Studying Bad Updates of Top Free-to-Download Apps in the Google Play Store

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

Abstract— 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 paper, 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 feelings of an app's user-base about a particular update.

Index Terms—mobile app reviews, Google Play Store, bad updates, Android mobile apps

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 [27].

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) [30]. App stores are starting to show the update rating [12]. 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 [17], [18], [26], [41], [49] (including our own prior work [22], [24], [33]) 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. [10] studied topics that are raised in reviews of each update of an app. Gao et al. observed that the distribution of the raised topics for an app changes with each update. In addition, Martin et al. [30] 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 1 shows the percentage of negative reviews per update of the "GasBuddy: Find Cheap Gas" app. As shown in Figure 1, the percentage of negative reviews increased after the U_4 update (September 13th 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 14th 2016), developers deployed the U_5 update that restored the favorites search. As shown in Figure 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 paper, 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

- *Safwat Hassan and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada.*
  E-mail:{shassan, ahmed}@cs.queensu.ca
- *Cor-Paul Bezemer is with the Department of Electrical and Computer Engineering, University of Alberta, Canada.*
  E-mail: bezemer@ualberta.ca

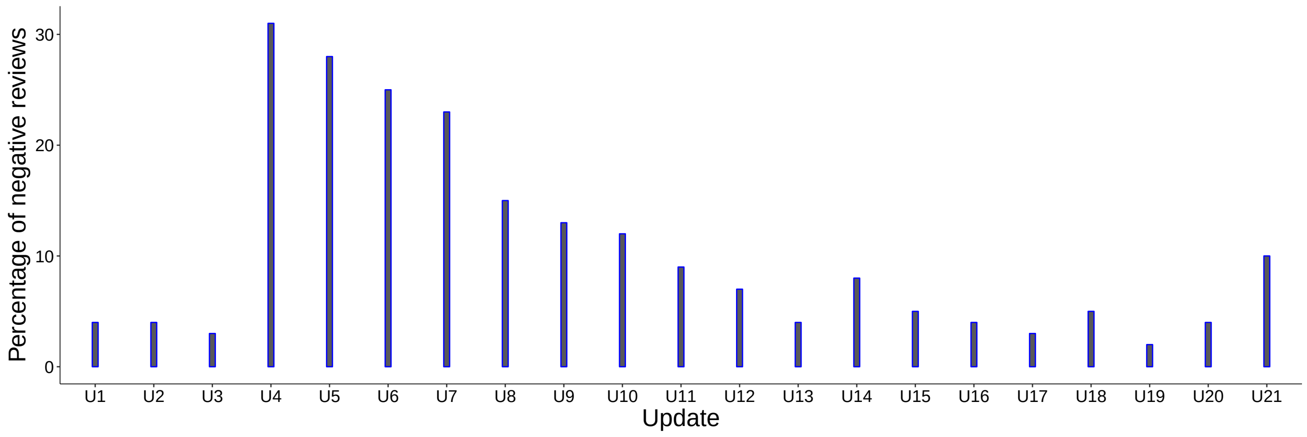


Fig. 1: The percentage of negative reviews for the *"GasBuddy: Find Cheap Gas"* app.

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 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.

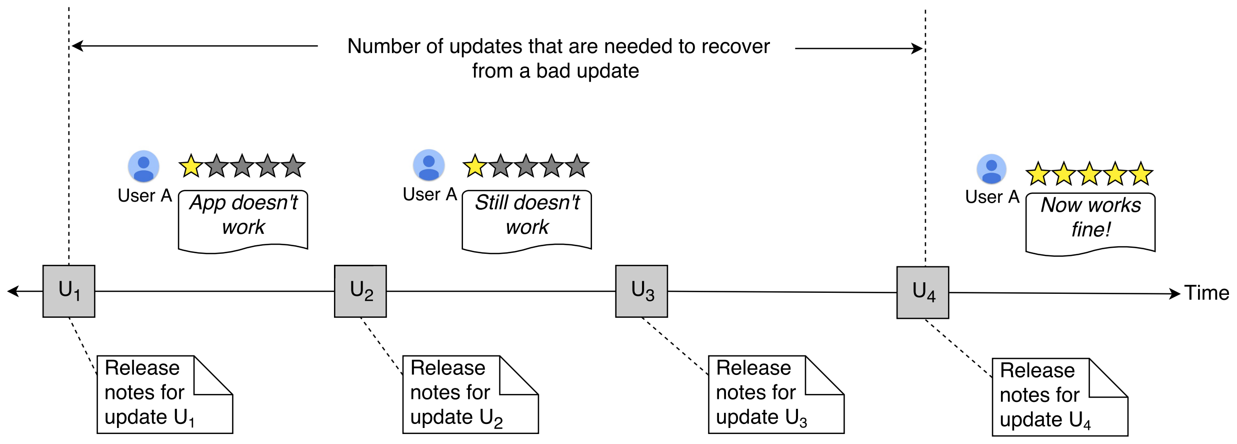
Our findings show that bad updates are not only perceived as bad because of functional issues as previously observed in prior app-centric studies [33]. Instead, we observed that crashes, additional cost and user interface issues are the second most-often raised issues in bad updates.

