, not all of these app categories rank high when it comes to the number of apps with bad updates. Hence, we cannot conclude that the number of deployed bad updates per app is a direct consequence of the total number of updates of an app.

Bad updates are not only perceived as bad because of functional issues. Instead, crash, additional cost and user interface issues often occur in bad updates whereas at app-level these issues do not occur as often. In addition, we observed that feature removal and user interface issues have the highest negativity ratio.

5.4.2 RQ2: How do developers recover from a bad update?

Motivation: Receiving low ratings and negative reviews can be devastating for an app (Martin, 2015). Hence, it is important that in the event of a bad update, developers can recover quickly by releasing a good update. Analyzing how developers behave after a bad update can provide insights on how developers can recover from future bad updates. Therefore, we studied if, how and after how long do developers recover from a bad update.

Approach: For each bad update U_i, we studied how many updates are needed to recover from this bad update (e.g., by addressing the primary reported issue). To identify whether the primary issue was addressed, we manually conducted the following

Table 5.9: User review changes and release notes for the *"Handcent Next SMS"* app.

User review and release notes	**Date**	**Rating**
Release Notes: (Version 7.0.0) *"Next SMS new features: - New set of emoji, more fun with animated ones. - Application level changes to greatly improve the overall speed. - UI overhaul to give a more immersive and polished material design. - Redesigned pop-up window look and functionality. - Optimization for much lower power usage. - Add night mode to make it easier on the eyes."*	11-8-2016	-
User: *"This is the only app I use that consistently gets awful reviews everytime you 'improve' it. Wasted, useless heading space, distracting grey shadows around time stamp in convos, jarring electric blue box around contact pictures in convo lists, and my contact picture disappeared completely. And what is with the hideous stop sign red unread message counter I cant get rid of!? It gives me the anxiety to have my texting app scream at me about how many messages I need to read. Please, can you tell me the point?"*	11-8-2016	★☆☆☆☆
Release Notes: (Version 7.0.1) *"Less space on conversation item, display conversations. - Fixed share pic at gallery. - Fixed some force close issues. - Improve MMS. ** Please be patient if you get blank inbox when first upgrade to ,it need some minutes optimize sync messages ***"*	12-8-2016	-
User: *"First update fixed a few things. Still seeing strange stop sign red unread message counter, but better already"*	12-8-2016	★★★★☆
Release Notes: (Version 7.0.5) *"Remove top navigator bar for saving space. - Add blacklist feature display main window. - Improve function. - Add resend message feature. - Fixed background of password input for privacy box. - Fixed known issue. - Add avatar self. -Add My Theme table theme service window.'*	15-8-2016	-
User: *"First update fixed a few things. Took nearly 5 days, but it did eventually load all my convos. Still seeing strange stop sign red unread message counter, but better already."*	15-8-2016	★★★★★

Table 5.10: The number of bad updates for which we observed evidence that developers could recover from the bad update, the number of bad updates for which users were still complaining at the end of the study period and the number of bad updates for which there is not enough information to verify whether an issue was addressed.

Decision	# of bad updates	Description
Developer recovers from a bad update	105	We observed changes in user reviews mentioning that the primary issue was addressed.
Users still complain	58	We observed that users still complain about the primary issue and either (1) the release notes for the following updates do not mention any details about the issue (43 updates), (2) the release notes mention a fix for the issue but users continue to complain (9 updates) or (3) there are no further updates at the end of our study (6 updates).
There is not enough information	97	We could not verify (a) precisely when the primary issue was fixed as user reviews were changed at different times (6 updates) or (b) whether the primary issue was addressed (87 updates) as (1) there was no primary issue (18 updates) or (2) we did not find any changes in the reviews after the bad update (63 updates). We also read the release notes of the apps of these 63 updates and we still could not verify whether the primary issue was addressed as (1) the release notes for the following updates do not mention any details about the issue (52 updates) or (2) developers mentioned a fix for the issue but there were no changes in the review contents to verify whether the issue was addressed (11 updates).

steps:

Step 1: Examine changes in reviews of the bad update. As described in Section 5.2, our crawler stores changes in reviews. Hence, for all negative reviews of a bad update U_i, we manually examined the changes that users made after the bad update U_i. By tracking the changes in the prior reviews, we could figure out whether users still complained about the primary issue or whether the issue was addressed. If a user reported that the issue was addressed, we used the posting time of the updated review to identify which update addressed the issue. In total, we analyzed 12,987 review revisions out of the 81,273 negative reviews that belong to the top 250 bad updates.

Step 2: Examine the release notes. We examined the release notes of all the updates that were deployed after the bad update U_i until we observed an update that mentioned in the release notes that the primary issue was addressed.

As described in Section 5.2, the Google Play Store provides the current app data (e.g., the current review contents and the currently deployed update). To track the changes in user reviews and examine the following release notes of a bad update, we need to crawl the Google Play Store over a period of time to track these changes. Hence, as described in Section 5.2, we built our own crawler and crawled the Google Play Store for 12 months to collect the changes in user reviews and the deployed updates of an app. Table 5.9 shows an example of changes in a single user review over time for an issue that occurred in the "Handcent Next SMS" app. On August 11^{th} 2016, developers deployed update U_i (version 7.0.0) that made changes to the user interface. Users started complaining about the new user interface. On the next day, developers deployed update U_{i+1} but the update did not address all user interface complaints. Then

on August 15^{th} 2016, the developers deployed update U_{i+2} that improved the user interface and many users updated their reviews accordingly by increasing their ratings. As shown in Table 5.9 it took two updates (from version 7.0.0 to version 7.0.5) to recover from the user interface issue.

Findings: **In 42% of the studied updates, we could verify that app developers could recover from a bad update.** Table 5.10 shows the number of bad updates for which we observed evidence that developers could recover from the bad update, the number of bad updates for which users were still complaining at the end of the study period and the number of bad updates for which there is not enough information to verify whether the raised issue was addressed. To understand the impact of solving the primary issue of a bad update on the rating of an app, we calculated the difference between the negativity ratio of a bad update and its following update that addressed the primary issue. We observed that the median negativity difference is 1.8, which means that the fixing updates have less negative reviews than the bad updates. Our findings show that although we could not verify that all apps could recover from a bad update, listening to user feedback and addressing the primary issue of a bad update can lead to an improvement of the rating of an update.

For the bad updates for which we have evidence that the developers could recover, we observed that they recover the most often from bad updates where response time, crashes, network problems, and functional issues are raised. Table 5.11 shows the number and percentage of updates from which they could recover, the median number of needed updates to recover from a bad update and the median value of the difference between the negativity ratio of a bad update and the recovery update. As shown in Table 5.11, apps are most likely to recover from bad updates where response

Table 5.11: The number of bad updates that raised a certain issue type, the number of recovered updates, the median number of updates that were needed to recover from each issue type and the median negativity difference after recovering from bad updates (ranked by the number of bad updates).

Issue type	**# of bad updates** (A)	**# of recovered updates** (B)	**% of recovered updates** (B/A)	**Median # of releases to recover**	**Median difference of negativity ratio**
Functional Complaint	70	41	59%	1	1.8
Crashing	44	30	68%	1	1.5
Additional Cost	35	7	20%	4	2.5
User Interface	23	10	43%	2.5	2.7
Privacy and Ethical Issue	23	7	30%	1	1.9
Unspecified	18	0	0%	NA	NA
Feature Request	17	6	35%	2.5	1.7
Uninteresting Content	16	0	0%	NA	NA
Network Problem	10	6	60%	1	1.9
Feature Removal	7	3	43%	1	2.4
Compatibility Issue	3	1	33%	4	2.2
Response Time	2	2	100%	2	1.6
Resource Heavy	1	0	0%	NA	NA

time, crashes, network problems, and functional issues are raised (100%, 68%, 60%, and 59% respectively). For the 44 bad updates where crashes are raised, we observed 30 out of 44 updates that eventually were addressed. For the remaining 14 updates, we could not confirm whether the primary issue was addressed as the release notes for the following updates were generic (e.g., *"bug fixes"*) and no previously posted reviews were updated to confirm that the issue was addressed. We also observed that the median number of required updates to recover from crashes and functional issues is one update, which indicates that these types of issues are addressed fast.

Identifying similarity across negative reviews (such as device model or SDK version) could help developers in the identification of the issues. We observed that the release notes of 2 out of 30 crash-fixing updates mentioned that the crash occurred only on certain devices (i.e., for a certain Android version or certain device model). For example, the release notes of the "Period Tracker" app say: *"Fixes crash on notes page for devices with OS 4.0 and below."* In another example, the release notes of the "HERE WeGo - Offline Maps & GPS" app mention that: *"Fixed a crash on app start that affected some Samsung Galaxy users."* We observed that in 14 out of 97 (14%) of the posted reviews of the bad update, the reviews mentioned that the crash occurred on *Samsung Galaxy* devices (e.g., the Samsung Galaxy J5 or Galaxy Note).

The Google Play Store shows the meta-data of a review to app developers (i.e., the installed SDK version and the device model of a user who posted the review). Prior research studied the relation between the device model and the overall app rating so developers could identify which devices impact their app ratings (Noei et al., 2017; Khalid et al., 2014). Developers could benefit from analyzing the meta-data of the negative reviews to detect devices and SDK versions that have issues. For example, developers can

calculate the percentage of negative reviews that have a certain device model or SDK version to identify which devices or SDK versions are more frequently associated with the reported issues of an update.

In 16 out of 23 (70%) of the bad updates where user interface issues were raised, developers mentioned in the release notes that they made improvements to the user interface. In only 43% of the bad updates where a user interface issue was raised, we observed evidence that developers could recover from a bad update. The median number of updates to recover from user interface issues is two, which suggests that user interface issues are not addressed as fast. For example, developers of the "ATV Extreme Winter Free" app deployed two updates to recover from the user interface issue (i.e., version 7.0.5 with release notes *"Less space on conversation item and display conversations"* and version 7.0.7 with release notes *"Remove top navigator bar for saving space"*). Only for update 7.0.7 did users update their review and increased their ratings.

User feedback may force developers to reduce the added cost. In 7 out of 35 updates (20%), we found evidence that developers could recover from the complaints about the additional cost (in 6 updates developers removed the additional cost after users complained about it, and in 1 update developers provided alternative solutions for the additional cost). For example, developers of the "MARVEL Contest of Champions" app deployed an update with additional in-app purchases. Users complained about the additional cost and started writing the hashtag *#boycottmcoc* in the review comments. For example, a user says *"New update is horrible. Used to love this game! Played for over 2 years. Spent LOTS on money. Now its the worst game Ive seen. Go back to the previous setup. #boycottmcoc"*. We also observed that the campaign that was

initiated by users to boycott the app became viral as many game players started complaining about the additional cost in the app, which forced the developers to reduce the additional app cost (Eli Hodapp, 2017). In our manual analysis of the 35 updates where the additional cost issue is raised, we identified three patterns for managing user complaints about additional cost:

- **Rolling back the additional cost (10 out of 35 updates).** These developers rolled-back the additional cost, e.g., by removing annoying advertisements or offering certain features for free. Note that in only 6 out of the 10 updates in which a developer rolled back the additional cost, we observed evidence that the additional cost issues were fixed and users increased their ratings. In the remaining four cases, we did not find changes in the user reviews to verify that the issue was fixed.

- **Providing alternative solutions for the additional cost (20 out of 35 updates).** These developers offered alternative solutions for the additional cost such as (1) offering a non-free version that does not contain advertisements (2 updates) or (2) keep on improving their app by adding new features without reducing the additional cost (18 updates). We observed that in only one of the two updates related to offering a non-free version, users liked the non-free version and increased their review rating.

- **Ignoring the user complaints (5 out of 35 updates).** These developers did not reduce the additional cost. We could not identify evidence that they attempted to satisfy the users in a different way.

Crashes and functional errors have a higher recovery rate and a faster recovery speed than other complaints. It is relatively difficult to recover from user interface issues compared to other complaints. In particular, 70% of the apps tried to recover from bad updates with user interface issues, while we could find evidence for only 43% of the updates that they succeeded to do so. Our findings show that app developers should carefully consult users before changing their apps to avoid bad updates.

5.5 Implications

Studying reviews at the update-level rather than at the app-level provides a richer view of the issues of an app. The app-level analysis does not indicate how users perceive each particular update. In particular, we observed during our analysis for RQ1 that it is important to read several reviews to understand the primary issue of an update. Hence, to understand how users perceive an update, we recommend that researchers analyze the overall sentiment of an update as captured by many reviews instead of focusing on a single review or a group of reviews for unrelated updates.

App developers need to consult with their users before deploying a new update that makes changes that can make users unhappy (e.g., changing the app's user interface). We observed that 55% of the raised issues in bad updates are not due to crashes or functional issues instead they are about other aspects (e.g., reducing features, adding cost, or changing in the user interface of an app). For example, we observed that additional cost and user interface issues are the second most raised issue types in bad updates, which indicates that mitigating such issues may enable developers to reduce the probability of bad updates. Existing tools could help in generating source code or testing user interface components. For example, Moran et al. (2018a,b)

proposed an approach that facilitates the generation of mobile app source code from UI design mock-ups. However, existing tools cannot automatically identify all non-functional issues such as user interface issues (23 updates) or feature request/removal issues (24 updates) that we encountered, because of the subjective nature of these issues. Therefore, developers should not rely solely on automated testing tools.

App developers should explicitly mention fixes in the release notes of the updates following bad updates to motivate users to download the new update. We observed that developers often do not mention explicitly in the release notes whether they addressed the user-raised issues. Instead, developers use either general words (e.g., *"bug fixes"*), or they reuse the release notes of the bad update. For example, we observed only 46 out of 105 (44%) fixing updates for which developers mention explicitly that the update addresses an issue that was raised in reviews of the previous update. We measured the differences in the negativity ratio (Neg_{Diff}) as the negativity ratio of a bad update - the negativity ratio of the fixing update. We measured the Neg_{Diff} in (1) where developers mentioned explicitly that an update addresses the raised issue and (2) where developers mentioned general release notes (e.g., *"bug fixes"*). We observed that the cases where developers mentioned explicitly that they addressed the issues of the previous bad update have a higher difference in the negativity ratio (the average $Neg_{Diff} = 1.9$) than the cases where developers do not mention that the issues were addressed (the average $Neg_{Diff} = 1.7$). Hence, developers should mention in the release notes the rationale of the new release (especially if the release addresses a critical issue that was raised in the previous update).

Store owners should provide both the overall rating of an app and the rating of the latest update so users can evaluate the new update before installing it. The Google

Play Store offers only the overall app rating. The overall app rating hides useful information about the latest update, such as whether the latest update was bad or good. The Apple App Store provides rating information for each update. Recently, the Apple App Store enabled app developers to display either the rating of the latest update or the overall rating of an app (Google, 2018). Future research is necessary to investigate the impact of this option. For example, studies need to be done on whether developers tweak this option to make their app ratings look better to users.

Store owners (such as Google) should provide both the overall app rating and the rating of the latest update, so that users have the ability to decide whether to download the new update.

5.6 Analyzing Good Updates

In our study, we only focused on bad updates. To complete our analysis, we study why users perceive an update as good. To analyze good updates, we calculated the positivity ratio in a similar way as the negativity ratio, we only counted positive ratings (i.e., ratings of four or five stars (Martin, 2015)) instead of the negative ones. Table 5.12 shows the mean and five-number summary of the positivity ratio of all updates. We followed the same approach for identifying the top 100 bad updates to identify the top 100 good updates using the positivity ratio. Table 5.13 shows the number of apps, number of collected reviews and the number of collected changes in reviews of good updates.

To understand what makes users perceive an update as good, we followed the same approach of identifying what do users complain about after a bad update, focusing only on positive reviews, as follows. First, we randomly selected 100 positive reviews

Table 5.12: Mean and five-number summary of the positivity ratio of the 19,150 updates.

Metric	**Mean**	**Min.**	**1st Qu.**	**Median**	**3rd Qu.**	**Max.**
Positivity ratio	1.0	0.0	1.0	1.0	1.0	4.5

Table 5.13: Description of the top 100 good updates dataset.

Number of studied apps	82
Number of studied updates	100
Number of collected reviews	36,358
Number of collected changes in reviews	2,668

for each of the top 25 good updates. Then, we manually read the 100 positive reviews of every good update and identified the primary reason for an update being perceived as a good update. In total, we manually read 1,879 positive reviews of the top 25 good updates. Table 5.14 shows the list of the identified primary reasons of good updates. As described in Section 5.2.3, Maalej and Nabil studied several machine learning techniques that could be used to automatically classify reviews into four high-level categories: bug report, feature request, user opinion or rating (Maalej and Nabil, 2015). In our analysis of what makes users perceive an update as good, we did not use Maalej and Nabil's high-level categories as these categories are too generic for our purpose. Finally, for each of the identified reasons for good updates, we calculated the number of updates for this reason.

We observed that developing apps that provide great functionality with an easy and straightforward user interface is by far the top reason for an update to be perceived as a good one. Table 5.15 shows the number and the percentage of the reasons for the good updates. In 68% of the analyzed updates, users liked an update because it provided great functionality. In 24% of the analyzed updates, developing a straightforward and

Table 5.14: The identified reasons for an update being perceived as a good update.