RQ2: *How do developers recover from a bad update?* Figure 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 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 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 paper is organized as follows. Section 2 discusses the related work. Section 3 describes our methodology for identifying bad updates. Section 4 describes our motivational study of bad updates. Section 5 analyzes the characteristics of bad updates. Section 6 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 7. Section 8 describes



Number of updates that are needed to recover from a bad update
User A
App doesn't work
User A
Still doesn't work
User A
Now works fine!
U_1
U_2
U_3
U_4
Time
Release notes for update U_1
Release notes for update U_2
Release notes for update U_3
Release notes for update U_4

Fig. 2: An overview of our approach for identifying how many updates are needed to recover from a bad update U_i

threats to the validity of our findings. Section 9 concludes the paper.

2 Related Work

In this section, we describe the work that is related to analyzing the characteristics of successful apps and analyzing user reviews.

2.1 Characteristics of Successful Apps

Researchers studied the characteristics of successful apps (i.e., apps with high ratings). In this subsection, we describe the related research and the difference between the existing work and our work.

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 [32], [36], [47], [48]. Ruiz et al. [43] 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 paper, we focus on studying mobile apps while taking an update-centric view.

Later, Martin et al. [28], [30] performed causal impact analysis to study the impact of the deployed updates of an app on the success of this app. The success of an app is measured regarding three metrics: (1) the average app rating (R), (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 the impactful updates. Our work partially responds to this wish by providing an approach for identifying bad updates as bad updates can be impactful (in particular, since stores are now showing update ratings as well) [12]. In addition, we carefully examined the characteristics of bad updates and how developers recovered from such updates.

Our prior work studied emergency updates of top Android mobile apps [13]. We found eight patterns of issues that often lead to emergency updates. Emergency updates could be due to bad updates as developers attempt to fix bad updates on a short notice.

2.2 User Reviews of Mobile Apps

In this subsection, we provide an overview of prior research that analyzes user reviews. For a more thorough overview, we refer to Martin et al.'s [31] and Genc-Nayebi et al.'s [11] surveys.

Researchers often analyze user reviews to extract useful information such as complaints and feature requests [17], [18], [19], [21], [22], [24], [26], [33], [37], [49]. For example, Iacob et al. [17], [18] proposed MARA (Mobile App Review Analyzer) which uses linguistic rules to identify reviews that contain bug reports or feature requests.

Maalej and Nabil [26] used different approaches 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) unspecified 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 approach that is used to classify reviews.

Khalid et al. [22], [24] 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. [33] 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 66% precision and a 65% recall in classifying reviews into the corresponding issue types.

Panichella et al. [41], [42] 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 approaches (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 [41]. Panichella et al. observed that combining machine learning approaches (e.g., sentiment analysis and text analysis) achieves higher accuracy than using a single approach. Panichella et al. evaluated ARdoc on reviews of different mobile apps and found that ARdoc could achieve a precision that ranges from 84% to 89% and a recall that ranges from 84% to 89%.

Di Sorbo et al. [45], [46] proposed SURF (Summarizer of User Reviews Feedback) which is based on Panichella et al.'s approach [41], [42]. First, the SURF approach labels reviews into the five categories that were proposed by Panichella et al. [41], [42]. Then, Di 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. [49] 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. [44] 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. [8] 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, the 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. [39] leveraged Chen et al.'s approach to filter non-informative reviews and proposed the CRISTAL approach. 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. [40] 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. [38] 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 source code elements.

Gao et al. [10] 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) approach 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.

Table 1 summarizes prior studies which analyze user reviews. Prior research mainly focused on app-level analysis of reviews. Such app-level analysis does not identify how users perceive every update and how to learn from users' feedback about every update. In this paper, we demonstrate the benefit of update-level analysis of user reviews to understand what leads to an update being perceived as a bad update and how developers address the raised issues in such updates.

In our prior work [14], we studied the dialogue between app users and developers in the Google Play Store. We found that reviews are not static and users change their reviews over time. For example, a developer reply may lead to an increase in the posted review rating. In this paper, we leverage the analysis of the changes in user reviews by studying if, how and after how long developers recover from a bad update. Gao et al.'s research showed that the distribution of the raised topics for an app changes with each update. Our work differs from Gao et al.'s research as we propose an approach for identifying bad updates and we leverage this approach by analyzing the characteristics of bad updates.

Our work differs from prior research on reviews as we performed an in-depth analysis of reviews' content at the update-level, rather than at the app-level. In particular, we provided an in-depth analysis of bad/good updates and how developers recover from bad updates. Hence, our analysis is useful for researchers to understand the opinion of users about an app at a much finer granularity.

3 METHODOLOGY

In this section, we describe our approach for studying bad updates from the Google Play Store. Figure 3 gives an overview of the steps of our approach. We detail each step below.

3.1 Collecting Data

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

TABLE 1: Summary of the prior studies which analyze user reviews (ordered by the publication year)

Study	Approach name	**Venue-Year**	**Description**
Iacob et al. [17], [18]	MARA	MSR-2013, MobiCASE-2013	An apporach for classifying reviews that contain 1) bug reports or 2) feature requests.
Khalid et al. [22], [24]	-	ICSE-2013, IEEE Soft.-2015	Provided a taxonomy of 13 issue types in negative reviews.
Chen et al. [8]	AR-Miner	ICSE-2014	An approach that filters non-informative reviews and groups similar reviews based on topic extraction.
Maalej and Nabil [26]	-	RE-2015	An approach that classifies reviews into four categories: (1) bug report, (2) feature request, (3) user experience, and (4) unspecified.
Panichella et al. [41], [42]	ARdoc	ICSME-2015, FSE-2016	An approach to classify user reviews into five categories: (1) feature request, (2) bug report, (3) providing information, (4) requesting information, and (5) others.
Palomba et al. [39], [40]	CRISTAL	ICSME-2015, JSS-2018	An approach that links user reviews to the corresponding code changes (i.e., code commits and bug reports).
McIlroy et al. [33]	-	EMSE-20016	An approach for classifying reviews into 14 issue types (e.g., crashes and UI issues).
Di Sorbo et al. [45], [46]	SURF	FSE-2016, ICSE-2017	An approach to label reviews into the five categories that are proposed by Panichella et al. [41], [42]. Di Sorbo et al. identified 12 topics (e.g., pricing) that are raised in user reviews. SURF identifies which topics of these 12 topics that are mentioned in every review. Di Sorbo et al.'s approach is useful to automatically identify reviews that are related to a certain category (e.g., feature request) and a certain topic (e.g., pricing).
Villarroel et al. [49] & Scalabrino et al. [44]	CLAP	ICSE-2016, TSE-2017	An approach to classify reviews into seven types: (1) bug report, (2) feature request, (3) performance issue, (4) energy issue, (5) security issue, (6) usability issues, and (7) others.
Palomba et al. [38]	CHANGEADVISOR	ICSE-2017	An approach to group user reviews that request similar features and map these reviews to the corresponding source code.
Hassan et al. [14]	-	EMSE-2017	Studied the dialogue between app users and developers in the Google Play Store and found that reviews are not static and users change their reviews over time.
Gao et al. [10]	IDEA	ICSE-2018	An approach that indentifies the emerging topics in every update of an app.

3.1.1 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 [1]. 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 [36]. 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.

3.1.2 Crawl App Data Over 12 Months

We used a Google Play crawler [5] 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 [6]). Each time the crawler collects app data, the crawler 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 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 2 describes our dataset. In the

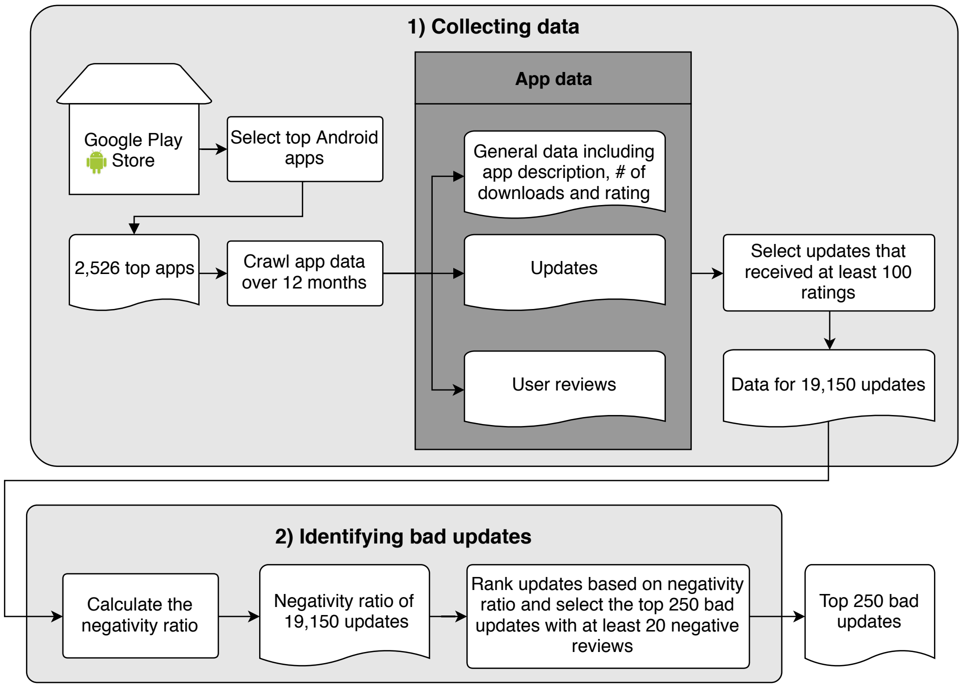

1) Collecting data
Google Play Store
Select top Android apps
2,526 top apps
Crawl app data over 12 months
App data
General data including app description, # of downloads and rating
Updates
User reviews
Select updates that received at least 100 ratings
Data for 19,150 updates
2) Identifying bad updates
Calculate the negativity ratio
Negativity ratio of 19,150 updates
Rank updates based on negativity ratio and select the top 250 bad updates with at least 20 negative reviews
Top 250 bad updates

Fig. 3: An overview of our approach for studying bad updates

TABLE 2: 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

TABLE 3: 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

next section, we explain how we used the collected data to identify bad updates.

3.2 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.