Reason for good update	**Description (*D*) - Example (*E*)**
Great functionality	***D:*** The user likes that the app provides great functionality (e.g., an important feature or interesting content). ***E:*** *"Nice having schedule always available and can be updated. Keeps me organized"*
Easy interface	***D:*** The user likes that the app has a straightforward and easy interface. ***E:*** *"Very easy to use."*
Ask for improvements	***D:*** The user asks for a new feature or an improvement to the app. ***E:*** *"Wish more reminders days before event"*
Better than competitors	***D:*** The user is satisfied that the app provides better functionality than competitor apps. ***E:*** *"Works great! Far better than HBO GO!"*
The update implemented a requested feature or addressed an issue	***D:*** The user is satisfied that the recent update implemented a previously requested feature or addressed a previously reported issue. ***E:*** *"Thank you for bringing the Chromecast support. Really helps when we don't have cable in every room."*
Low cost	***D:*** The user appreciates that the app is free or inexpensive. ***E:*** *"Free weather app"*
No specific information	***D:*** The user expresses a positive experience of using an app with no specific information. ***E:*** *"Great app!"*

Table 5.15: Statistics for the reasons for good updates (ranked by the number of updates).

Reason for good update	**# of updates**	**Percentage of updates**
Great functionality	17	68%
Simple and easy	6	24%
Ask for improvements	3	12%
The update implemented a requested feature or addressed an issue	3	12%
Low cost	3	12%
Better than competitors	1	4%
No specific information	1	4%

easy to use app was the reason for an update to be perceived as a good one. This finding shows the importance of developing straightforward user interfaces and providing great functionalities.

In 12% of the analyzed updates, users asked for improvements or complained about issues in the app. That means users still post positive reviews even if the app has minor issues or the app requires improvements. Developers can benefit from the positive reviews to identify the most appreciated features by their users and focus on improving and/or maintaining these features.

5.7 Threats to Validity

5.7.1 Construct Validity

We assume throughout this chapter that reviews belong to the latest update at the time of posting the reviews. In our previous work (Hassan et al., 2017), we observed that in some cases users still complained in the next few days after the release of a fixing

update, even though that update addressed the issue. To mitigate this problem, we did not include consecutive bad updates, as we cannot confidently determine to which update a complaint belongs. We describe this problem in more detail in Section 5.2.

In Section 5.3.2, we compared the raised issues in bad updates to the raised issues in regular updates of the same apps. Our results may be limited to characteristics of the studied 94 apps with bad updates. To validate whether our observations are still valid for other apps, we compared the raised issues of bad updates to the raised issues of all other updates. We followed the same approach as in Section 5.3.2, except (1) In **Step 2:** We included all updates except the top 100 bad updates (19,050 out of 19,150 updates) and (2) In **Step 3:** We randomly selected a statistically representative sample of the negative reviews with a confidence level of 95% and a confidence interval of 5% for each update of the 19,050 updates. Then, we grouped the collected random samples together. We ended up with 1,181,974 negative reviews (out of the collected 3,424,820 negative reviews). Finally, we randomly selected a statistically representative sample of 384 reviews (out of 1,181,974 reviews) with a confidence level of 95% and a confidence interval of 5%.

Figure 5.10 shows the distribution of each issue type for all updates and bad updates of our motivational study dataset. As shown in Figure 5.10, our observations still hold. For example, the frequency of unspecified issues in all updates is almost four times higher than that in bad updates. We observed that the percentage of update-related issues in all updates is less than in regular updates (7.3% and 19.6% respectively). This difference can be explained as users of the studied apps with bad updates are more likely to mention in their reviews that the raised issues are due to the latest update than the users of other apps.

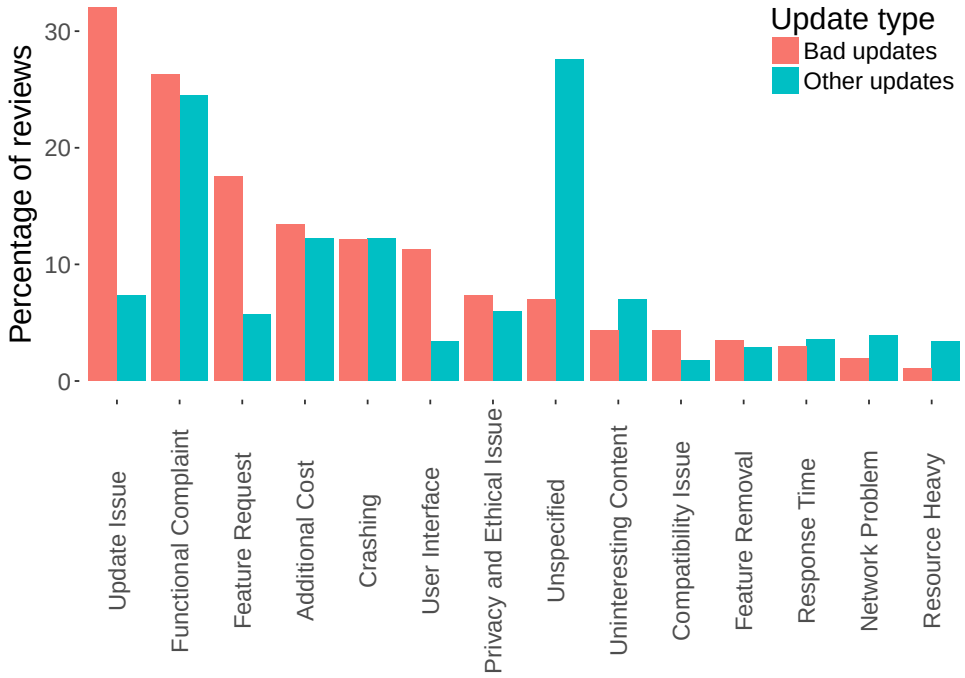

Figure 5.10: Distribution of each issue type for both all updates and bad updates of our motivational study dataset

5.7.2 Internal Validity

We collected data for the top 2,800 free popular apps in 2016 during almost one year. As described in Section 5.2, the Google Play Store provides the current app data (e.g., the current review contents and the currently deployed update). Crawling the Google Play Store once will not provide us with chronological information about the previously deployed updates and the changes in the user reviews of an app. Thus, we crawled the Google Play Store for almost one year to have enough data for identifying bad updates and analyzing why the overall user-base perceives an update as a bad update. Collecting data for a longer period and for more apps may provide more details about the characteristics of bad updates.

In our study, we analyzed the characteristics of the top 250 bad updates. We focused on the top bad updates as these updates provide good examples of unsuccessful updates. Our work performs an in-depth analysis of mobile app reviews while taking an update-centric view. Further studies can extend our work by including more than just the top bad 250 updates.

The results of our manual studies are impacted by our knowledge and experience. We are not the app owners, so our analysis may be inaccurate in some cases (especially if there was not enough data to understand user complaints). To increase the accuracy of our manual analysis, we included updates that contained at least 20 reviews. In addition, we used data from different sources (i.e., user reviews and release notes), so that, we could have a better understanding of the raised issues.

In our analysis of bad updates, we need to understand the primary issue of an update. We observed that it is important to read several reviews to understand the primary issue (rather than just a single review) because of different reasons. First, users may have different priorities, thereby making a single review extremely biased. Second, users may not report all issues in their review. Hence, to identify the primary issue of an update, we read a random sample of 100 reviews of every update to understand the overall impressions of the user-base about every update.

Recently, researchers tried to automatically label user-reviews with the corresponding issue type (e.g., crashing or bug reports) based on the review content. For example, in prior work (McIlroy et al., 2016b), we proposed an automated technique that labels reviews with the corresponding issue type with a precision 66% and a recall 65%. Later, Maalej and Nabil compared different techniques and algorithms (e.g., bag of

words and decision tree) to automatically classify reviews into four high-level categories: bug report, feature request, user opinion or rating (Maalej and Nabil, 2015). The proposed techniques by Maalej and Nabil have a higher accuracy than McIlroy et al.'s approach regarding the precision and recall (a precision ranges from 70% to 95% and recall ranges from 80% to 90%). In our study, we need a deeper understanding of the rationale behind the negative reviews. For example, for every bad update, we need to understand what are the raised issues and how developers could recover from every issue. Labeling reviews with generic high-level labels, such as bug reports or feature requests, will not provide us with insights about the nature of the raised issues and how developers addressed these issues. While manual analysis might consume more resources than using automated approaches, we decided to manually read and investigate the issues of bad updates to have more accurate results and to achieve a better understanding than we would have gotten with using automated approaches.

In our analysis, we observed that apps in the financial and social categories have the highest percentage of bad updates. This observation does not necessarily mean that apps in these categories have lower quality than apps in other categories. It might be about the passion of the user-base towards an app rather than that this app is of a lower quality. For example, users of apps in the financial category may expect higher quality updates from large financial corporations.

In our study, we observed that identifying similarity across negative reviews (such as device model or SDK version) could help developers in the identification of the issues. To analyze similarity across negative reviews, we need to determine the device model or the SDK version of a user who posted the review. The Google Play Store provides the device model or SDK version of the user that posted a review to the developer

of an app. In particular, in our collected dataset, we do not have the installed SDK version of a user who posted the review. The crawler could collect the device model for only 1.6% of the collected negative reviews. Hence, we could not perform further analysis about the similarity of the device model or the SDK version of a user who posted the review across negative reviews.

In our study, we applied our analysis of the bad updates for all app categories and the number of downloads. We analyzed the difference in the negativity ratio and positivity ratio across app categories and the number of downloads. To examine whether our analysis of bad updates should be repeated across app categories and the number of downloads. First, we applied the Scott-Knott test to group the negativity ratio of app categories into groups based on the negativity ratio (Jelihovschi et al., 2014). The Scott-Knott test is an analysis of variance test (ANOVA) that is used to validate if app categories or download ranges have statistical differences in negativity ratio. The Scott-Knott test places two distributions in different groups only if they are significantly different. The Scott-Knott test result indicated that all app categories fit into one group for the negativity ratio values. Hence, in our study, we did not need to rerun our analysis of bad updates for each app category. Second, we applied the Scott-Knott test to study the difference of the negativity ratio across the number of downloads. The Scott-Knott test results indicate that all download ranges fit into one category for the negativity ratio. Therefore, we also did not need to repeat our analysis across the different number of download ranges.

5.7.3 External Validity

The Google Play Store shows only the most recent 500 reviews per app which means that previously-posted reviews or changes in the existing reviews will not be accessible. In our study, we needed to collect as many reviews for each update as possible to understand the primary issue of a bad update. As shown in Section 5.4.2, we needed to track the changes in user reviews to identify when the raised issues in bad updates are addressed. Martin et al. (2015) discussed the sampling error in analyzing store data. To minimize the sampling error, we adjusted our crawler to visit the store many times per day. During our crawling period from April 20^{th} 2016 to April 13^{th} 2017), the crawler connected to the store 759,413 times. During our study, we found 1,284 out of 759,413 crawling cases (0.16%) in which the crawler found 500 new reviews. This means that in 99.84% of the crawling times, the crawler could collect all crawlable store data for the studied apps (i.e., we did not miss any data). Hence, we are confident about the analysis of user's complaints about a bad update and how developers recover from such bad updates as we miss a very minor amount of data (such as reviews or changes in the user reviews) for the studied apps that could impact our analysis.

5.8 Chapter Summary

Below are the key findings of our study:

1. An update-level analysis of reviews is necessary to capture the overall impressions of the user-base about a particular update. An app-level analysis is not sufficient to capture these transient impressions.

2. Bad updates are not only perceived as bad because of functional issues. Instead,

crash, additional cost and user interface issues often occur in bad updates whereas at the app-level these issues do not occur as often. We also observed that feature removal and user interface issues have the highest median negativity ratio.

3. We observed evidence that bad updates where response time, crashes, network problems, and functional issues are raised have the highest probability of their issues being addressed (100%, 68%, 60%, and 59% respectively). However, developers do not often mention in the release notes that the updates after the bad updates address the previously-reported issues. Therefore, we recommend that app developers mention in their release notes the rationale for the new update to motivate users to download the fix.

4. Uninteresting content and additional cost issues have the lowest recovery rate. Additional cost and user interface issues require the largest number of updates to recover. In addition, feature removal and user interface issues have the highest median negativity ratio. As such issues are difficult to detect automatically, app developers should consult users before releasing a new update to avoid such bad updates.

Our findings highlight the need for studying reviews at the update-level instead of at the app-level as is commonly done in literature nowadays. In the next chapter, we study the dialogue between users and developers to help design next-generation app reviewing mechanisms.

CHAPTER 6

Studying the Dialogue Between Users and Developers

THE popularity of mobile apps continues to grow over the past few years. Mobile app stores, such as the Google Play Store and Apple's App Store provide a unique user feedback mechanism to app developers through the possibility of posting app reviews. In the Google Play Store (and soon in the Apple App Store), developers are able to respond to such user feedback.

Over the past years, mobile app reviews have been studied excessively by researchers. However, much of prior work (including our own prior work) incorrectly assumes that reviews are static in nature and that users never update their reviews. In a recent study, McIlroy et al. (2017) started analyzing the dynamic nature of the review-response mechanism. McIlroy et al.'s study showed that responding to a review often has a positive effect on the rating that is given by the user to an app.

In this chapter, we revisit McIlroy et al.'s finding in more depth by studying 4.5 million reviews with 126,686 responses for 2,328 top free-to-download apps in the Google Play Store. One of the major findings of our chapter is that the assumption that reviews are static is incorrect. In particular, we find that developers and users in some cases use this response mechanism as a rudimentary user support tool, where dialogues emerge between users and developers through updated reviews and responses. Even though the messages are often simple, we find instances of as many as ten user-developer back-and-forth messages that occur via the response mechanism.

Using a mixed-effect model, we identify that the likelihood of a developer responding to a review increases as the review rating gets lower or as the review content gets longer. In addition, we identify four patterns of developers: 1) developers who primarily respond to only negative reviews, 2) developers who primarily respond to negative reviews or to reviews based on their contents, 3) developers who primarily respond to reviews which are posted shortly after the latest release of their app, and 4) developers who primarily respond to reviews which are posted long after the latest release of their app.

We perform a qualitative analysis of developer responses to understand what drives developers to respond to a review. We manually analyzed a statistically representative random sample of 347 reviews with responses for the top ten apps with the highest number of developer responses. We identify seven drivers that make a developer respond to a review, of which the most important ones are to thank the users for using the app and to ask the user for more details about the reported issue.

Our findings show that it can be worthwhile for app owners to respond to reviews, as responding may lead to an increase in the given rating. In addition, our findings

show that studying the dialogue between user and developer can provide valuable insights that can lead to improvements in the app store and user support process.

6.1 Introduction

Mobile apps continue to rapidly gain popularity over the last few years. Mobile apps can be downloaded from app stores, such as the Google Play Store, which has more than 3.1 million apps available as of July 2017 (AppBrain, 2018). These app stores allow users to express their opinion about an app through posting reviews, including a rating, that other potential users of the app see in the store.

A 2015 survey shows that 69% of the users consider the app rating as an important or very important deciding factor when downloading an app. In addition, 77% of the users will not download an app that has a rating that is lower than 3 stars (Martin, 2015). Hence, the success of an app is closely tied to the reviews and ratings that it receives.

The reviewing and rating processes have always been one-way mechanisms, as developers[1] were not allowed to respond to the reviews or ratings. As a result, issues that are raised in reviews can discourage other users from downloading the app, even though the raised issue may be inaccurate or might be easily solved.

Recently, app stores have given developers the opportunity to engage in a dialogue with the users of their apps by responding to user reviews. The Google Play Store provides guidelines for responding to a posted review (Google, 2018b). In addition, Apple's App Store is expected to add support for responding to a posted review soon (Perez, 2017). Such dialogue allows developers to provide possible solutions for the issues that

[1]Throughout this chapter, we use 'developer' to indicate the person(s) or company who are responsible for making an app.

are raised in the review, or to ask users to clarify their unsatisfaction with their app. In addition, developers can also encourage users to change their review or rating, which in turn can result in more downloads as reviews become more positive.

Prior work on mobile app reviews focuses on extracting useful information for developers from those reviews, such as bug reports or feature requests. However, prior work (including our own) falsely assumes the following: (1) app reviews and ratings are considered to be immutable, even though a user may change them over time, and (2) reviewing and rating apps is considered a one-way mechanism, even though the dynamic nature of app reviews and the response mechanism can lead to a rich dialogue between developers and users.

In this chapter, we empirically study the dynamic nature of app reviews. In particular, we study the dialogue that takes place between users and developers in 2,328 free-to-download apps in the Google Play Store. We study 126,686 dialogues that contain messages between the user and the developer. First, we conduct a preliminary study of the benefit of responding to a user review. We show that responding to a review increases the chances of users updating their given rating for an app by up to six times compared to not responding. Second, we conduct the following studies:

Study I: A study of the characteristics of user-developer dialogues

Motivation: To understand how users and developers interact via the review system in the Google Play Store, we study the characteristics of these dialogues that emerge through back-and-forth (updated) app reviews and developer responses.

Results: A user-developer dialogue was triggered by 2.8% of the user reviews. If users change the review rating after a developer response, users tend to increase

the review rating (in 4.4% of the reviews that received a response). In comparison, only 0.7% of the reviews that do not receive a response change their rating. Hence, app owners should assign more effort to responding to user reviews as a response is likely to lead to an increased rating.

Study II: A quantitative study of the likelihood of a developer responding

Motivation: In our preliminary study and Study I, we observed that responding to reviews can be beneficial for developers. Therefore, in our second study, we investigate which metrics are related to the likelihood of a developer responding to a review. The goal of Study II is to provide recommendations for store owners and developers to easier identify reviews that may require a response. For example, store owners could automatically highlight reviews that a developer is likely to respond to.

Results: When developers respond, they tend to respond to reviews that are longer and have a low rating. In addition, we find four patterns of developers: (1) developers who primarily respond to only negative reviews, (2) developers who primarily respond to negative reviews or to reviews based on their contents, (3) developers who primarily respond to reviews which are posted shortly after the latest release of their app, and (4) developers who primarily respond to reviews which are posted long after the latest release of their app.