3.2.1 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 two stars [27]) **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:

$$Negativity\ ratio(U_i) = \frac{PNR(U_i)}{PNRB(U_i)} \tag{1}$$

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 3 shows the mean and five-number summary of the negativity ratio of all 19,150 studied updates.

3.2.2 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 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

TABLE 4: 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

to help us understand why users perceive the update as a bad update.

Table 4 shows the number of apps, the number of collected reviews and the number of collected review revisions of the studied bad updates.

3.3 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. The first and the second author 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) [33]. We used the same issue types as McIlroy et al. [33]. Note that Maalej and Nabil used several approaches (e.g., bag of words) to automatically classify user reviews into four high-level categories: bug report, feature request, user opinion or rating [26]. In our analysis of negative reviews, 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 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 authors, then both authors discuss how they interpreted the reviews until both authors agree on the identified issue types of the manually analyzed reviews. Finally, we calculated the agreement between both authors using Cohen's Kappa interrater agreement [9]. Cohen's Kappa measures the agreement between the two authors and provides value ranges from -1 to +1 [9]. The highest Cohen's Kappa measurement value (i.e., +1) means that both authors identified the same issue types in all examined reviews.

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

4 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 subsections.

4.1 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 shows an example of changes of the percentage of negative reviews for three different apps. We observe from Figure 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 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 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 6 shows the histogram of the standard deviation of the percentage of negative reviews per app. Table 6 shows the mean and the five-number summary of the $Range_{neg\%}$ per app.

We observe from Figure 6 and Table 6 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 shows the average percentage of negative

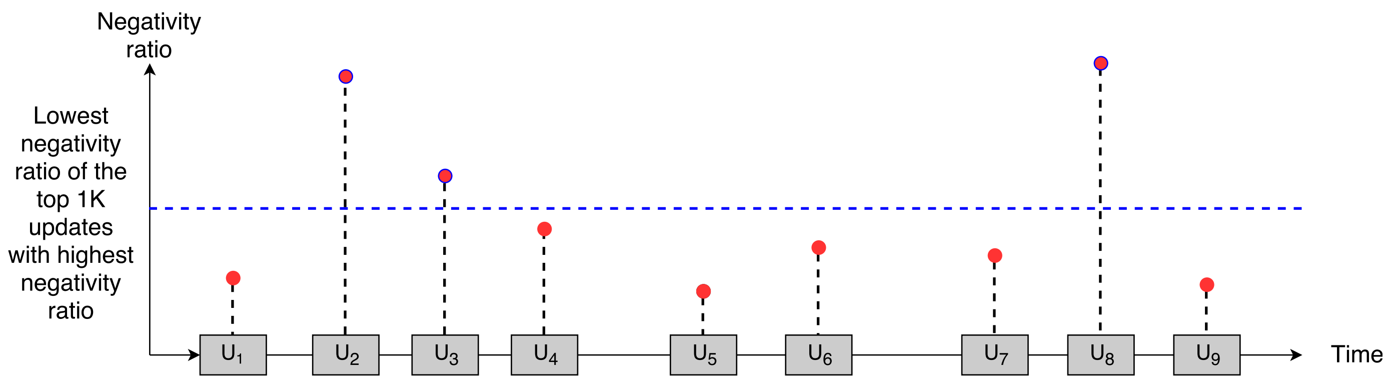

Negativity ratio
Lowest negativity ratio of the top 1K updates with highest negativity ratio
U1
U2
U3
U4
U5
U6
U7
U8
U9
Time

Fig. 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

TABLE 5: 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"*

TABLE 6: 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%

reviews for the same three cases. As shown in Figure 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.

4.2 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., what do users complain about after a

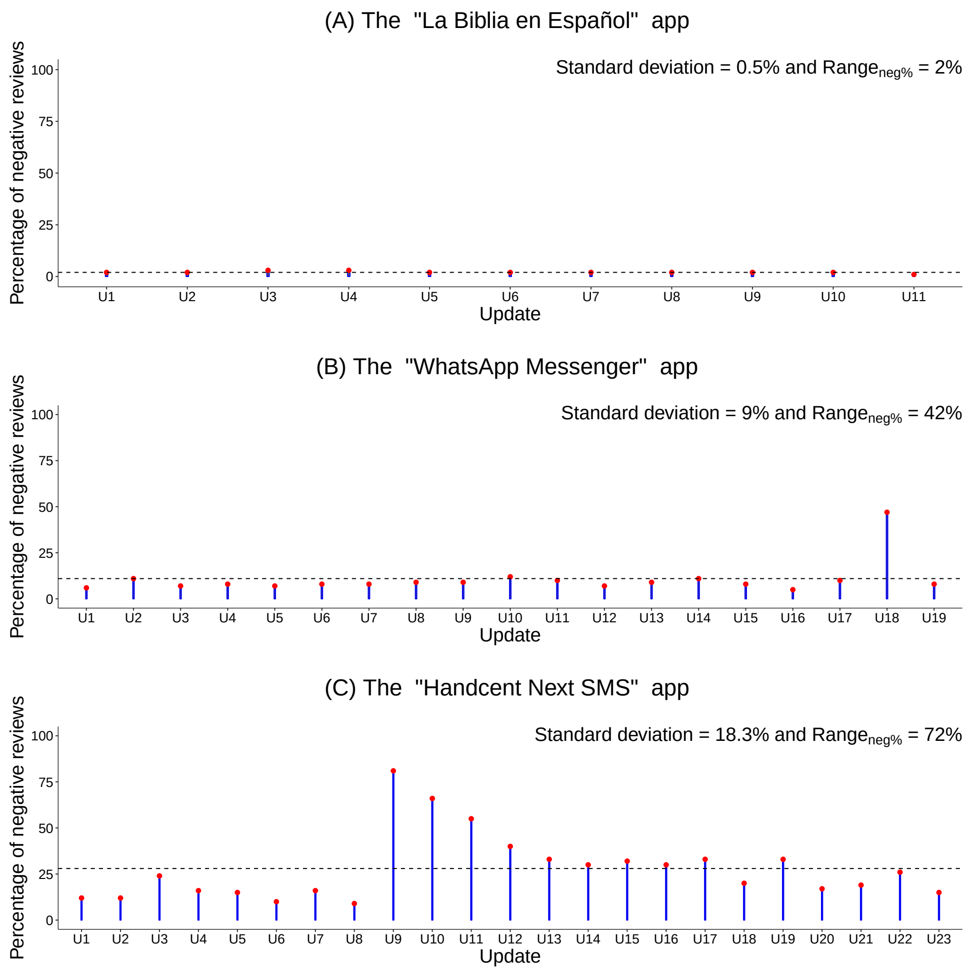


Fig. 5: An example of: (A) the "La Biblia en Español" app with a low standard deviation (0.5%) and a small $Range_{neg\%}$ (2%), (B) the "WhatsApp Messenger" app with a burst in the percentage of negative reviews and a fast recovery of the percentage of negative reviews (the standard deviation value 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.

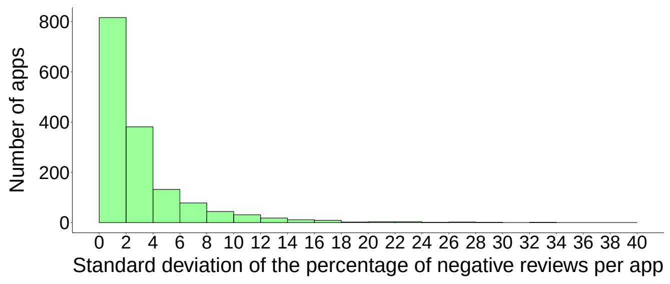


Fig. 6: A histogram of the standard deviation of the percentage of negative reviews per app

bad update?) and how developers often recover from bad updates.

Approach: Figure 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 3, 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

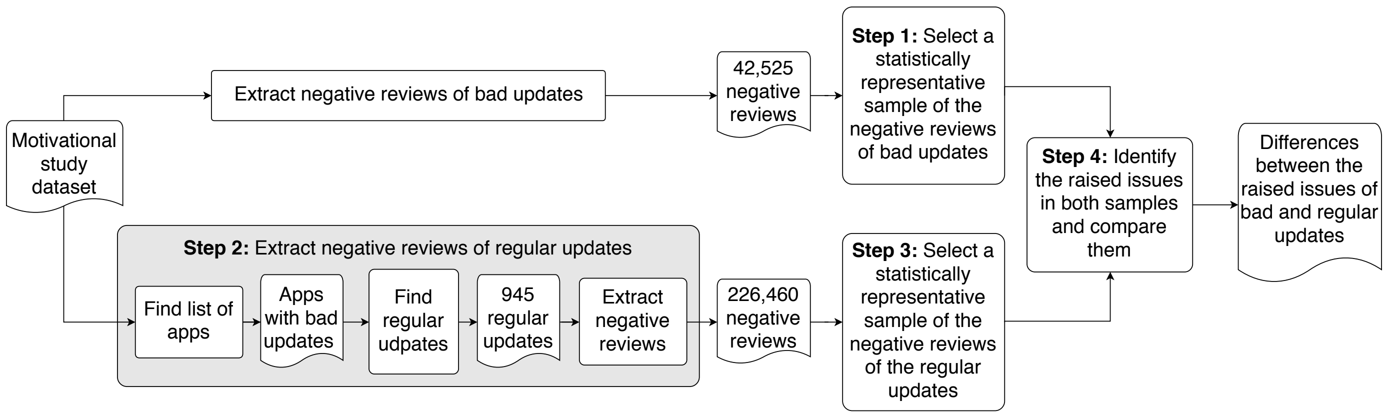

Motivational study dataset
Extract negative reviews of bad updates
42,525 negative reviews
Step 1: Select a statistically representative sample of the negative reviews of bad updates
Step 2: Extract negative reviews of regular updates
Find list of apps
Apps with bad updates
Find regular udpates
945 regular updates
Extract negative reviews
226,460 negative reviews
Step 3: Select a statistically representative sample of the negative reviews of the regular updates
Step 4: Identify the raised issues in both samples and compare them
Differences between the raised issues of bad and regular updates

Fig. 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

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 3.3). We measured the agreement between both authors. 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 [2], [4]. 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 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 authors. We observed that the