Study III: A qualitative study of what drives a developer to respond

Motivation: In Study II we found how we can identify reviews that may require a response. However, the results obtained from Study II do not explain the actual contents of a response. Understanding the contents of developer responses

can lead to improvements in the review-response mechanism, or to improvements in the way that developers use reviews and responses. For example, if developers turn out to ask users for more details in many of their responses, such requests can be better accommodated by next-generation automated review mechanisms. Therefore, we conduct a manual study of the contents of responses to understand better what drives developers to respond to reviews.

Results: We manually examined a statistically representative random sample of 347 reviews for the top ten apps with the highest number of developer responses. We identify seven different drivers for responding, of which the most important ones are to thank the user for using the app and to ask for more details about the reported issue. In addition, we uncover interesting opportunities for developers, such as the opportunity to automatically generate frequently asked questions (FAQs) that can be published to improve the user support process.

The main contributions of this chapter are as follows:

1. Our chapter is the first work to demonstrate the dynamic nature of reviews. Much of prior work (including our own prior work) incorrectly assumes that reviews are static in nature and users never update their reviews. Our chapter shows that this is an incorrect assumption – an assumption that impacts studies in this nascent research area. For example, prior work on studying user complaints in mobile app reviews is impacted, as the static view that such work currently has of reviews cannot reflect changing user complaints.

2. Furthermore, we are the first to demonstrate a peculiar use of the app-review mechanisms as a user support medium – we do acknowledge that not as many

developers are using the review system in this way. Nevertheless, this is a use which further work needs to explore, especially given that the Apple App Store is expected to add support for responding to a posted review soon (Perez, 2017). Given the nascent nature of app reviews, it might be worthwhile that such type of use is encouraged and better supported in next-generation app-review mechanisms.

By encouraging such use, future mining studies of reviews will have access to a richer and more complete view of user concerns. Currently, in many instances developers encourage users to continue discussions via other mediums (e.g., email). Hence, the reviews do not provide an in-depth view of the user concerns (as previously thought – for instance, this observation impacts recent publications which mine reviews to gather requirements).

3. Furthermore, our work is the first work to deeply explore developer responses in a systematic manner. Our analysis is much deeper and uses considerably more data. The more in-depth analysis led us to reformulate some of our prior observations/recommendations. For instance, in comparison to our prior work (McIlroy et al., 2017), we still see that responding to user reviews may increase the rating but we find that the chances of a user updating a review based on a developer response to be quite low (even though they are up to six times higher than without a response). A deeper investigation shows that all too often, the developer responses are rather simplistic and are just asking users to raise their ratings.

4. Finally, our classification of developer-responses highlights the value of providing canned or even automated responses in next-generation app-review mechanisms. For instance, 45% of the developer-responses ask for more details. Furthermore, the ability to auto-construct FAQs based on the commonly repeated questions and answers in the user-developer dialogue is a valuable one for next-generation app-review mechanisms.

The rest of this chapter is organized as follows. Section 6.2 describes our methodology for collecting the user-developer dialogues from the Google Play Store. Section 6.3 describes a preliminary study for the user-developer interactions. Section 6.4 discusses the characteristics of user-developer dialogues in the Google Play Store. Section 6.5 and 6.6 present our quantitative and qualitative studies of what drives developers to respond. Section 6.7 discusses the implications of our studies. Section 6.8 discusses the limitations and threats to the validity of our findings. Section 6.9 discusses the related work and describes the difference between our prior work and the current work that is presented in this chapter. Finally, Section 6.10 presents our conclusion.

6.2 Data Collection

In this section, we describe our approach for collecting the user-developer dialogues from the Google Play Store. Figure 6.1 gives an overview of our approach.

6.2.1 Selecting Apps

We select the apps to study based on the following criteria:

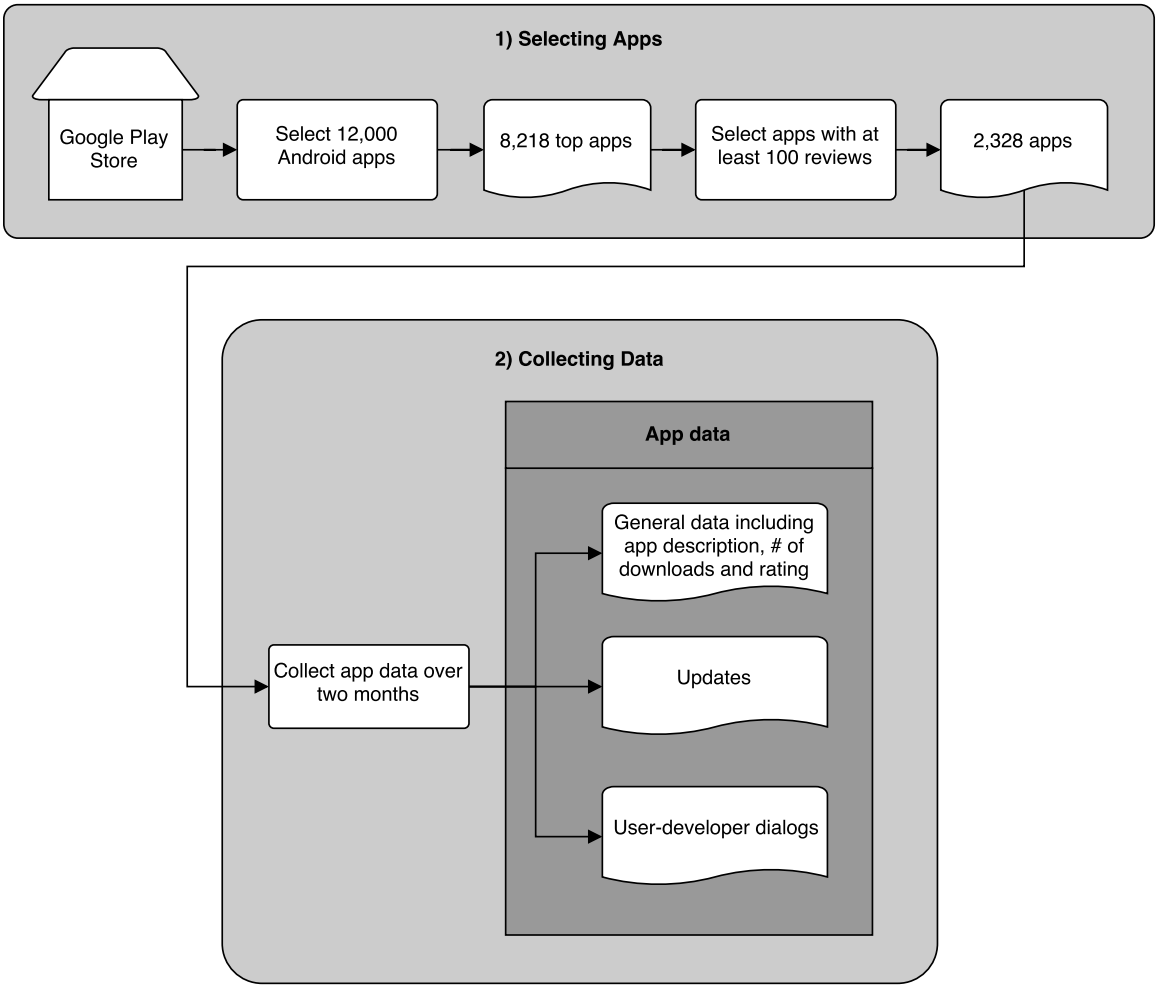

Figure 6.1: An overview of our approach for collecting user-developer dialogues

1. **App popularity**: We focus on popular apps since popular apps contain more reviews than unpopular apps (Harman et al., 2012), which should facilitate enough data for studying user-developer dialogues.

2. **App maturity**: We focus on mature apps to ensure that the apps have had enough time to gather reviews and responses.

We select the top popular 12,000 free-to-download apps in 2013 according to AppAnnie (2018) as these apps were top apps one year prior to our study. This decision

was made to ensure that the studied apps had enough reviews and that their development teams were concerned about their apps (i.e., not abandoned apps). In our study, we focus on free-to-download apps to avoid the influence of the pricing of the app. The price of a paid app is very likely to be a confounding factor, as the app price may affect the expectation of a user.

After verification, we find that 8,218 apps out of these 12,000 free-to-download apps are still available in the Google Play Store. As an additional measure for selecting mature apps only, we select only apps with at least 100 reviews. After removing all apps that have less than 100 reviews, we end up with 2,328 apps that match our selection criteria.

6.2.2 Collecting Data

We collected a new dataset for this study (i.e., we did not use the same datasets for the studies in Chapter 4 and Chapter 5) as we need to collect developer responses and to track the changes made to developer responses. We use a Google Play crawler (Akdeniz, 2013) to collect data from the Google Play Store. For each studied app, we collect the following data:

1. **General data:** app title, app description, number of downloads, app rating and app developer.

2. **Updates:** date of each update.

3. **User-developer dialogues:** review title, review text, review time, reviewer name, rating, user device model, the time it took a developer to respond and the text in the developer response.

Table 6.1: Dataset description

Number of studied apps	2,328
Number of collected reviews (A)	4,474,023
Number of collected changes in reviews	355,600
Number of collected responses (B)	126,686
Percentage of reviews with responses ($100 * \frac{B}{A}$)	2.8%

The crawler uses the Samsung S3 device model to connect to the Google Play Store, as the Samsung S3 is a popular mobile device (AppBrain, 2018). When the crawler connects to the store, the crawler collects the latest 500 reviews for each configured app.

During our study, we noticed that apps differ in the amount of new data that can be collected for an app per day. For example, the Facebook app receives thousands of new reviews per day while other apps receive only a small number of new reviews per day. To avoid flooding the Google Play Store with requests, we adapted the crawler to automatically adjust the number of times that it checks for new data to match the rate with which new review content is being posted for each studied app.

Every time the crawler connects to the Google Play Store, it checks whether there is new or updated data available for each studied app. The crawler collects the new or the updated data and appends that data to a database in which data for all apps is stored. As a result, we get a chronological overview of newly-posted reviews, ratings and responses and changes to those reviews, ratings and responses.

We run our crawler from September 22^{nd} 2015 to December 3^{rd} 2015. During that period, we collected almost 4.5 million reviews for 2,328 apps and over 355,000 changes that are made to those reviews. More than 126 thousand collected reviews have received a response from the app developer. Table 6.1 describes our dataset.

Table 6.2 shows the mean and five-number summary of the number of reviews, the

Table 6.2: Mean and five-number summary of collected data for every studied app

	Mean	**Min.**	**1st Qu.**	**Median**	**3rd Qu.**	**Max.**
Number of reviews per app (A_{app})	1,922	100	177	357	1,026	184,693
Number of reviews with responses per app (B_{app})	53	0	0	0	8	6,055
Percentage of reviews with responses per app ($100 * \frac{B_{app}}{A_{app}}$)	4.9%	0.0%	0.0%	0.0%	1.4%	98.6%

number of reviews with responses and the percentage of responses for each studied app.

6.3 Preliminary Study

Ideally, responding to a review results in either a rating increase or an updated review that contains more information about the issue that was raised by the user. In this section, we describe the results of our preliminary study to demonstrate the chances of a user updating the review or rating after a developer responded to the review.

A user-developer dialogue was triggered by 2.8% of the user reviews. As shown in Table 6.1, developers responded to 2.8% of the user reviews during the studied period. Figure 6.2 shows the distribution of the percentage of reviews that have triggered a dialogue between the user and developer, i.e., reviews that have received a developer response, for each studied app. For clarity we display only the data for the 794 apps for which the developer responded at least once to a user review. As shown by Figure 6.2, there is a broad variation in the percentage of reviews that receive a response for each studied app. The broad variation suggests that developers of different apps assign a different value to the opportunity of engaging in a dialogue with the user through the Google Play Store.

The chances of a user updating their rating after receiving a response are six

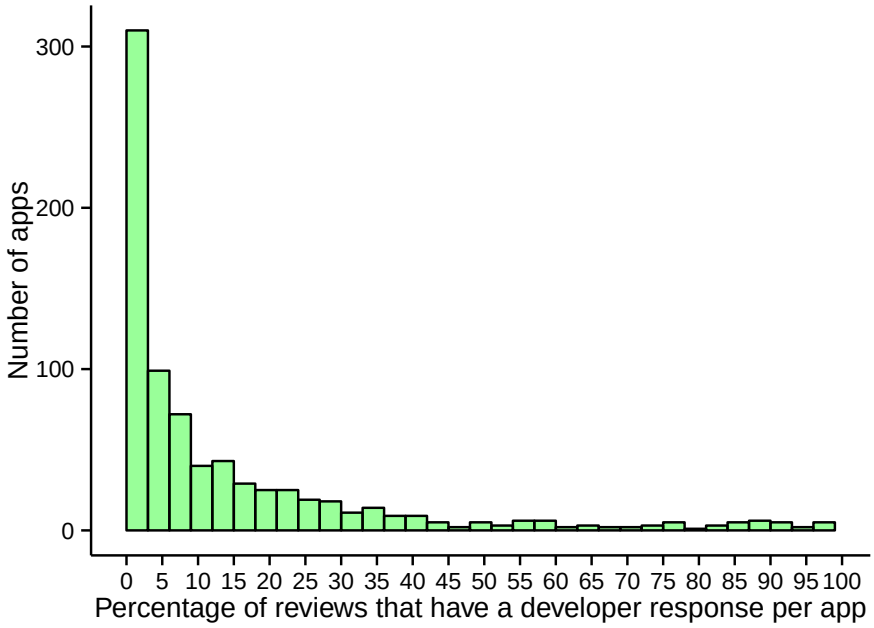

Figure 6.2: The percentage of reviews to which a developer responded for each studied app. The figure displays only the 794 apps that have at least one review to which a developer responded.

times as high compared to users who did not receive a response. Table 6.3 shows the percentage of reviews and ratings that change with and without receiving a response. Of the reviews that did not receive a response, only 0.7% increased their rating, while 4.4% of the reviews that received a response increased their rating. The percentage of decreased ratings does not change (0.7%), indicating that a response is more likely to have a positive effect on the rating. In addition, we studied the number of times a given rating both increased and decreased over time (0.0% of reviews without a response and 0.1% of reviews with a response). These rare cases are caused by a user changing the rating several times, usually over a longer period of using the app. For example, Table 6.4 shows an example dialogue in which the user increased and decreased the rating. As illustrated in Chapter 2, for the represented example in Table 6.4, a developer

Table 6.3: The percentage of reviews and ratings that change with and without receiving a response.

Reviews without a response:	Total number	% of total reviews without a response
Collected reviews	4,350,541	100%
Changed review contents	100,142	2.3%
Increased ratings	28,903	0.7%
Decreased ratings	31,344	0.7%
Both increased and decreased ratings	4,469	0.0%
Reviews with a response:	Total number	% of total reviews with a response
Collected reviews	123,556	100%
Changed review contents	9,246	7.5%
Increased ratings	5,457	4.4%
Decreased ratings	925	0.7%
Both increased and decreased ratings	81	0.1%

should receive four email notifications (i.e., an email each time the user changes in the review). In some cases, users may update their reviews several times after receiving developer responses. For example, a user may follow up to a developer response by writing few details about an issue and then the user modifies the posted review to add more details. As illustrated in Chapter 2, the Google Play Store will not notify the developer with the new modification. Thus, the developer may lose potential information that may help in solving the reported issue.

Responding to reviews positively impacts app ratings. The chances of seeing an increase in the given rating are six times as high after responding to a review.

Now that the potential of responding to reviews is clear, we study the phenomenon of the user-developer dialogue in the Google Play store in more depth in the remainder of the chapter.

Table 6.4: An excerpt from a single user-developer dialogue for the *Piano+* app. The dialogue emerged as the user and developer updated their review and response. We omitted the rest of the dialogue as the review and response updates became repetitive.

Dialogue	**State**	**Rating**
User: *"I can't play a song from the song book :("*	S_1	★★★★☆
Developer: *"Hi Brandon, we feel so sorry for any inconvenience you experienced. Please leave us a message at support@rubycell.com to let us know more clearly about your problem. If possible, show us a screenshot of the issue so that we can offer you some help. We truly look forward to your response. Many thanks. "*	S_2	-
User: *"I can't play a song from the song book :(On Samsung galaxy S4"*	S_1	★★★★☆
Developer: *"Hi Brandon, we feel so sorry for any inconvenience you experienced. Please contact us at support@rubycell.com and have a detailed discussion. We faithfully promise to help you address this problem. Hope for your response soon! "*	S_2	-
User: *"I can't play a song from the song book :(On Samsung galaxy S4... Sorry about this but I love this app and not being able to play it sucks."*	S_1	★★★★☆
Developer: *"Hi again, as we said before, we really want to receive your message at support@rubycell.com and have a detailed discussion about this problem with you. Please give us a hand to get it fixed. Many thanks!"*	S_2	-
User: *"Sorry the problem was my phone I fixed it The best piano app on here by a long shot Download this app if you haven't already you'll enjoy it!!"*	S_1	★★★★★
Developer: *"Hi Brendon, we are very glad to know that your problem was handled and hope you will find more good time with Piano+. Best wishes. "*	S_2	-
User: *"I think this version was Ok, but there are many things I don't like, like the absence of the play all button (instead the play button is play all) sometimes I just want to listen/ play to the 1st part of the song but with this update you need to listen to both, (also the play/play all button is glitchy, letting the game play the song for you is glitchy with that), also some parts of songs (for ex. MIDIs) won't display the not for that part but instead the other part will play it, AlsoNotAsPolished"*	S_1	★★★☆☆
Developer: *"Hi there, in new version, though "Play all" is not available we sure that the current "Play" button has same function. It is the matter of change in name. If you still have a problem with this button, do not hesitate to send us your email. We promise to provide you with the most suitable answer. Many thanks. "*	S_2	-

6.4 Study I: A Study of the Characteristics of User-Developer Dialogues

6.4.1 Motivation

Most prior studies of the app review mechanism in mobile app stores focus on extracting developer-relevant information from *reviews*. However, reviews form only part of the app review mechanism. As explained in the previous sections, the possibility of *responding* to such reviews leads to new opportunities for developers to increase the rating of their app and elicit more information about issues that are raised by their users. Prior work (McIlroy et al., 2017) has only touched the surface of the emerging phenomenon of developers responding to reviews. In this section, we examine the characteristics of this phenomenon in more detail.