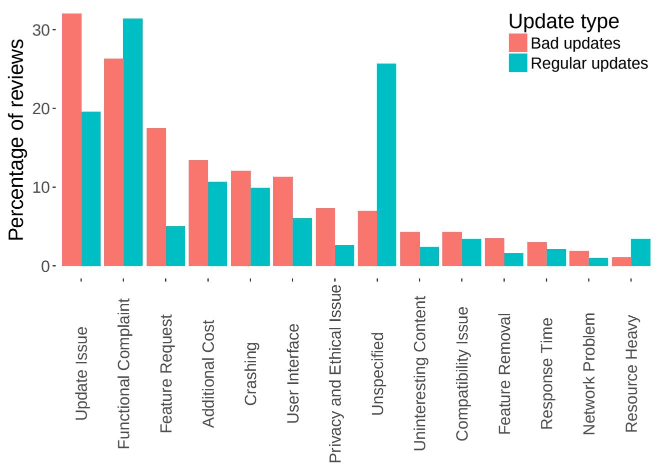


Fig. 8: Distribution of each issue type for both regular updates and bad updates of our motivational study dataset

Cohen's Kappa interrater agreement between both authors 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.*

5 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.

5.1 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 [15], [16], [17], [18], [19], [21], [26], [37], [49]. 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 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, the first and the second author 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 authors independently read the reviews to identify the raised issues in bad updates. If there was a conflict between the two authors, then both authors discussed how they interpreted the reviews until both authors agreed on the identified issues for every bad update. As described in Section 3.3, we used the issue types that are listed in Table 5. The Cohen's Kappa interrater agreement between both authors 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 4, 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 7 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 [23], [36].

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. [33]. 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

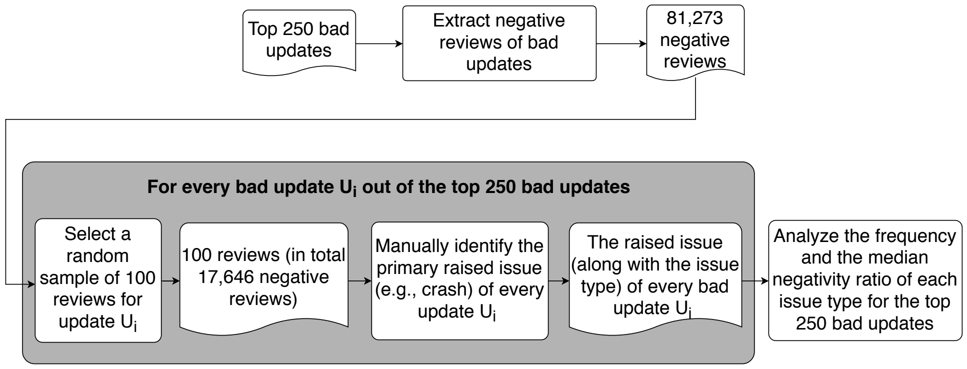

Top 250 bad updates
Extract negative reviews of bad updates
81,273 negative reviews
For every bad update U_i out of the top 250 bad updates
Select a random sample of 100 reviews for update U_i
100 reviews (in total 17,646 negative reviews)
Manually identify the primary raised issue (e.g., crash) of every update U_i
The raised issue (along with the issue type) of every bad update U_i
Analyze the frequency and the median negativity ratio of each issue type for the top 250 bad updates

Fig. 9: An overview of our approach for studying the raised issues in bad updates

TABLE 7: 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

* This column shows the median negativity ratio of the updates that are flagged with this issue type as a primary issue.

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.

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 [7], [25]. 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 8 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 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 9 shows the number of apps with at least one bad update, the number of updates, the percentage of bad updates, and the median number of updates per app in each app category. As shown in Table 9, 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.

TABLE 8: 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"*

TABLE 9: 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

Bad updates are not only perceived as bad because of functional issues. Instead, crash, additional cost and user interface issues are often occurring 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.

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

Motivation: Receiving low ratings and negative reviews can be devastating for an app [27]. 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 steps:

Step 1: Examine changes in reviews of the bad update. As described in Section 3, 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

TABLE 10: 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 11: 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).

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 3, 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 3, 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 10 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 11th 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 15th 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 10 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 11 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

TABLE 12: 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

* This column represents the number of bad updates for which we observed evidence that developers could recover from the update.
** This column shows the median of the difference between the negativity ratio of a bad update and the following update that addressed the primary issue of that bad update.

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 12 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 12, apps are most likely to recover from bad updates where response 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 [23], [36]. 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 [3]. 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.

6 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. [34], [35] proposed an approach that facilitates the generation of mobile app source code from UI design mockups. 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 [12]. 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

TABLE 13: 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 14: 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

that users have the ability to decide whether to download the new update.

7 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 [27]) instead of the negative ones. Table 13 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 14 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 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 15 shows the list of the identified primary reasons of good updates. As described in Section 3.3, Maalej and Nabil studied several machine learning approaches that could be used to automatically classify reviews into four high-level categories: bug report, feature request, user opinion or rating [26]. 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 16 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 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.

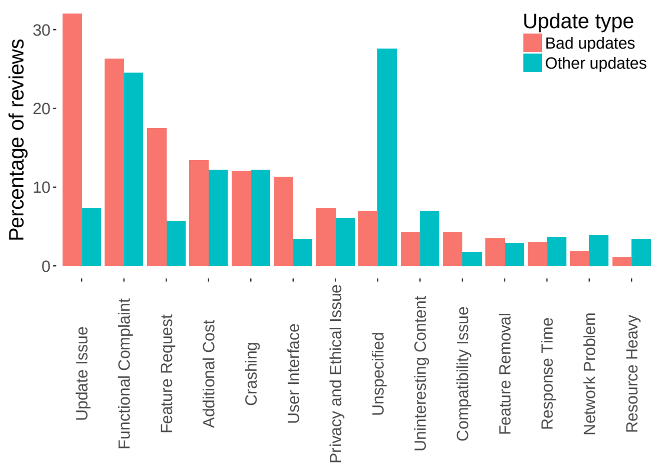


Fig. 10: Distribution of each issue type for both all updates and bad updates of our motivational study dataset

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.

8 Threats to Validity

8.1 Construct Validity

We assume throughout this paper that reviews belong to the latest update at the time of posting the reviews. In our previous work [13], 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 3.

In Section 4.2, 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 4.2, 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 10 shows the distribution of each issue type for all updates and bad updates of our motivational study dataset.

TABLE 15: 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 16: 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%

As shown in Figure 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.

8.2 Internal Validity

We collected data for the top 2,800 free popular apps in 2016 during almost one year. As described in Section 3, 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 feeling of the user-base about every update.

Recently, researchers attempted to automatically label user-reviews with the corresponding issue type (e.g., crashing or bug reports) based on the review content. For example, McIlroy et al. [33] proposed an automated approach that labels reviews with the corresponding issue type with a precision of 66% and a recall of 65%. Later, Maalej and Nabil compared different approaches and algorithms (e.g., bag of words and decision tree) to automatically classify reviews into four high-level categories: (1) bug report, (2) feature request, (3) user experience, and (4) unspecified [26]. Although the proposed approaches by Maalej and Nabil have a higher precision (70 to 95%) and recall (80 to 90%) than McIlroy et al.'s approach, their approach classifies reviews into broad categories (e.g., bug report and feature requests). Such broad categories are too high-level for our study. In our study, we need a deeper understanding of the rationale behind the negative reviews. In particular, for every bad update, we need to understand what are the raised issues and how developers could recover from such issues. 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. For example, if we used Maalej and Nabil's approach to classify the reviews that were listed in Table 5, both reviews *"It does not update cart. Also keeps login me off"* and *"Unable to sync lyrics on samsung j5"* would be identified as bug reports while these two reviews raise different issue types (functional complaint and compatibility issue). This example clearly demonstrates the need for our manual analysis. Hence, even if we would use an automated approach to classify reviews into broad categories, we still need to manually go through the posted reviews of every bad update in order to get in-depth insights about (1) what are the raised issues in every bad update (e.g., users could not log in to the app, or users complain about privacy issues such as the sharing of their GPS location) and (2) how developers could recover from bad updates (e.g., whether the login issue is resolved and whether the developer resolved the previously mentioned privacy issues). In conclusion, while manual analysis might consume more resources than an automated approach, 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 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 [20]. 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.

8.3 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 5.2, we needed to track the changes in user reviews to identify when the raised issues in bad updates are addressed. Martin et al. [29] 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.

9 Conclusions

Below are the key findings of our study:

1) An update-level analysis of reviews is necessary to capture the overall feelings of the user-base about a particular update. An app-level analysis is not sufficient to capture these transient feelings.
2) Bad updates are not only perceived as bad because of functional issues. Instead, crash, additional cost and user interface issues are often occurring 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 were 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.

References

[1] App Annie. https://www.appannie.com/ (Last accessed: July 2018).

[2] Chi-squared test of independence. http://www.r-tutor.com/elementary-statistics/goodness-fit/chi-squared-test-independence. (Last accessed: July 2018).

[3] 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-organize-boycottmcoc-movement/ (Last accessed: July 2018).

[4] Tutorial: Pearson's Chi-square test for independence. http://www.ling.upenn.edu/~clight/chisquared.htm. (Last accessed: July 2018).

[5] Akdeniz. Google Play Crawler. https://github.com/Akdeniz/google-play-crawler (Last accessed: July 2018), Sept. 2013.

[6] AppBrain. Top Android phones. http://www.appbrain.com/stats/top-android-phones. (Last accessed: July 2018).

[7] S. Borgatti. Introduction to grounded theory. http://www.analytictech.com/mb870/introtogt.htm. (Last accessed: July 2018).

[8] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang. AR-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 767–778, 2014.

[9] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[10] C. Gao, J. Zeng, M. R. Lyu, and I. King. Online app review analysis for identifying emerging issues. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, 2018.

[11] N. Genc-Nayebi and A. Abran. A systematic literature review: Opinion mining studies from mobile app store user reviews. *Journal of Systems and Software*, 125:207–219, 2017.

[12] Google. Ratings, Reviews, and Responses. https://developer.apple.com/app-store/ratings-and-reviews/ (Last accessed: July 2018).

[13] S. Hassan, W. Shang, and A. E. Hassan. An empirical study of emergency updates for top Android mobile apps. *Empirical Software Engineering*, 22(1):505–546, 2017.