6.4.2 Approach

We studied the length and speed of the dialogue between a user and developer, and the effect of a developer responding to a review.

To calculate the length of the dialogue, we counted the number of iterations in the user-developer dialogue during the studied period for each review. We counted a transition from state S_1 to S_2 in the state diagram in Figure 2.1 as a single iteration. Hence, the dialogue in Table 2.1 consists of 3 iterations.

To calculate the speed of the dialogue, we measured the developer response time as the time difference (in hours) between the time when a user posts a review and the time when a developer responds to the user review.

To quantify the effect of a developer responding to a review, we studied the difference in the review rating before and after receiving a developer response. We used the Wilcoxon signed-rank test (Gehan, 1965; Wilcoxon, 2018), a paired statistical test that validates if two samples have the same distribution. A p-value of less than 0.05 means that the difference between the distributions of the two samples is statistically significant and that there is a change in the review rating after a developer response.

In addition, we calculated Long et al. (2003)'s delta (d) effect size to quantify the difference in the distributions of the metrics. We used the following thresholds for interpreting d, as provided by Romano et al. (Romano et al., 2006):

$$\text{Effect size} = \begin{cases} negligible(N), & \text{if } |d| \leq 0.147. \\ small(S), & \text{if } 0.147 < |d| \leq 0.33. \\ medium(M), & \text{if } 0.33 < |d| \leq 0.474. \\ large(L), & \text{if } 0.474 < |d| \leq 1. \end{cases}$$

We conducted several manual analyses to better understand our findings. For example, we manually studied why users decrease their rating after a developer responds to their review. In all manual analyses, we went through the following process:

Step 1: We used an iterative process that is similar to the coding method (Seaman, 1999) to study the following cases:

- To understand why user-developer dialogues last for just one iteration, we read the user-developer dialogues which lasted for only one iteration and we identified why such dialogues did not continue.

- To understand why the percentage of reviews with responses is high for some

apps, we read the user-developer dialogues for the five apps whose developers responded to at least 95% of the reviews and we identified why such developers responded to almost all the reviews for their apps.

- To understand why users decrease the rating after a developer response, we read the user-developer dialogues in which the rating is decreased and we identified the most likely explanation for the decrease.

- To understand why users increase the rating after a developer response, we read the user-developer dialogues in which the rating is increased and we identified the most likely explanation for the increase.

To eliminate human bias, two researchers (including myself and a collaborator as two *coders*) conducted the manual analysis. Both coders independently read the user-developer dialogues and identified the explanations for the studied cases. When there was no exact explanation specified in the dialogue, the explanation was identified based on the experience and intuition of the coders.

Step 2: Both coders compared their results. To resolve conflicts, the coders discussed the differences to come to a consensus on the most likely explanation.

6.4.3 Results

The vast majority of user-developer dialogues last for just one iteration. Table 6.5 shows the distribution of the number of iterations within a user-developer dialogue. During our studied period, we found that user-developer dialogues can be as elaborate as up to ten iterations. However, 97.5% of the dialogues end after one iteration.

Table 6.5: The length and speed of the user-developer dialogues.

Number of reviews	Number of review-response iterations	Median response time (hours)	Median review update time (days)	Number of apps
4,350,541	0	NA	NA	2,328
123,556	1	14.7	4.1	794
2,774	2	15.9	5.4	308
255	3	12.9	4.8	78
61	4	10	4.1	35
18	5	14	2.0	15
11	6	6.1	4.6	9
5	7	3.1	3.6	5
4	8	0.4	2.0	4
1	9	0.5	24.2	1
1	10	0.2	NA	1

After manual inspection of a statistically representative random sample (with a confidence level of 95% and a confidence interval of 10%) of 96 user-developer dialogues that ended after one iteration, we find the following explanations for the large number of short dialogues:

1. Users do not always require a response from the developer (67% of the sample). For example, the developer responds simply to thank the user for downloading the app or posting a review.

2. Users are requested by the developer to continue the dialogue through another channel, such as email (29% of the sample).

3. Users do not provide the additional information that is requested by the developer response (10% of the sample).

Table 6.4 shows an excerpt of a longer dialogue between a user and app developer that we found in our study. In this dialogue, the review-response mechanism is used in two ways. First, the user is asked to provide more details through another channel (email). Second, the user is explained that the 'play all' functionality is now provided by the 'play' button. In addition, the dialogue demonstrates how a developer response can impact the user rating (i.e., a rating increase and decrease occur – the given ratings vary from three to five stars!).

Developers tend to respond faster as the number of iterations in the user-developer dialogue increases. As shown in Table 6.5, the median value for the developer response time decreases as the number of iterations in the user-developer dialogue increases. We manually analyzed the user-developer dialogues that have more than five iterations. We observed that users and developers leverage longer dialogues as a user support mechanism. During the user-developer dialogue, users either continue the discussion (e.g., follow up on the reported issue) or start a new discussion (e.g., raise a new complaint). As explained in Chapter 2, the Google Play Store notifies app developers when users follow up to a response. Table 6.5 suggests that developers give priority to responding to such notifications.

An interesting observation in Table 6.5 is that the median review update time is much longer (i.e., several days) than the median response time (i.e., several hours). Upon inspection, the longer median review update time appears to be caused by users posting new issues that they encountered after using the app for some time in the updated reviews. Next generation reviewing systems should somehow consider the changes in users' long term impressions while calculating the app rating. Moreover, store owners should notify app developers about such changes so developers can track

the changes in users' impressions about their app.

When more than 95% of the reviews of an app have a response, the responses are mostly repetitive. As shown in Table 6.2, the maximum percentage of reviews with responses for an app is 98.6%. This high percentage indicates that the developer of such apps responded to almost all reviews. To understand better why these the percentage of reviews with responses is high for some apps, we studied the five apps whose developers responded to at least 95% of the reviews. Together, these five apps responded 4,301 times to 4,431 reviews.

Two researchers (including myself and a collaborator) manually analyzed a statistically representative random sample of 353 out of the 4,301 responses with a confidence level of 95% and a confidence interval of 5%. We found that in 275 out of 353 (78%) cases in the sample, the response is generic in nature and based on a template. 55% of the responses in the sample ask the user to contact the company email in case they have issues with the app. For example, the "PrankDial - #1 Prank Call" app uses the following template response: *"Please send us a mail to support@prankdial.com if you are having any issues with the app. Thank you."* In 61% of the sample, the developer simply expresses appreciation for using the app using a template response. For example, the "Podcast Player - Free" app uses the template response: *"Thanks for the 5 stars, *user*!"*, in which **user** is replaced with the name of the user posting the review.

The value of the type of responses above for a user is questionable. On the one hand, a response shows that the developer is appreciative of the user. On the other hand, such unremarkable responses lead to more clutter in the Google Play Store. Google has set a quota of 500 responses per day per app in the Google Play Developer API (Google, 2018c), which suggests that Google is trying to prevent developers from posting a large

Table 6.6: Rating change after a developer response. The diagonal values (i.e., bold values) mean that there is no change in the user rating value. The values above the diagonal mean that the rating value is increased and values below the diagonal mean that the rating value is decreased.

	After response				
Before response	★☆☆☆☆	★★☆☆☆	★★★☆☆	★★★★☆	★★★★★
★☆☆☆☆	**48.8%**	4.6%	7.7%	11.5%	27.5%
★★☆☆☆	15.3%	**31.9%**	10.4%	14.4%	28.0%
★★★☆☆	7.1%	7.8%	**33.1%**	17.7%	34.3%
★★★★☆	3.4%	2.1%	7.2%	**43.2%**	44.1%
★★★★★	4.5%	3.3%	4.5%	3.9%	**83.7%**

number of automated responses to their reviews.

To study the value of template-based responses, we compared the percentage of rating increases and decreases after template-based and non-template-based responses. The percentage of rating decreases was similar (approximately 8.0%) across template-based and non-template-based responses. Surprisingly, the percentage of ratings that are increased after a non-template-based response is lower (42.6%) than after a template-based response (48.2%). However, after manual inspection, we observed that the template-based responses that resulted in an increased rating were responses that contained useful information for the user. For example, the response contained an email address that could be used to contact the developers, or the developers responded using a template-based solution for an issue that was raised in the review. Hence, these observations suggest that providing useful information in the response improves the chances of a rating increase, even if that information is template-based.

A developer response can have a positive impact on the given rating. We tracked the changes in the review rating during the user-developer dialogue. We observed that

in 11,813 out of the 126,686 iterations (9.3%)[2], users follow-up on the developer response. The Wilcoxon signed-rank test indicates that there is a significant change in the review rating before and after a developer response, with a small effect size.

Table 6.6 shows the percentage of the change in the user review rating after a developer responds to a review. The bold values indicate that the rating does not change. As illustrated by Table 6.6, users tend to increase the review rating (in particular, to 5-stars) if they change the review rating after a developer response. We calculated that in 5,504 reviews, the rating increases after a developer response. We found 942 reviews in which the user decreases the rating after a developer response. Two researchers (including myself and a collaborator) manually examined a statistically representative random sample of 87 of such reviews (a confidence level of 95% and a confidence interval of 10%). We identified the following reasons for decreasing a rating:

1. The user raises a new issue after a developer response, and the decreased rating is based on the severity of the new issue (31% of the sample).

2. The user has an issue and the developer responds that the issue cannot be fixed or the issue will be solved in one of the next releases (13% of the sample).

3. The user has an issue and the developer recommends a solution for the raised issue. The user complains that the provided solution does not work (15% of the sample).

4. The user has an issue and the developer responds to that issue by asking for more details. The user decreases the rating as follows:

[2]Note that this number is different from the numbers in Table 6.3, since in Table 6.3 we count the number of reviews/ratings that changed at least once during the dialogue.

(a) The user decreases the rating without providing the requested details to the developer (15% of the sample). We believe that the user was expecting the issue to be solved directly without asking for more details.

(b) The user provides the requested details and expresses their disappointment that the issue is not solved yet (23% of the sample).

We found two dialogues in which a user decreased the rating without explanation after having a positive experience with the app. A possible explanation is that those users misinterpreted the scale of the rating system (i.e., they mistakenly assumed that a 1-star rating is better than a 5-star rating).

App developers can solve 34% of the reported issues without deploying an app update. We found that in 5,504 reviews, the rating increased after a developer response. To understand why users increased the rating after a developer response, we manually investigated a statistically representative random sample of 94 of such reviews (a confidence level of 95% and a confidence interval of 10%). Table 6.7 shows the identified reasons for a rating increase. As illustrated in Table 6.7, developers could guide users to solve 34% of the reported issues without having to deploy an app update. This percentage shows that in one-third of the studied dialogues, it was relatively easy for the developer to achieve a rating increase.

The majority of user-developer dialogues last for one iteration. However, if the user decides to change the given rating during the dialogue, the rating is increased in most cases. In one-third of the studied dialogues, a rating increase was achieved by simply explaining to the user how to resolve the reported issue.

Table 6.7: The identified reasons for rating increase (ranked by the percentage of responses).

Percentage of responses	Reason for rating increase
34%	The developer guides the user to solve the reported issue without having to deploy an app update.
24%	The developer deploys an app update to address the reported issue.
13%	The details of the solution are communicated outside the store.
7%	The developer asks a satisfied user for a rating increase.
6%	The user appreciates that the developer cared by responding, even though the issue cannot be fixed.
5%	The developer indicates that the reported issue is temporary and the rating is increased when the issue is addressed.
5%	The user raises a new issue.
4%	No reason could be identified.

6.5 Study II: A Quantitative Study of the Likelihood of a Developer Responding

6.5.1 Motivation

In the previous section, we illustrated that responding to user reviews can have a positive effect on the user rating. Reading and responding to user reviews is a time-consuming process, especially if an app has a large number of user reviews. Providing a better understanding of the metrics that drive a developer to respond, can be used by app developers and store owners to facilitate a better and more efficient user-developer dialogue. For example, the identified metrics can be used by store owners to highlight reviews that are more likely to require a response, thereby helping developers to prioritize the reviews to read and respond to.

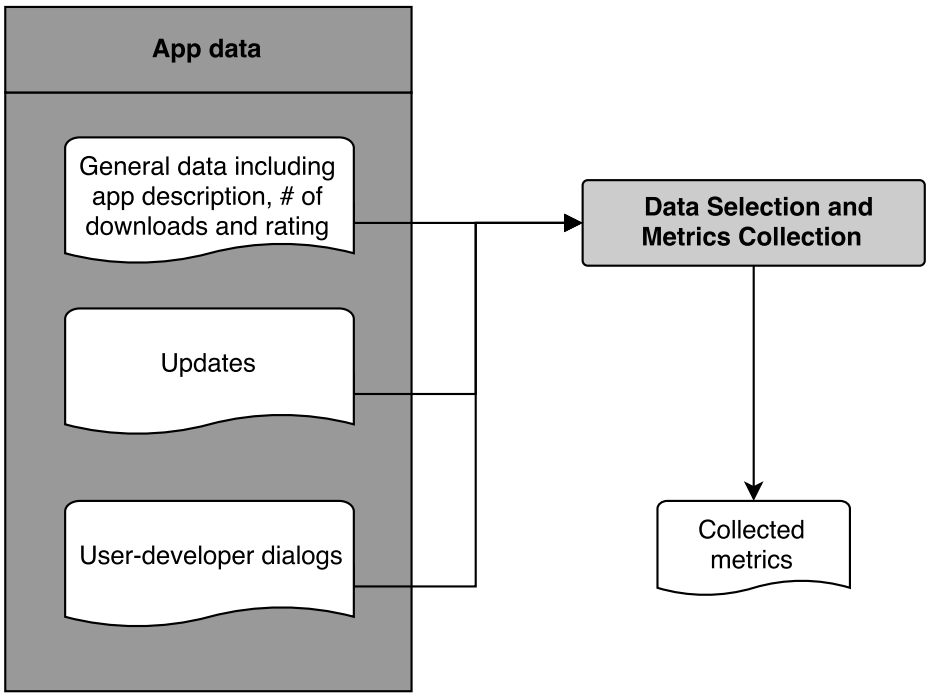

Figure 6.3: An overview of our data selection and metrics collection step.

6.5.2 Approach

In this section, we describe our approach for analyzing the relationship between the studied metrics and the likelihood of a developer responding. Figures 6.3 and 6.4 depict the steps of our approach. As shown in Figures 6.3 and 6.4, our approach contains three steps. First, we collected metrics that may have a relationship with the likelihood of a developer responding. Second, we constructed statistical models (i.e., mixed effect models) to capture the relationship between the studied metrics and the likelihood of a developer responding. Finally, we analyzed the constructed models to understand which metrics are related to the likelihood of a developer responding. In the next sections, we detail every step.

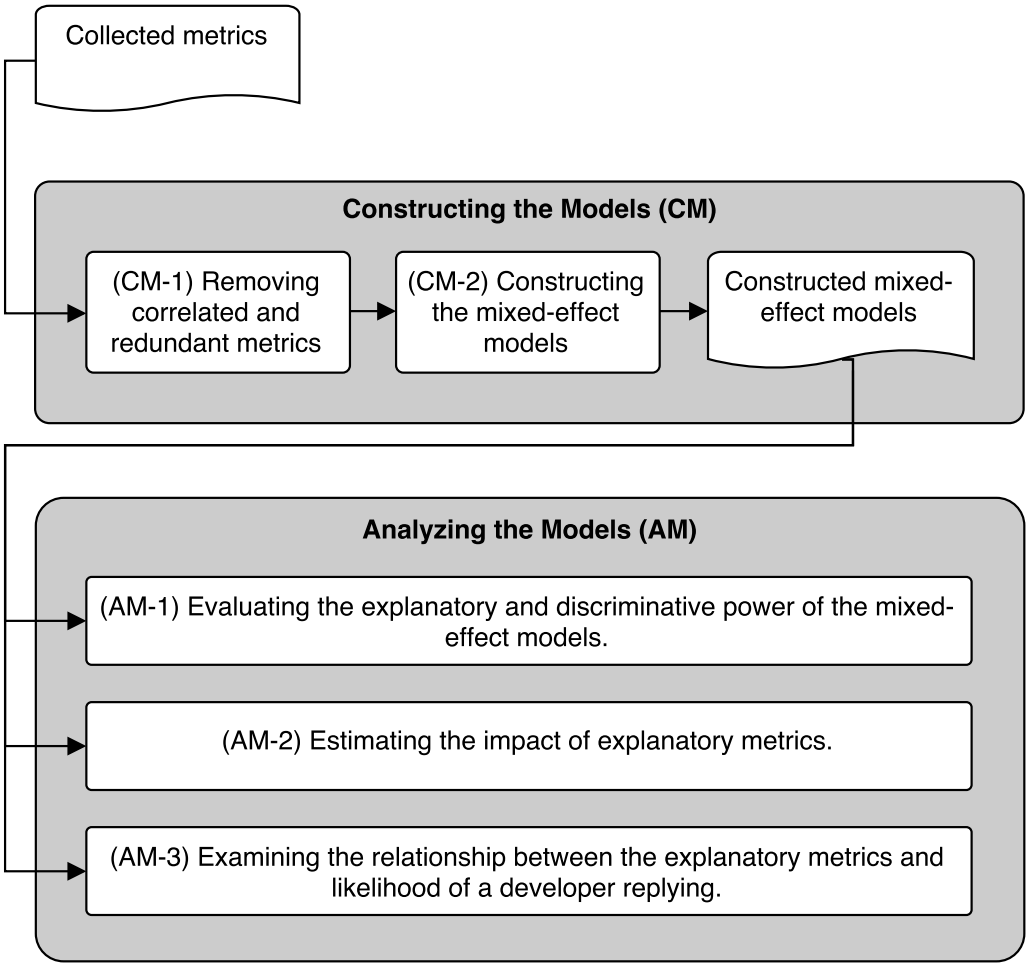

Figure 6.4: An overview of our approach for studying the relationship between the studied metrics and the likelihood of a developer responding to a user review.