[14] S. Hassan, C. Tantithamthavorn, C.-P. Bezemer, and A. E. Hassan. Studying the dialogue between users and developers of free apps in the Google Play Store. *Empirical Software Engineering*, 23(3):1275–1312, 2017.

[15] H. Hu, C.-P. Bezemer, and A. E. Hassan. 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*, 2018.

[16] H. Hu, S. Wang, C.-P. Bezemer, and A. E. Hassan. Studying the consistency of star ratings and reviews of popular free hybrid Android and iOS apps. *Empirical Software Engineering*, 2018.

[17] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 41–44, 2013.

[18] C. Iacob, R. Harrison, and S. Faily. 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, 2013.

[19] C. Iacob, V. Veerappa, and R. Harrison. What are you complaining about?: a study of online reviews of mobile applications. In *Proceedings of the 27th International BCS Human Computer Interaction Conference*, BCS-HCI '13, page 29, 2013.

[20] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman. ScottKnott: A package for performing the Scott-Knott clustering algorithm in R. *Trends in Applied and Computational Mathematics*, 15(1):3–17, 2014.

[21] S. Keertipati, B. T. R. Savarimuthu, and S. A. Licorish. 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, 2016.

[22] H. Khalid. On identifying user complaints of iOS apps. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 1474–1476, 2013.

[23] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the devices to test your app on: a case study of Android game apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 610–620, 2014.

[24] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[25] S. H. Khandkar. Open coding. http://pages.cpsc.ucalgary.ca/~saul/wiki/uploads/CPSC681/open-coding.pdf. (Last accessed: July 2018).

[26] W. Maalej and H. Nabil. 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, 2015.

[27] P. Martin. 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: July 2018).

[28] W. Martin. Causal impact for app store analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 659–661, 2016.

[29] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. The app sampling problem for app store mining. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 123–133, 2015.

[30] W. Martin, F. Sarro, and M. Harman. 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, 2016.

[31] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9):817–847, 2017.

[32] S. McIlroy, N. Ali, and A. E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering*, 21(3):1346–1370, 2016.

[33] S. McIlroy, N. Ali, H. Khalid, and A. E. Hassan. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering*, 21(3):1067–1106, 2016.

[34] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, 2018.

[35] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, 2018.

[36] E. Noei, M. D. Syer, Y. Zou, A. E. Hassan, and I. Keivanloo. A study of the relation of mobile device attributes with the user-perceived quality of Android apps. *Empirical Software Engineering*, 22(6):3088–3116, 2017.

[37] J. Oh, D. Kim, U. Lee, J. Lee, and J. Song. 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, 2013.

[38] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. C. Gall, F. Ferrucci, and A. D. Lucia. 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, 2017.

[39] F. Palomba, M. L. Vásquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 291–300, 2015.

[40] F. Palomba, M. L. Vásquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software*, 137:143–162, 2018.

[41] S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can I improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 281–290, 2015.

[42] S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. 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, 2016.

[43] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan. Examining the rating system used in mobile-app stores. *IEEE Software*, 33(6):86–92, 2016.

[44] S. Scalabrino, G. Bavota, B. Russo, R. Oliveto, and M. D. Penta. Listening to the crowd for the release planning of mobile apps. *IEEE Transactions on Software Engineering*, pages 1–1, 2017.

[45] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 499–510, 2016.

[46] A. D. Sorbo, S. Panichella, C. V. Alexandru, C. A. Visaggio, and G. Canfora. SURF: summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 55–58, 2017.

[47] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. What are the characteristics of high-rated apps? A case study on free Android applications. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 301–310, 2015.

[48] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. 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, 2013.

[49] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. D. Penta. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 14–24, 2016.

Safwat Hassan is a Ph.D. candidate at School of Computing, Queen's University. Safwat worked as a software engineer for ten years in different corporations like Egyptian Space Agency (ESA), HP, EDS, VF Germany (outsourced by HP), and Etisalat. During his ten years work experience, he worked on different large-scale systems (varying from Web-Based systems to embedded systems) and in diverse project types (design service, customer support, and R&D) across various domains (Telecommunication, Supply-chain, and Aerospace). His research interest includes data mining, big data analytics, software engineering, mobile app store analytics. Contact him at shassan@cs.queensu.ca.

Cor-Paul Bezemer currently works as an assistant professor in the Electrical and Computer-ing Engineering department at the University of Alberta in Canada. Before that, he was a post-doctoral research fellow in the Software Analysis and Intelligence Lab (SAIL) at Queen's University in Kingston, Canada. His research interests cover a wide variety of software engineering and performance engineering-related topics. His work has been published at premier software engineering venues such as the TSE and EMSE journals and the ESEC-FSE, ICSME and ICPE conferences. He is one of the vice-chairs of the SPEC research group on DevOps Performance. Before moving to Canada, he studied at Delft University of Technology in the Netherlands, where he received his BSc (2007), MSc (2009) and PhD (2014) degree in Computer Science. More about Cor-Paul can be read on his website: https://www.ece.ualberta.ca/~bezemer/

Ahmed E. Hassan is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. Hassan received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. Hassan also serves on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. Contact him at ahmed@cs.queensu.ca.