Data Selection and Metrics Collection

In order to create a model to understand which metrics are related to a developer responding to a user review, we need to study apps that have sufficient user reviews and developer responses. Hence, we removed apps in which less than 5%[3] of the reviews have a response. After removing such apps, there remained 415 apps with 556,314 reviews and 103,612 responses in total. Consequently, the results that are presented in this section are valid for developers who respond to at least 5% of the posted reviews.

Table 6.8 presents the two levels of metrics that we collected in our study. app-level metrics are at the app level (e.g., the app category) while review-level metrics are at the review level (e.g., the review rating).

In the app-level metrics, we collected the app ID, app rating and app category. In the review-level metrics, we collected the review title length, review text length, days since last release, review rating and positive/negative sentiment.

We used the SentiStrength (2017) tool to measure the review sentiment metrics. SentiStrength is designed for analyzing sentiments in a short text such as online reviews (Thelwall et al., 2010). Several studies have shown that the SentiStrength tool is applicable to the software engineering domain (e.g., (Tourani et al., 2014) and (Guzman et al., 2014)). In particular, the SentiStrength tool was previously used to analyze user reviews of mobile apps (e.g., (Guzman and Maalej, 2014), (Maalej and Nabil, 2015) and (McIlroy et al., 2016b)).

SentiStrength provides for each analyzed sentence two values: (1) a positive sentiment value that ranges from 1 to 5 and (2) a negative sentiment value that ranges from -1 to -5. Tourani et al. (2014) studied the sentiment analysis in developer emails

[3]We experimented with several thresholds, as explained in Section 6.8, which resulted in very similar models.

and found that the maximum sentiment value per line can be used to represent the sentiment value for the entire email text. Hence, we followed the same approach by calculating the sentiment value per sentence and consider the maximum sentiment values across all sentences as the sentiment value for the entire review text.

For example, a user for the *"Daily Yoga - Get Fit & Relaxed"* app posts a review *"**Brilliant App Very easy to use and extremely good fun!** Would recommend it to all ages. I would down load it twice if I could!"*. The highlighted sentence has the highest positive sentiment value (i.e., a sentiment value of 5) while the other sentences have lower sentiment values (i.e., sentiment values of 1 and 2), so we considered this review to have a positive sentiment value of 5.

Constructing the Models (CM)

In order to examine the metrics that have a relationship with the likelihood of a developer responding, we performed the following steps to construct our models. Note that we are not trying to predict whether a developer will respond to a review.

(CM-1) Removing correlated and redundant metrics. We removed highly correlated review-level metrics before constructing our models to remove the risk that those correlated metrics interfere with our interpretation of the models (Shepperd et al., 2014). We measured the correlation between the review-level metrics using the Spearman rank correlation test (cut-off value for ρ is 0.7). We then used a metric clustering approach to construct a hierarchical overview of the inter-metric correlation using the implementation of the `varclus` function that is provided by the `Hmisc` `R` package (Hmisc, 2018). To detect multicollinearity, we performed a redundant analysis using the implementation of the `redun` function that is provided by the `Hmisc` `R` package (Hmisc,

Table 6.8: Collected metrics for each studied review.

Metric category	Metric	Values	Description (*D*) - Rationale (*R*)
App-level metrics	App ID	Categorical	***D:*** Identifier for the studied app. ***R:*** Including the app identifier can be used to understand how each app behaves when it comes to user-developer interactions.
	App rating	Categorical	***D:*** The app rating (e.g., 3.4). ***R:*** Apps with certain popularity (i.e., app rating value) may provide better user support than apps with different rating.
	App category	Categorical	***D:*** The category of the app (e.g., games or tools). ***R:*** Apps in certain categories may provide better user support than apps in other categories.
Review-level metrics	Review title length	Numerical	***D:*** The number of characters in the review title. ***R:*** Developers may tend to respond to issues with well-described titles.
	Review text length	Numerical	***D:*** The number of characters in the review text. ***R:*** Developers may tend to respond to issues with well-described text.
	Days since last release	Numerical	***D:*** The number of days between the review posting date and the last released update of the app. ***R:*** Users tend to post reviews in the short period after a new update (Pagano and Maalej, 2013). Hence, developers may tend to respond to reviews that are posted shortly after an update.
	Review rating	Numerical	***D:*** The star rating of a user review. ***R:*** The review rating reflects the review content (e.g., negative reviews have a low rating) (Khalid et al., 2015). Hence, developers may tend to respond to reviews with a low rating to increase the overall rating of their app.
	Positive / Negative sentiment	Numerical	***D:*** Two metrics that indicate the positive and negative sentiment values for the review content. The positive sentiment metric has values that range from 1 to 5. The negative sentiment metric has values that range from -1 to -5. ***R:*** The review sentiment can be used to indicate the purpose of the user review (e.g., user complaints have a negative sentiment) (Maalej and Nabil, 2015). Developers may tend to respond to reviews based on the review sentiment.

2018). After doing the correlation and redundancy analysis, we found that no metrics were correlated or redundant. Hence, no metrics were removed from our dataset.

(CM-2) Constructing the mixed-effect models. As explained in Section 6.5.2, the collected metrics represent two hierarchical levels (i.e., the app level and the review level). Constructing a simple regression model that ignores the hierarchical nature of the collected metrics may produce inaccurate results (Anderson et al., 2001). As the user reviews were collected from different mobile apps, the likelihood of a developer responding may vary depending on the mobile app. For example, developers of one particular app may have the policy to never respond to reviews or the metrics that are related to the response likelihood might differ between groups of apps (something that we observe later on in our analysis).

In a traditional linear regression model, all subjects have the same relation with the outcome y:

$$y = \alpha x + \beta + \epsilon \tag{6.1}$$

with the slope α and the intercept β being fixed and ϵ being the standard error. Because all subjects have the same relation with y, such a model cannot express differences for subjects at different hierarchical levels.

To study the user review characteristics while considering mobile application context, we used a mixed-effect model (Snijders and Bosker, 2012) instead. A mixed-effect model includes two types of metrics, i.e., explanatory metrics and context metrics. Explanatory metrics (review-level metrics) are used to explain the data at the user review level, while context metrics refer to the app level (i.e., app category and app ID). A mixed-effect model expresses the relationship between the outcome (i.e., a developer

responding) and the review-level metrics (e.g., review rating), while taking into consideration the different app-level metrics (e.g., the app ID).

There are two ways to construct a mixed-effect model: (1) a random intercept model and (2) a random slope and intercept model (Snijders, 2005). A random intercept model contains different intercepts (for the app-level metrics) and fixed slopes (for the review-level metrics). Such a model assumes that each app has its own baseline likelihood to respond, while the final likelihood to respond is related to a general model that is the same across all apps (i.e., a fixed slope for the review-level metrics).

A random slope and intercept model contains different intercepts (for the app-level metrics) and different slopes (for the review-level metrics). Such a model assumes that the likelihood to respond varies per app and per review-level metric for each app. In our study, we constructed both types of mixed-effect models using the `glmer` function of the `lme4` `R` package (lme4, 2018).

Analyzing the Models (AM)

After building the mixed-effect models, we evaluated the explanatory and discriminative power of the models and we estimated the impact of the review-level metrics. In addition, we examined the relationship of each metric to the likelihood of a developer responding.

(AM-1) Evaluating the explanatory and discriminative power of the mixed-effect models. Explanatory power measures the goodness-of-fit of a model (Eisenhauer, 2009). To calculate the explanatory power $expl_{\text{mixed}}$ of a mixed-effect model, we calculated:

$$expl_{\text{mixed}} = \frac{D_{\text{null}} - D_{\text{mixed}}}{D_{\text{null}}} \tag{6.2}$$

where D_{null} is the deviance (which is a goodness-of-fit measure) of the null model and D_{mixed} is the deviance of the mixed-effect model. The null model is the simplest possible mixed-effect model that contains only app-level metrics. The explanatory power explains the proportion of the data that can be explained by the review-level metrics, with 0 being the lowest proportion, and 1 being the highest proportion.

The discriminative power of a model measures the model's ability to discriminate between whether a developer will or will not respond to a review. We calculated the discriminative power using the Area Under the receiver operator characteristic Curve (AUC). The Receiver Operator Characteristic (ROC) curve plots the true positive rate against the false positive rate for different threshold settings. AUC values range between 0 (worst performance), 0.5 (performance of random guessing), and 1 (best performance) (Hanley and McNeil, 1982; StackExchange, 2015).

(AM-2) Estimating the impact of review-level metrics. To understand the most impactful metrics in our mixed-effect models, we used Wald statistics to estimate the relative contribution (χ^2) and the statistical significance (p-value) of each review-level metric in the model. The larger the χ^2 value, the larger the impact that a particular review-level metric has on the performance of the model. We used the ANOVA implementation that is provided by the `Anova` function of the `car` `R` package (Fox and Weisberg, 2018) to calculate the Wald χ^2 and the p-value of each review-level metric in the model.

(AM-3) Examining the relationship between the review-level metrics and likelihood of a developer responding. To study the relationship between the review-level metrics and the likelihood of a developer responding, we plotted the likelihood that developers will respond corresponding to the changes in each review-level metric while holding

Table 6.9: Summary of the model analysis of the mixed-effect models that we used to understand the relationship between review metrics and the likelihood of a developer responding to a review.

Model Statistics	**Random intercept model**	**Random intercept & random slope model**	
Explanatory power	0.28	0.34	
Discriminative power (AUC)	0.92	0.93	

Random Intercept Statistics

App-level Metrics	**Variance**	**Variance**	
App ID	4.4	9.4	
App rating	1.8	2.2	
App category	0.1	0.4	

Random Slope Statistics

Review-level Metrics	**Variance**	**Variance**	
Review rating	—[a]	1.6	

Review-level Metrics Statistics for the Random Intercept Model

Review-level Metrics	**Coeff ± Std. Error**	χ^2	
Review rating	-1.018 ±0.00	62,533	***
Review text length	0.003 ±0.00	2,317	***
Days difference	-0.007 ±0.00	504	***
Review title length	0.005 ±0.00	192	***
Negative sentiment	0.040 ±0.01	39	***
Positive sentiment	0.010 ±0.01	4	-

Review-level Metrics Statistics for the Random Intercept & Random Slope Model

Review-level Metrics	**Coeff ± Std. Error**	χ^2	
Review text length	0.003 ±0.00	2,278	***
Days difference	-0.006 ±0.00	457	***
Review title length	0.006 ±0.00	235	***
Negative sentiment	0.024 ±0.01	13	***
Positive sentiment	0.011 ±0.01	3	-

Statistical significance of the χ^2 test:
'***' <0.001, '-' ≤ 1

[a]There is no variance value for the review rating metric for the 'Random Intercept' model because the random intercept model has a fixed slope for the review-level metrics.

the other review-level metrics at their median values.

6.5.3 Results

Table 6.9 shows the statistics for the two mixed-effect models.

Different mobile apps have a different likelihood of a developer responding. Table 6.9 shows that the random intercept of the app ID metric varies across different apps, which indicates that apps have a different likelihood of a developer responding. As shown in Table 6.9, the random intercept of the app rating metric varies across different apps, which indicates that apps with different ratings have a different likelihood of a developer responding. Table 6.9 shows that the app category metric has the lowest influence in the mixed-effect model.

Reviews with a low rating have the highest likelihood of receiving a response. Table 6.9 displays the ANOVA results of the random intercept model. The review rating metric accounts for the largest χ^2 value in the model, indicating that review rating contributes the most to our model.

Figure 6.5 shows the relationship between the likelihood of a developer responding and each review-level metric in the random intercept model. We omit the plots for the random intercept and random slope model because the plots are very similar to Figure 6.5. As illustrated by Figure 6.5, the likelihood that a developer will respond increases as the review rating gets lower. Hence, we conclude that developers are likely to respond to reviews with a low rating.

The relation that the review rating has with the likelihood of a developer responding is different across mobile apps. Table 6.9 shows that there is a variance in the review rating coefficient (for the random slope and random intercept model) which

indicates that app developers differ in the way their likelihood of responding is related to the review rating.

Longer reviews have a higher likelihood of receiving a response. Figure 6.5 shows the likelihood of a developer responding for each review-level metric. The grey area in Figure 6.5 represents the confidence interval of 95%. As shown in Figure 6.5, the likelihood of a developer responding increases as the length of the review text increases. However, as most reviews are short, the confidence interval for the longer reviews is larger.

6.5.4 Identifying Response Patterns

Since we found that each mobile app exhibits different behavior (as the app-level metrics and the review rating slope change per app), we performed a deeper investigation to identify the common patterns for developer responses. Figure 6.6 shows an overview of our approach.

To group apps that share a behavioral pattern when it comes to responding to a user review, we constructed a logistic regression model for each app using the `glm` function that is provided by the `stats` `R` package (Stats, 2018). The logistic regression model indicates the likelihood of a developer responding based on the values of the review-level metrics.

We found that the constructed models differ in the importance of each review-level metric and the relation of each review-level metric (i.e., the review-level metric is either positively or negatively related) to the likelihood that a developer responds. Hence, we digested the generated app model by extracting the following key features from each model:

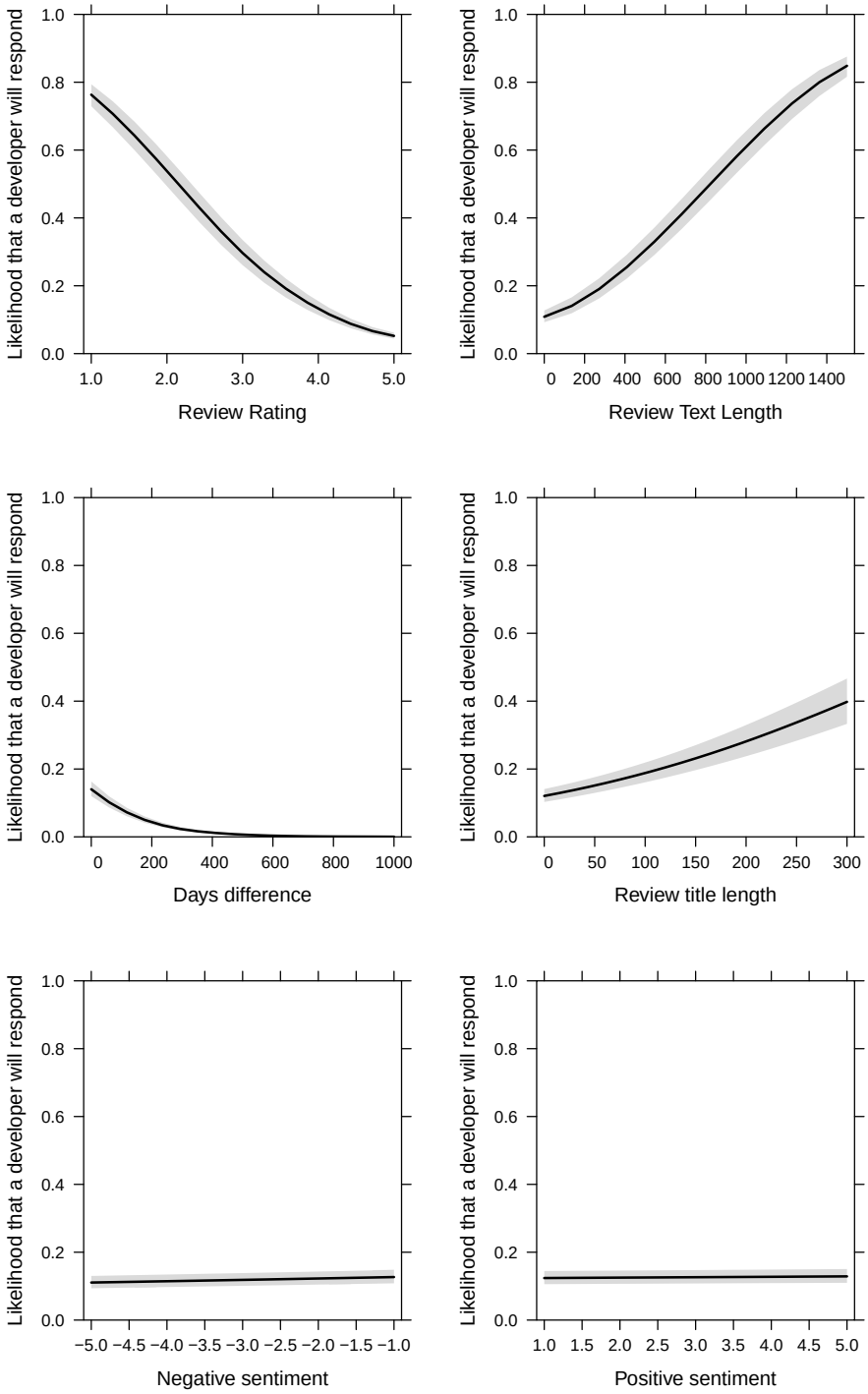

Figure 6.5: The relationship between the review-level metrics and the likelihood that a developer will respond. The grey area represents the confidence interval. Note that these plots show the likelihood that a developer will respond for apps with responses to at least 5% of the reviews.

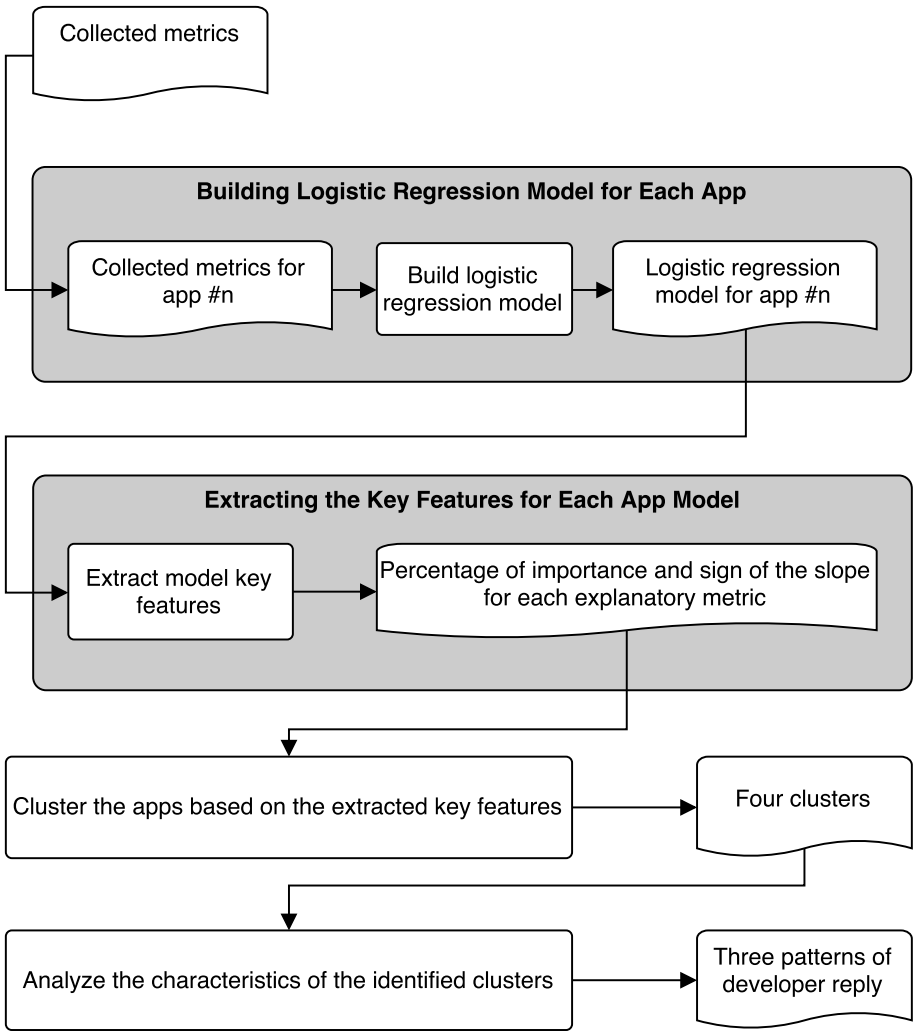

Figure 6.6: An overview of our approach for identifying the developer response patterns.

1. **The percentage of importance for each review-level metric.** To calculate the percentage of importance for each review-level metric, first we measured the power of each review-level metric. We used Wald statistics to estimate the relative contribution (χ^2) for each review-level metric in the model. We then calculated the percentage of χ^2 for each review-level metric to the total χ^2 for all review-level metrics. For example, the percentage of importance for the review

Table 6.10: The extracted key features for the generated logistic regression model for the *"MapFactor GPS Navigation Maps"* app

The review-level metric	χ^2	**Slope**	**Percentage of importance**	**Sign of the slope**
Review rating	55.32	-5.80	79.2%	Negative
Negative sentiment	6.61	-1.49	9.5%	Negative
Review title length	2.68	0.03	3.8%	Positive
Review text length	2.24	0.00	3.2%	Positive
Positive sentiment	1.99	-0.46	2.8%	Negative
Days difference	0.98	0.02	1.4%	Positive

rating in the model described by Table 6.9 is 95.6%.

2. **The sign of the slope for each review-level metric.** In order to estimate the direction of the relationship of review-level metrics for each model, we extracted the sign of the slope for each review-level metric.

Table 6.10 shows the extracted key features for the generated logistic regression model for the *"MapFactor GPS Navigation Maps"* app. We used the extracted key features from each model as input to a clustering technique. We used the clustering implementation of the `kmeans` function that is provided by the `stats R` package (Stats, 2018). In order to identify the optimal number of clusters, we started with the minimal number of clusters (two clusters) and increased that number until the newly-found clusters are only sub-clusters of the previous run. Eventually, we identify four clusters of developer response patterns using our heuristic. The four clusters can be explained as follows:

1. **Developers who primarily respond to only negative reviews.** In the first cluster

(231 apps), developers responded mainly to negative reviews (the mean value for the importance of the review rating metric is 78%).

2. **Developers who primarily respond to negative or longer reviews.** In the second cluster (95 apps), developers responded mostly to negative reviews (the mean value for the importance of the review rating metric is 40%). However, in addition to the negative reviews, developers in the second cluster appear to pay attention to the length of a review as well (the mean value for the importance of the review text length is 26%).

3. **Developers who primarily respond to reviews which are posted shortly after the latest release.** In the third cluster (50 apps), developers responded mostly to reviews which are posted soon after the latest update (the mean value for the importance of the days since last release metric is 47%).

4. **Developers who primarily respond to reviews which are posted longer after the latest release.** In the fourth cluster (39 apps), developers have a higher likelihood of responding to reviews that are posted long after the deployment of the latest update (the mean value for the importance of the days since last release metric is 42%).

The explanation for the first and second cluster is quite obvious, as negative and longer reviews are the most likely to be candidates for a rating increase. The third and fourth cluster are less intuitive. We manually read a sample review to which a developer responded for each app in the third and fourth cluster. We noticed that for both clusters developers responded shortly after the review is posted. For the third cluster, we noticed that in most cases users complain about an issue in the latest update and

developers try to solve the issue or announce that they are working on fixing the issue. For example, we noticed that in 18 apps (out of 50 apps belonging to this pattern) the median number of days between the update and a posted review is less than a week. Hence, the third cluster can be explained as developers focusing on issues that arise directly after an update.

The fourth cluster can be explained as developers who focus mostly on issues that are raised longer after an update. A possible explanation for the focus of these developers is that they focus on reviews that raise issues that need to be fixed. Hence, shortly after an update, the same obvious issues are reported by many users, yet these developers respond to only a few of them. Instead, they shift their focus to responding to issues in reviews that are raised at a later stage, which are less likely to be noticed by users, therefore taking a longer time to be reported in reviews.

6.6 Study III: A Qualitative Study of What Drives a Developer to Respond

6.6.1 Motivation

In the previous section, we gave a quantitative view of the metrics that are related to the likelihood of a developer responding to a review. These metrics can be used to identify reviews that are likely to require a response. However, these metrics do not explain the actual contents of a response, nor do they explain the rationale for a developer responding to a review. In this section, we study more qualitatively what drives developers to respond. Getting a better understanding of what drives developers to

respond to a user review can be beneficial for developers, as it allows them to identify flaws in their user support process. For example, it can help them to improve or extend the Frequently Asked Questions (FAQs) on their website. In addition, identifying the drivers for responding can assist the app store owners by revealing useful functionality for developers that can be added to their store.

6.6.2 Approach

In order to identify what drives developers to respond to a user review, we conducted a manual analysis of their responses. First, we ranked the apps by the number of responses by a developer. To avoid bias in our manual study, we examined a statistically representative random sample (with a 95% confidence level and a 5% confidence interval) of the responses for each of the 10 apps with the highest number of responses. Table 6.11 shows a summary of the selected ten apps together with the size of the selected sample. In total, we selected 3,431 responses.

For our manual analysis, we examined a statistically representative random sample of 347 responses from the selected 3,431 responses (with a 95% confidence level and a 5% confidence interval). Two researchers (including myself and a collaborator as two *coders*) went through the following process to identify the drivers that make developers respond to a user review:

Step 1: The coders independently followed a method similar to the open coding method (Khandkar, 2009; Borgatti, 1996) to identify the drivers that make developers response. The open coding method iteratively builds a list of identified drivers. For every studied developer response, the coder identifies the driver. If the driver is not in the list of identified drivers, the coder extends the list and revisits all developer responses using the

Table 6.11: Data summary of the top ten apps with the highest number of reviews with responses.

App name	Number of collected reviews with responses	Number of downloads	Average rating	Category	Number of selected responses
Clean Master (Boost & AppLock)	6,128	500M - 1,000M	4.66	Tools	362
Daily News, eBooks & Magazines	4,554	10M - 50M	4.24	News & Magazines	354
PicsArt Photo Studio	4,481	100M - 500M	4.44	Photography	354
AppLock	4,471	100M - 500M	4.35	Tools	354
Solo Launcher Clean Smooth Diy	4,187	100M - 500M	4.48	Personalization	352
Hungama Music - Songs & Videos	3,949	10M - 50M	4.11	Music & Audio	350
The PCH App	2,452	1M- 5M	4.46	Lifestyle	332
UC Browser - Fast Download	2,193	100M - 500M	4.55	Communication	327
FrostWire - Torrent Downloader	2,026	10M - 50M	3.93	Media & Video	323
WeChat	2,023	100M - 500M	4.25	Communication	323

new list of identified drivers. The open coding method terminates when there are no newer drivers to be identified and all developer responses are studied.

Using the open coding method, we can identify multiple drivers for a single developer response. For example, a user posted a review *"Good"* and the developer response is *"Dear Rama Thanks for your valuable feedback. Please revise your ratings to support us better since 3 star that you have rated is lower and 5 star is the highest."*. We tagged this response with the "Thank the user" and "Ask for endorsement" labels.

The output from this step is the list of drivers that the coders identified for each studied response.

Step 2: The first coder compared the drivers that were identified by both coders for all studied responses and marked the differences.

Step 3: The coders discussed the differences and came to a consensus about the final codes for each studied response. We found 19 developer responses that had differences in the identified drivers between the two coders. After discussion, all differences were easy to resolve.

6.6.3 Results

We identified seven drivers that make a developer response to a review. Table 6.12 shows the identified drivers for responding along with their description and examples of a developer response. Table 6.13 shows statistics about the identified drivers for responding. In particular, Table 6.13 shows for each driver the number of responses, the percentage of the total number of responses in the sample, the number of apps whose developers had that driver at least once for responding, the number of reviews that were changed after a developer response and the number of reviews that changed

Table 6.12: The identified drivers for responding.

Driver for responding	**Description (*D*) - Example (*E*)**
Advertise other products	***D:*** The developer advertises one of his other products. ***E:*** *"Hi Michael we hope you enjoy the PCH App. Please feel free to try one of our other fun and exciting opportunities to win like PCH Lotto Blast and PCH Cash Slots. If you ever need any assistance please check out our app FAQ page within the app itself."*
Ask for endorsement	***D:*** The developer asks a user to market for the apps by increasing the review rating or by sharing the good experience with others. ***E:*** *"If you like AppLock please rate 5 stars to support us and share it thanks."*
Ask for more details	***D:*** The developer asks for additional details in order to solve the user complaint. ***E:*** *"Hi Pratik Could you please let us know if this is happening all the time or is it a recent occurrence? Does it happen with a particular song or all of them? Kindly share more details about this issue and we will provide you with optimum support."*
Justify the advertisements	***D:*** The developer explains the rationale for the existing advertisements in the app. ***E:*** *"Sorry for making you trouble with ads and we also completely understand your concern. We are always try to improve our app and we want to provide our user a good and free clean master. So we hope you could understand that we need this part for our income. Thank you!"*
Provide guidance	***D:*** The developers provides guidance and additional details to the users. ***E:*** *"Please open phone settings, then select security, then select apps with usage access, enable AppLock."*
Wait for updates	***D:*** The developer notifies the user that the reported issue will be solved or the requested feature will be added and asks the user to wait for upcoming updates. ***E:*** *"Your feedback is received and will be handled as soon as possible. Stay tuned with us for updates. Thanks for your support!"*
Thank the user	***D:*** The developer thanks the user for using the app or posting a review. ***E:*** *"Hi Shani Thank you for your fantastic review!!"*
Unspecified	***D:*** The developer response is generic or the developer response is not in the English language. ***E:*** *"Dear User We hope you enjoy using the app and becomes a satisfied users of the app."*

Table 6.13: Statistics for the drivers for responding (ranked by the number of responses).

Driver	Number of responses	Percentage of responses*	Number of apps	Number of changed reviews	Number of changed rating
Thank the user	219	63%	10	8	5
Ask for more details	155	45%	9	12	6
Provide guidance	86	25%	10	7	3
Ask for endorsement	83	24%	9	3	2
Advertise other products	47	14%	3	1	0
Wait for updates	31	9%	8	2	1
Justify the advertisements	5	1%	3	0	0
Unspecified	3	1%	2	0	0

*Since a response can have several drivers, this column does not add up to 100%.

their rating after a developer response.

In 24% of the responses, app developers use the review mechanism to ask users to increase their given rating. In general, users tend to ignore these requests and do not update their rating. Table 6.13 shows that the "Ask for endorsement" label occurs in responses for nine out of ten studied apps, which indicates that this kind of request is common. Only in 3 out of 83 developer responses, users change their review after a developer asks for a rating increase and in only two cases the rating actually increases. The observation about developers asking for a rating increase is in line with our finding in Section 6.5.3, where we show that developers are more likely to respond to reviews with low ratings. These observations combined confirm (in accordance with literature (Martin, 2015)) the importance of ratings to developers.

In 45% of the developer responses, developers ask for more details about the reported issues. Users tend to post reviews that contain a complaint but do not contain enough information for the developer to address the complaint. For example, developers ask for more details about the hardware or the OS version on which an issue

occurred. Developers ask the user to provide the additional details through one of the following channels: (1) directly in the dialogue (i.e., by updating the review), (2) using the crash reporting mechanism in the app or (3) by email.

Moran et al. (2015) propose an approach for guiding users to report the steps that produce reported issues. Store owners and developers can include more detailed information about the reported issues by following Moran et al.'s approach.

In 25% of the responses, developers provide guidance to users to solve their issues. We notice that often, users encounter the same issue as others. During our manual analysis, we find eight cases of the same issue that is reported by multiple users to which the developer responses mostly the same. For each case, we manually extract keywords from the responses (e.g., 'clear cache') and search for responses containing those keywords for that app in the total set of reviews. We then manually check how many times the response occurs in the total set of reviews (not just our manually coded reviews). Table 6.14 shows the top five responses that occur the most often, together with a description of the reported problem and the given response.

As Table 6.14 shows, the repeated responses often indicate an issue that occurs for many users. Knowing about such issues can be beneficial for developers, as they can improve their app to address the issue. Another possibility is to address the issue in the Frequently Asked Questions (FAQs) section of the app to help users solve the issue without having to post a review (which often includes a low rating). For example, the "AppLock" issue that is addressed by 520 reviews can be summarized by the question *"How can I use the AppLock app to lock other apps?"* By studying the reviews and responses, developers can semi-automatically (e.g., using keywords) extract issues that should be addressed in the FAQ section.

Table 6.14: The identified similar responses.

App	Number of similar reviews	Description (*D*) - Response (*R*)
AppLock	520	***D:*** The user asks how to use the *"AppLock"* app to lock other apps. ***R:*** *"Please open phone settings security apps with usage access enable AppLock."*
PicsArt	271	***D:*** The user complains that the app is very slow. ***R:*** *"Hi Vincent this issue can sometimes be solved by clearing the cache. To do so go to your device's Settings - Apps - PicsArt and tap 'Clear data' and 'Clear cache'. If the issue persists please feel free to contact us at support@picsart.com and we will be happy to investigate the issue so you can use the app without problems. "*
AppLock	109	***D:*** The user complains that the app stops working. ***R:*** *"Please open AppLock to check whether the app is locked. Then try to enable advanced protection."*
WeChat	70	***D:*** The user complains that the app does not display or upload pictures. ***R:*** *"You can try following tips to troubleshoot your problems: 1.Download and install the latest version of WeChat. We suggest you migrate your chat history before uninstalling WeChat to prevent data loss. 2.Select any one of the chat pages and send '//switchsdcard'. If the above tips can't fix your issue contact us again."*
FrostWire	26	***D:*** The user complains that the app does not work after updating to Android Marshmallow (Android version 6). ***R:*** *"Have you just updated your operating system to Android 6? We are still working on a compatible release... lots to change. In the mean time you can go to Phone Settings and go to Apps find FrostWire then select Permissions and enable all of the permissions listed. Everything else should work just fine :)"*

Developers can benefit from the analysis of user reviews and developer responses by extracting repeated questions. These questions can help to identify issues that should either be fixed, or discussed in the Frequently Asked Questions (FAQs) section of the app.

6.7 Implications

In this section, we discuss the implications of our work for researchers and app store owners.

6.7.1 Implications for Researchers

App reviews and app ratings are not immutable. Prior work on app reviews has always assumed that reviews and ratings are posted and thereafter not changed. However, our work shows that users may change their reviews and ratings over time. Even though the portion of changed reviews (3.7% in our studied period) and ratings (1.4%) is fairly low, researchers need to be aware of the potential impact of changes in reviews and ratings on their work. Future studies are necessary to better understand the implications of changing reviews and ratings on prior research findings.

Reviews only do not provide an in-depth view of user concerns. Recently, several approaches were proposed for extracting requirements from app reviews. For example, Martin et al. (2017)'s survey shows that existing research uses reviews as a tool for requirements engineering (e.g., by extracting user complaints and feature requests). In our study, we observed that in many cases, developers ask a user for more information based on the review (in particular the final revision of a review). Hence, studying

only reviews does not give a complete overview of user concerns. Future studies on extracting requirements from app reviews should investigate how user concerns in app reviews can be combined with concerns expressed through other channels, such as email or a support forum.

Developers use the review mechanism as a user support medium. Even though not many developers use the review mechanism in this way, future studies should monitor this type of usage to identify possible improvements to the review mechanism that may better facilitate developers and users.

6.7.2 Implications for Store Owners

Developers need to resort to alternative channels, such as email, to be of full assistance to a user. We encountered in many cases developers asking users to contact them through an alternative channel, such as email. One possible explanation is that the user is forwarded to a more technical support team. Another possible explanation is that the review mechanism of the app store is not sufficient to provide adequate support. Mobile app store owners should study in depth how developers and users are using the review mechanism to find out how the mechanism can be improved.

Developers need to be notified whenever users update their reviews. As illustrated in Chapter 2, developers are not always notified when users update their reviews, which prevents developers from tracking users' changes in their impressions about the app. Moreover, missing notifications about user updates may hinder developers from noticing any additional information that is added by users to the posted reviews.

Developers can be assisted by highlighting reviews that are more likely to require

a response. We showed that, depending on the app, there can be several types of reviews identified that developers respond to. Store owners can improve the response mechanism by highlighting reviews that are longer or have a low rating. The highlighting could be done using thresholds for the length of the review text, and the number of given stars. These thresholds can be automatically defined for developers based on their response behavior, or they could be configured by the developers themselves.

Not all developers follow the published guidelines for responding to reviews. Although Google has set guidelines for responding to reviews in the Google Play store (Google, 2018b), we showed that not all developers follow these guidelines. For example, several developers thank every user for their review. Developers are discouraged from posting such responses in the guidelines as they do not contain useful information for other users. A possible solution for avoiding such responses is to automatically filter repetitive or very short responses. Another possible solution to lower the number of useless responses is to put a limit on the total number of responses that can be posted per day. With such a limit, developers need to be more selective in deciding on which reviews to respond to. Google has recently introduced a limit of 500 responses per day per app in the Google Play store API (Google, 2018c). Future studies must investigate whether a limit of 500 responses is sufficient to ban useless responses from the store.

Many users may share the same concerns about an app. Store owners can improve the way in which reviews are displayed by clustering reviews and responses based on the concerns that are expressed and addressed in them. The most often-expressed concerns (and if available, their responses/solutions!) can then be displayed on the product page of an app. Clustering reviews and responses is beneficial for both developers and users, as (1) it gives developers the chance to explain to users how to handle

a concern after downloading the app, and (2) it gives users the chance to adjust their expectations of the app.

6.8 Threats to Validity

6.8.1 Construct Validity

In our quantitative study of the metrics that are related to the response likelihood of a developer, we filtered out apps for which less than 5% of the reviews have responses. To evaluate whether our filtering has an impact on our results, we constructed our models using a threshold of 3%, 5% and 8%. We found that the most important metrics (i.e., the review rating and review text length) do not change across thresholds. Hence, we conclude that the threshold of 5% does not impact our results.

There are several approaches to understand what drives a developer to respond. For example, interviewing and surveying developers might be one way. In this chapter, we chose to instead look at the actual responses. Both approaches have their benefits and limitations. For example, with surveys developers might miss reporting on some reasons since we are depending on their recollection. Nevertheless, a mining approach has its limitation as well. For example, the collected data cannot represent all reasons for responding to user reviews. Thus, we suggest that future studies are needed to triangulate our observations through developer surveys.

6.8.2 Internal Validity

In our study, we analyzed the top free popular apps from 2013 for a two months period. Including more apps and extending the study period may provide more insights about

developer responses. While we studied considerably more data than prior work on developer responses (McIlroy et al., 2017), future studies should revisit our findings for data that is collected during a longer period (i.e., years).

We assume throughout this chapter that the posted responses are written by app developers. We were unable to find information about the team size and structure of the teams that built the top 25 apps with the highest percentage of developer responses. Clutch[4], a company that lists 2,532 mobile app development firms, shows that 1,732 out of 2,532 (68%) of these firms have less than 50 employees. Hence, while we do not have strong evidence that responses were written by actual software developers, we do have evidence that most mobile app development firms are small. Therefore, due to the small team size, it is likely that responses to reviews are posted either by app developers themselves or by other team members who are in close contact with the app developers. Future studies should investigate further the team size and structure of mobile app development teams.

The results for our manual studies are impacted by the experience of the coders and the amount of the collected data. To reduce the errors in the manual analysis, three researchers (including myself and two collaborators with at least two researchers in each manual study) were involved in the manual analysis process. In addition, while doing the manual analysis, we extracted reasons for posting reviews and responses based on the implications of their contents. Hence, while we put considerable effort to mitigate the bias, the extracted reasons may be biased by our experience and intuition.

[4] https://clutch.co/

6.8.3 External Validity

App store data changes very rapidly and the Google Play Store provides only the latest 500 reviews per app. Therefore, the crawler may miss to crawl data when the crawls are not executed as often as required. Martin et al. (2015) illustrate the sampling problem in analyzing app store data. In order to overcome this issue, we scheduled our crawler to connect to the Google Play Store several times per day to collect as many reviews as possible. In our study, we collect the data for 8,218 apps. During the study period (from September 22^{nd} 2015 to December 3^{rd} 2015), in only 307 out of 547,966 crawls (0.06%) the maximum of 500 reviews (either new or updated reviews) could be downloaded. This shows that in at least 99.94% of the crawls, we were able to collect all the new reviews and the changes that were made to the existing reviews.

We studied developer responses for reviews of free apps only. One of the main reasons for removing non-free apps is that the pricing of an app is very likely to act as a major confounding factor. Developers and users of paid apps may have different expectations and attitudes than developers and users of free apps. Hence, our findings are valid for free apps only.

In this chapter, we classify developer response patterns based on general characteristics (e.g., review rating). The characteristics of developer responses may vary based on other factors such as the company size, team structure and cultural differences. Further studies are needed to investigate the nature of user-developer dialogues based on different factors (e.g., the cultural differences). However, our documentation of the type of patterns is an essential first step for future studies.

6.9 Related Work

As shown in Section 3.4, McIlroy et al. (2017) observed that users often increase the given rating after a developer responds to the posted review. In this thesis, we revisited McIlroy et al.'s findings by conducting a more in-depth study on a larger dataset (as well tracking such user-developer dialogues over an extended period of time).

McIlroy et al. found that 13.8% of the apps respond to at least one review. In our study, we found that 794 out of 2,328 apps (34.1%) respond to at least one review. A possible explanation for this difference is that we studied apps that have more than 100 reviews only (to ensure maturity of the app). By including all still-active apps (8,218 apps) in our analysis, we found that there are 1,311 (out of 8,218) apps (16%) in which developers respond at least once to a user review. McIlroy et al. indicated that apps with a large number of reviews tend not to respond to reviews as they may be overwhelmed by the large number of reviews. In this thesis, we showed that this perception appears to be changing, since we focused on apps that have at least 100 reviews.

In addition, McIlroy et al. found that a developer response has an impact on the review rating in 38.7% of the studied cases. In this thesis, we found the number of reviews in which the rating is changed after a developer responds to be much lower (3.8%). Our findings did confirm that the rating change is often positive. A possible explanation for this difference is that we found that in many of the responses, the user is not actually being assisted directly by the developers. The large number of "Thank you" responses may influence the number of rating changes. The lack of assistance in the responses may be due to the large number of reviews that developers are dealing with, leading to (semi-)automated responses.

6.10 Chapter Summary

The Google Play Store provides a mechanism that allows users and developers to engage in a dialogue. On the one hand, users leverage this mechanism by reading, writing, and updating reviews for an app. On the other hand, developers use this mechanism to respond to the posted user reviews.

In our study, we analyzed 4.5 million reviews with 126 thousand responses for 2,328 top free popular apps in the Google Play Store. We found that users and developers leverage the user-developer dialogues as a user support mechanism. However, in most cases the dialogue between the user and the developer is short.

Below are some of the most notable findings of our study:

1. Reviews and ratings change over time by a user. Hence, researchers need to consider the dynamic nature of reviews while analyzing user reviews.

2. When the rating is changed after a developer response, it is increased in most cases. Hence, app owners should focus more on responding to reviews, especially given that in many cases reported issues can be resolved without having to deploy an app update.

3. Users and developers use the user-developer dialogue as a user support mechanism. However, in most cases, the dialogue is short.

4. In general, developers are more likely to respond to reviews with a low rating and that are longer. The more in-depth analysis led us to identify four different patterns of developers. App store owners can facilitate a better responding mechanism by automatically highlighting reviews that are likely to require a response.

5. In 45% of the responses, the user is asked to provide more details about a reported issue. App store owners can improve the review-response mechanism by automatically providing developers with more details about the raised issues.

The presented findings in this chapter show that responding to a review improves the chances of a user rating increase. However, we observed that developers do not leverage the potential of the response mechanism, as a large percentage of the responses consists of template-based responses. In particular, app owners should focus more on sending tailored responses that actually address the concerns that are raised by users. In addition, app store owners should improve their review-response mechanism as currently, developers need to ask the user to contact them via another communication channel to retrieve more details about a raised issue.

CHAPTER 7

Conclusions and Future Work

THIS chapter summarizes our work and presents the potential opportunities for further work.

App stores provide a unique updating mechanism that enables developers to publish their updates rapidly. In addition, app stores allow users to download the latest updates and post their feedback about such updates. In return, developers can respond to user reviews (i.e., feedback) through a user-developer dialogue and publish emergency updates that fix any raised issues.

In this thesis, we study the user-developer interactions through the updating and reviewing mechanisms of the Google Play Store along three perspectives: (1) studying the common developer mistakes that lead to emergency updates, (2) studying how

the reviewing mechanism can help spot good and bad updates, and (3) studying user-developer dialogues. Our research provides useful insights and recommendations for store owners to improve the existing updating and reviewing mechanisms and enhance the overall quality and user experience for the offered apps in stores.

7.1 Thesis Contributions

This thesis aims to demonstrate that studying user-developer interactions through the updating and reviewing mechanisms of app stores can help store owners improve the overall quality of the provided apps in their stores and enhance the overall experience of app users.

The main contributions of this thesis are as follows:

1. **Demonstrating the Dynamic Nature of App Reviews.** We demonstrate that the reviews are not static. Instead, users change their posted reviews to represent their current thoughts about an update. For example, users change their review to raise a new issue or to announce that their raised issue is resolved. We leverage the dynamic nature of user reviews to (1) analyze how users and developers use the reviewing mechanism as a user support medium (see Chapter 6) and (2) analyze how and after how long do developers recover from a bad update (see Chapter 5).

2. **Identifying Patterns of Emergency Updates.** We propose an approach for identifying emergency updates (i.e., updates that are published soon after the previous update). We analyze the characteristics of the top 1,000 emergency updates. We identify eight patterns of emergency updates. Our study shows that there

are commonly repeated mistakes across the analyzed emergency updates. Store owners can leverage our studies to develop mechanisms to prevent app developers from repeating such mistakes (see Chapter 4).

3. **Demonstrating the Importance of Update-Level Analysis of Reviews.** We show the importance of analyzing review at the update-level over the traditional app-level analysis of user reviews. In particular, we demonstrate that the traditional app-centric analysis of user reviews is not able to understand how the user-base perceive an update as a bad update. Our work performs an in-depth analysis of mobile app reviews through an update-centric view. Store owners could benefit from our proposed approach by studying the reviews of each update to analyze how the user-perceived quality of an app changes over time (see Chapter 5).

4. **Dynamically Spotting Good and Bad Updates.** We propose an approach for spotting good and bad updates. We use our approach to identify the top 250 bad updates. Our analysis of bad updates shows that bad updates are not only perceived as bad because of functional issues. Instead, crash, additional cost and user interface issues often occur in bad updates whereas at the app-level these issues do not occur as often. We also observed that feature removal and user interface issues have the highest median negativity ratio. Store owners can leverage our approach to spot good and bad updates and proactively limit the distribution of bad updates (see Chapter 5).

7.2 Further Work

In this section, we explore the potential opportunities for improving our work.

7.2.1 Expanding and Revisiting Prior Work Using Update-Level Analysis

In this thesis, we demonstrate the necessity of an update-level analysis of reviews to capture the impressions of an app's user-base about a particular update. Researchers could leverage our work by expanding and revisiting the existing reviews analysis studies using the update-level analysis. For example, prior work proposed techniques that can be used to identify reviews with useful information such as bug reports or feature requests. Researchers could leverage update-level analysis to study (1) how the reported bugs or requested features evolve over updates of an app, (2) how these raised issues or requested features can impact the rank of an app within its peer competitor apps. In addition, prior research analyzed the characteristics of successful apps and tried to predict whether an app will be perceived as a successful app. With the insights that are provided by our work, researchers can extract the key features of every update and build models that predict whether an update will be perceived as a good update or as a bad update. In addition, researchers could identify which artifacts that need to be changed in order to make the next update perceived as a good or bad update.

7.2.2 Helping Design the Next-Generation App Updating and Reviewing Mechanisms

In this thesis, we perform an in-depth analysis of user-developer interactions through the updating and reviewing mechanisms of app stores. Our work can help store owners design the next-generation updating and reviewing mechanisms, as follows:

- Store owners can leverage our user-developer dialogue work while designing the

next-generation reviewing mechanisms. For example, the next-generation reviewing mechanisms can show (1) the time that it takes for developers to respond to user reviews and (2) the impact of developers' response on app rating, which can encourage app developers to provide fast and helpful responses to users. In addition, store owners can leverage the user-developer dialogues to automatically generate the FAQs of an app.

- Store owners can benefit from our update-level analysis while designing the next-generation reviewing mechanism to show (1) rating details of every update, (2) the most disliked features in every update, and (3) the most useful features in every update. Hence, users can make a better decision as of whether to install the recent update of an app. In addition, app developers could benefit from the update-level analysis to understand how users perceive their updates and rapidly react to any raised issues.

- Store owners can benefit from our emergency updates analysis while designing the next-generation updating mechanism to continuously identify the commonly repeated mistakes that lead to emergency updates. They can also develop mechanisms to prevent app developers from repeating such mistakes. In addition, we demonstrate how the reviewing mechanism can help spot good and bad updates, so store owners can design the next-generation updating mechanism to leverage user reviews to proactively limit the distribution of bad updates.

Bibliography

(2013). Structure of an Android app. http://sofia.cs.vt.edu/sofia-2114/book/chapter2.html. (Last accessed August 2018).

(2018). Patch Tuesday. http://en.wikipedia.org/wiki/Patch_Tuesday. (Last accessed August 2018).

ABI Research (2013). Android will account for 58% of smartphone app downloads in 2013, with iOS commanding a market share of 75% in tablet apps. https://www.abiresearch.com/press/android-will-account-for-58-of-smartphone-app-down/. (Last accessed: August 2018).

Akdeniz (2013). Google play crawler. https://github.com/Akdeniz/google-play-crawler. (Last accessed: August 2018).

Anderson, D. R., Burnham, K. P., Gould, W. R., and Cherry, S. (2001). Concerns about finding effects that are actually spurious. *Wildlife Society Bulletin*, pages 311–316.

Apktool (2018). Apktool. http://ibotpeaches.github.io/Apktool/. (Last accessed August 2018).

AppAnnie (2018). App Annie. https://www.appannie.com/. (Last accessed August 2018).

AppBrain (2018). Free versus paid Android apps. http://www.appbrain.com/stats/free-and-paid-android-applications. (Last accessed: August 2018).

AppBrain (2018). Top Android phones. http://www.appbrain.com/stats/top-android-phones. (Last accessed August 2018).

Apple (2018a). App store improvements. https://developer.apple.com/support/app-store-improvements/. (Last accessed: August 2018).

Apple (2018b). iOS human interface guidelines: iOS design themes. https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/ColorImagesText.html. (Last accessed August 2018).

Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 712–721.

Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A. (2014). Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 588–598.

Bavota, G., Vásquez, M. L., Bernal-Cárdenas, C. E., Penta, M. D., Oliveto, R., and Poshyvanyk, D. (2015). The impact of API change- and fault-proneness on the user ratings of Android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407.

Borgatti, S. (1996). Introduction to grounded theory. http://www.analytictech.com/mb870/introtogt.htm. (Last accessed August 2018).

CFR (2018). CFR - another java decompiler. http://www.benf.org/other/cfr/. (Last accessed August 2018).

Chen, N., Lin, J., Hoi, S. C. H., Xiao, X., and Zhang, B. (2014). AR-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 767–778.

Coblis (2011). Coblis - color blindness simulator. http://www.color-blindness.com/coblis-color-blindness-simulator/. (Last accessed August 2018).

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46.

Color-Oracle (2018). Color oracle, design for the color impaired. http://colororacle.org. (Last accessed August 2018).

dex2jar (2016). dex2jar download. http://sourceforge.net/projects/dex2jar/. (Last accessed August 2018).

Eisenhauer, J. G. (2009). Explanatory power and statistical significance. *Teaching Statistics*, 31(2):42–46.

Eli Hodapp (2017). Players upset about recent 'Marvel Contest of Champions' changes organize "#BoycottMCOC Movement" . http://toucharcade.com/2017/03/09/players-upset-about-recent-marvel-contest-of-champions-changes. (Last accessed August 2018).

F-Droid (2018). F-Droid. http://f-droid.org/. (Last accessed August 2018).

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. A. (2011). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638.

FindBugs (2015). Findbugs™- find bugs in java programs. http://findbugs.sourceforge.net. (Last accessed August 2018).

Fox, J. and Weisberg, S. (2018). Package 'car'. https://cran.r-project.org/web/packages/car/car.pdf. (Last accessed August 2018).

Gao, C., Zeng, J., Lyu, M. R., and King, I. (2018). Online app review analysis for identifying emerging issues. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18.

Gehan, E. A. (1965). A generalized Wilcoxon test for comparing arbitrarily singly-censored samples. *Biometrika*, 52(1-2):203–223.

Google (2018a). App manifest, Android developers. http://developer.android.com/guide/topics/manifest/manifest-intro.html. (Last accessed August 2018).

Google (2018b). Google my business help - read and reply to reviews. https://

support.google.com/business/answer/3474050?hl=en. (Last accessed August 2018).

Google (2018c). Google Play Developer API - Reply to Reviews. https://developers.google.com/android-publisher/reply-to-reviews. (Last accessed: August 2018).

Google (2018d). Managing projects overview - Android developers. https://developer.android.com/tools/projects/index.html. (Last accessed August 2018).

Google (2018e). Permission, Android developers. http://developer.android.com/guide/topics/manifest/permission-element.html. (Last accessed August 2018).

Google (2018). Ratings, Reviews, and Responses. https://developer.apple.com/app-store/ratings-and-reviews/. (Last accessed August 2018).

Google (2018a). Update your apps - developer console help. https://support.google.com/googleplay/android-developer/answer/113476?hl=en. (Last accessed August 2018).

Google (2018b). Uses-feature, Android developers. http://developer.android.com/guide/topics/manifest/uses-feature-element.html. (Last accessed August 2018).

Google (2018c). Uses-sdk, Android developers. http://developer.android.com/guide/topics/manifest/uses-sdk-element.html. (Last accessed August 2018).

Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. (2014). Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 1025–1035.

Guana, V., Rocha, F., Hindle, A., and Stroulia, E. (2012). Do the stars align? Multidimensional analysis of Android's layered architecture. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, MSR '12, pages 124–127.

Guzman, E., Azócar, D., and Li, Y. (2014). Sentiment analysis of commit comments in GitHub: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 352–355.

Guzman, E. and Maalej, W. (2014). How do users like this feature? A fine grained sentiment analysis of app reviews. In *Proceedings of the 22nd International Requirements Engineering Conference*, RE '14, pages 153–162.

Han, D., Zhang, C., Fan, X., Hindle, A., Wong, K., and Stroulia, E. (2012). Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE '12, pages 83–92.

Hanley, J. A. and McNeil, B. J. (1982). The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143(1):29–36.

Hannah Alvarez (2014). A guide to color, ux, and conversion rates. http://www.usertesting.com/blog/2014/12/02/color-ux-conversion-rates/. (Last accessed August 2018).

Harman, M., Jia, Y., and Zhang, Y. (2012). App store mining and analysis: MSR for app

stores. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, MSR '12, pages 108–111.

Hassan, A. E. and Holt, R. C. (2004). Predicting change propagation in software systems. In *Proceedings of the 20th International Conference on Software Maintenance*, ICSM '04, pages 284–293.

Hassan, S., Shang, W., and Hassan, A. E. (2017). An empirical study of emergency updates for top Android mobile apps. *Empirical Software Engineering*, 22(1):505–546.

Hemmati, H., Fang, Z., and Mäntylä, M. V. (2015). Prioritizing manual test cases in traditional and rapid release environments. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation*, ICST '15, pages 1–10.

Hendysoft (2013). Smali2java. http://www.hensence.com/en/smali2java/. (Last accessed August 2018).

Hmisc (2018). Package 'hmisc'. https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf. (Last accessed August 2018).

Hu, H., Bezemer, C.-P., and Hassan, A. E. (2018a). Studying the consistency of star ratings and the complaints in 1 & 2-star user reviews for top free cross-platform Android and iOS apps. *Empirical Software Engineering*.

Hu, H., Wang, S., Bezemer, C.-P., and Hassan, A. E. (2018b). Studying the consistency of star ratings and reviews of popular free hybrid Android and iOS apps. *Empirical Software Engineering*.

Hyzap (2018). Hyzap. https://www.heyzap.com. (Last accessed August 2018).

Iacob, C. and Harrison, R. (2013). 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.

Iacob, C., Harrison, R., and Faily, S. (2013a). Online reviews as first class artifacts in mobile app development. In *Proceedings of the 5th International Conference on Mobile Computing, Applications, and Services*, MobiCASE '13, pages 47–53.

Iacob, C., Veerappa, V., and Harrison, R. (2013b). 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, page 29.

Icons Mind (2017). Which color is right for your mobile app icon. https://www.iconsmind.com/color-right-mobile-app-icon/. (Last accessed August 2018).

Jelihovschi, E. G., Faria, J. C., and Allaman, I. B. (2014). ScottKnott: A package for performing the Scott-Knott clustering algorithm in R. *Trends in Applied and Computational Mathematics*, 15(1):3–17.

Joorabchi, M. E., Mesbah, A., and Kruchten, P. (2013). Real challenges in mobile app development. In *Proceedings of the 7th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '13, pages 15–24.

Keertipati, S., Savarimuthu, B. T. R., and Licorish, S. A. (2016). Approaches for prioritizing feature improvements extracted from app reviews. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 33:1–33:6.

Khalid, H. (2013). On identifying user complaints of iOS apps. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 1474–1476.

Khalid, H., Nagappan, M., Shihab, E., and Hassan, A. E. (2014). 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 '14, pages 610–620.

Khalid, H., Shihab, E., Nagappan, M., and Hassan, A. E. (2015). What do mobile app users complain about? *IEEE Software*, 32(3):70–77.

Khandkar, S. H. (2009). Open coding. http://pages.cpsc.ucalgary.ca/~saul/wiki/uploads/CPSC681/open-coding.pdf. (Last accessed August 2018).

Khomh, F., Adams, B., Dhaliwal, T., and Zou, Y. (2015). Understanding the impact of rapid releases on software quality - the case of Firefox. *Empirical Software Engineering*, 20(2):336–373.

Khomh, F., Dhaliwal, T., Zou, Y., and Adams, B. (2012). Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, MSR '12, pages 179–188.

Ling (2008). Tutorial: Pearson's Chi-square test for independence. http://www.ling.upenn.edu/~clight/chisquared.htm. (Last accessed August 2018).

lme4 (2018). Package 'lme4'. https://cran.r-project.org/web/packages/lme4/lme4.pdf. (Last accessed August 2018).

Long, J. D., Feng, D., and Cliff, N. (2003). *Ordinal Analysis of Behavioral Data*. John Wiley & Sons, Inc.

Maalej, W. and Nabil, H. (2015). Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Proceedings of the 23rd International Requirements Engineering Conference*, RE '15, pages 116–125.

Mäntylä, M., Khomh, F., Adams, B., Engström, E., and Petersen, K. (2013). On rapid releases and software testing. In *Proceedings of the 29th International Conference on Software Maintenance*, ICSM '13, pages 20–29.

Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., and Petersen, K. (2015). On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425.

Martin, P. (2015). 77% will not download a retail app rated lower than 3 stars. https://blog.testmunk.com/77-will-not-download-a-retail-app-rated-lower-than-3-stars/. (Last accessed August 2018).

Martin, W. (2016). Causal impact for app store analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 659–661.

Martin, W., Harman, M., Jia, Y., Sarro, F., and Zhang, Y. (2015). The app sampling problem for app store mining. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 123–133.

Martin, W., Sarro, F., and Harman, M. (2016). Causal impact analysis for app releases in Google Play. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 435–446.

Martin, W., Sarro, F., Jia, Y., Zhang, Y., and Harman, M. (2017). A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9):817–847.

McIlroy, S., Ali, N., and Hassan, A. E. (2016a). Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering*, 21(3):1346–1370.

McIlroy, S., Ali, N., Khalid, H., and Hassan, A. E. (2016b). Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering*, 21(3):1067–1106.

McIlroy, S., Shang, W., Ali, N., and Hassan, A. E. (2017). Is it worth responding to reviews? studying the top free apps in Google Play. *IEEE Software*, 34(3):64–71.

Miller, C. (2017). Apple ramping up app store cleaning efforts, has already removed 'hundreds of thousands' of apps. https://9to5mac.com/2017/06/21/apple-ramping-up-app-store-cleaning-efforts-has-already-removed-hundreds-of-thousands-of-apps/. (Last accessed: August 2018).

mobiThinking (2013). Global mobile statistics 2013 section e: Mobile apps, app stores, pricing and failure rates. http://mobiforge.com/research-analysis/global-mobile-statistics-2013-section-e-mobile-apps-app-stores-pricing-and-failu
mT. (Last accessed: August 2018).

Moran, K., Bernal-Cárdenas, C., Curcio, M., Bonett, R., and Poshyvanyk, D. (2018a). Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*.

Moran, K., Li, B., Bernal-Cárdenas, C., Jelf, D., and Poshyvanyk, D. (2018b). Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18.

Moran, K., Vásquez, M. L., Bernal-Cárdenas, C., and Poshyvanyk, D. (2015). Auto-completing bug reports for Android applications. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pages 673–686.

Neil Patel (2017). How to use the psychology of color to increase website conversions. https://blog.kissmetrics.com/psychology-of-color-and-conversions/. (Last accessed August 2018).

Noei, E., Syer, M. D., Zou, Y., Hassan, A. E., and Keivanloo, I. (2017). A study of the relation of mobile device attributes with the user-perceived quality of Android apps. *Empirical Software Engineering*, 22(6):3088–3116.

Oh, J., Kim, D., Lee, U., Lee, J., and Song, J. (2013). Facilitating developer-user interactions with mobile app review digests. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1809–1814.

Pagano, D. and Maalej, W. (2013). User feedback in the appstore: An empirical study. In *Proceedings of the 21st International Requirements Engineering Conference*, RE '13, pages 125–134.

Palomba, F., Salza, P., Ciurumelea, A., Panichella, S., Gall, H. C., Ferrucci, F., and Lucia, A. D. (2017). Recommending and localizing change requests for mobile apps based

on user reviews. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 106–117.

Palomba, F., Vásquez, M. L., Bavota, G., Oliveto, R., Penta, M. D., Poshyvanyk, D., and Lucia, A. D. (2015). User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 291–300.

Palomba, F., Vásquez, M. L., Bavota, G., Oliveto, R., Penta, M. D., Poshyvanyk, D., and Lucia, A. D. (2018). Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software*, 137:143–162.

Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. (2013). WHYPER: towards automating risk assessment of mobile applications. In *Proceedings of the 22th USENIX Security Symposium*, pages 527–542.

Panichella, S., Sorbo, A. D., Guzman, E., Visaggio, C. A., Canfora, G., and Gall, H. C. (2015). How can I improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 281–290.

Panichella, S., Sorbo, A. D., Guzman, E., Visaggio, C. A., Canfora, G., and Gall, H. C. (2016). ARdoc: app reviews development oriented classifier. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 1023–1027.

Pathak, A., Hu, Y. C., and Zhang, M. (2011). Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HOTNETS '11, page 5.

Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. (2012). What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280.

Perez, S. (2017). Apple will finally let developers respond to App Store reviews. https://techcrunch.com/2017/01/24/apple-will-finally-let-developers-respond-to-app-store-reviews/. (Last accessed: August 2018).

Pete Houston (2011). Store and use files in assets. https://xjaphx.wordpress.com/2011/10/02/store-and-use-files-in-assets/. (Last accessed August 2018).

Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. (2012). Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 107–120.

Robotium (2016). Robotium. https://code.google.com/p/robotium/. (Last accessed August 2018).

Romano, J., Kromrey, J. D., Coraggio, J., Skowronek, J., and Devine, L. (2006). Exploring methods for evaluating group differences on the NSSE and other surveys: Are the

t-test and Cohen's d indices the most appropriate choices. In *Annual meeting of the Southern Association for Institutional Research*.

RTutorial (2018). Chi-squared test of independence. http://www.r-tutor.com/elementary-statistics/goodness-fit/chi-squared-test-independence. (Last accessed August 2018).

Ruiz, I. J. M., Adams, B., Nagappan, M., Dienst, S., Berger, T., and Hassan, A. E. (2014). A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2):78–86.

Ruiz, I. J. M., Nagappan, M., Adams, B., Berger, T., Dienst, S., and Hassan, A. E. (2016). Examining the rating system used in mobile-app stores. *IEEE Software*, 33(6):86–92.

Ruiz, I. J. M., Nagappan, M., Adams, B., and Hassan, A. E. (2012). Understanding reuse in the Android market. In *Proceedings of the 20th IEEE International Conference on Program Comprehension*, ICPC '12, pages 113–122.

Scalabrino, S., Bavota, G., Russo, B., Oliveto, R., and Penta, M. D. (2017). Listening to the crowd for the release planning of mobile apps. *IEEE Transactions on Software Engineering*, pages 1–1.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572.

SentiStrength (2017). SentiStrength. http://sentistrength.wlv.ac.uk. (Last accessed August 2018).

Shepperd, M. J., Bowes, D., and Hall, T. (2014). Researcher bias: The use of machine

learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616.

Simon Vig Therkildsen (2012). What API level should I target? http://simonvt.net/2012/02/07/what-api-level-should-i-target/. (Last accessed August 2018).

Snijders, T. A. and Bosker, R. J. (2012). *Multilevel Analysis: An Introduction to Basic and Advanced Multilevel Modeling*. Sage Publications.

Snijders, T. A. B. (2005). *Fixed and Random Effects, in Encyclopedia of Statistics in Behavioral Science*. John Wiley & Sons, Ltd.

Sorbo, A. D., Panichella, S., Alexandru, C. V., Shimagaki, J., Visaggio, C. A., Canfora, G., and Gall, H. C. (2016). What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 499–510.

Sorbo, A. D., Panichella, S., Alexandru, C. V., Visaggio, C. A., and Canfora, G. (2017). SURF: summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 55–58.

Souza, R. R. G., von Flach G. Chavez, C., and Bittencourt, R. A. (2014). Do rapid releases affect bug reopening? A case study of Firefox. In *Proceedings of the 28th Brazilian Symposium on Software Engineering*, SBES '14, pages 31–40.

Souza, R. R. G., von Flach G. Chavez, C., and Bittencourt, R. A. (2015). Rapid releases and patch backouts: A software analytics approach. *IEEE Software*, 32(2):89–96.

StackExchange (2015). What does AUC stand for and what is it? http://stats.stackexchange.com/questions/132777/what-does-auc-stand-for-and-what-is-it. (Last accessed August 2018).

stackoverflow (2011). Difference between /res and /assets directories. http://stackoverflow.com/questions/5583608/difference-between-res-and-assets-directories. (Last accessed August 2018).

stackoverflow (2011). Remove extra unwanted permissions from manifest Android, stackoverflow. http://stackoverflow.com/questions/8257412/remove-extra-unwanted-permissions-from-manifest-android. (Last accessed August 2018).

stackoverflow (2013). Clean up unused Android permissions, stackoverflow. http://stackoverflow.com/questions/18362305/clean-up-unused-android-permissions. (Last accessed August 2018).

stackoverflow (2013). Storage limit in raw and asset folder in Android. http://stackoverflow.com/questions/14995756/storage-limit-in-raw-and-asset-folder-in-android. (Last accessed August 2018).

stackoverflow (2014). How to check if Android permission is actually being used?, stackoverflow. http://stackoverflow.com/questions/24858462/how-to-check-if-android-permission-is-actually-being-used. (Last accessed August 2018).

Statista (2016). Cumulative number of apps downloaded from the google play as of may 2016 (in billions). http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/. (Last accessed: August 2018).

Statista (2018). Number of apps available in leading app stores as of 1st quarter 2018. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. (Last accessed: August 2018).

Stats (2018). Documentation for package 'stats'. https://stat.ethz.ch/R-manual/R-patched/library/stats/html/00Index.html. (Last accessed August 2018).

Stuart Dredge (2013). Information commissioner's office releases app privacy guidelines. http://www.theguardian.com/technology/2013/dec/19/information-commissioners-office-app-privacy-guidelines. (Last accessed August 2018).

Syer, M. D., Nagappan, M., Adams, B., and Hassan, A. E. (2015). Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, 23(3):485–508.

Syer, M. D., Nagappan, M., Hassan, A. E., and Adams, B. (2013). Revisiting prior empirical findings for mobile apps: an empirical case study on the 15 most popular open-source Android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 283–297.

Telerik (2014). Extra Android permissions always set. http://www.telerik.com/forums/extra-android-permissions-always-set. (Last accessed August 2018).

Thelwall, M., Buckley, K., Paltoglou, G., Cai, D., and Kappas, A. (2010). Sentiment in short strength detection informal text. *JASIST*, 61(12):2544–2558.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015). What are the characteristics of high-rated apps? A case study on free Android applications. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 301–310.

Tourani, P., Jiang, Y., and Adams, B. (2014). Monitoring sentiment in open source mailing lists: exploratory study on the Apache ecosystem. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, pages 34–44.

Vásquez, M. L., Bavota, G., Bernal-Cárdenas, C., Penta, M. D., Oliveto, R., and Poshyvanyk, D. (2013). API change and fault proneness: a threat to the success of Android apps. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '13, pages 477–487.

Villarroel, L., Bavota, G., Russo, B., Oliveto, R., and Penta, M. D. (2016). Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 14–24.

Wan, M., Jin, Y., Li, D., and Halfond, W. G. J. (2015). Detecting display energy hotspots

in Android apps. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation*, ICST '15, pages 1–10.

Wilcoxon (2018). Wilcoxon rank sum and signed rank tests. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/wilcox.test.html. (Last accessed August 2018).

Xu, W., Zhang, F., and Zhu, S. (2013). Permlyzer: Analyzing permission usage in Android applications. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*, ISSRE '13, pages 400–410.

Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572.