An Empirical Investigation into the Reproduction of Bug Reports for Android Apps

Jack Johnson*, Junayed Mahmud†, Tyler Wendland*, Kevin Moran†, Julia Rubin‡, Mattia Fazzini*
*University of Minnesota, MN, USA; joh19267@umn.edu, wendl155@umn.edu, mfazzini@umn.edu
†George Mason University, VA, USA; jmahmud@gmu.edu, kpmoran@gmu.edu
‡University of British Columbia, BC, Canada; mjulia@ece.ubc.ca

***Abstract*—One of the key tasks related to ensuring mobile app quality is the reporting, management, and resolution of bug reports. As such, researchers have committed considerable resources toward automating various tasks of the bug management process for mobile apps, such as reproduction and triaging. However, the success of these automated approaches is largely dictated by the characteristics and properties of the bug reports they operate upon. As such, understanding mobile app bug reports is imperative to drive the continued advancement of report management techniques. While prior studies have examined high-level statistics of large sets of reports, we currently lack an in-depth investigation of how the information typically reported in mobile app issue trackers relates to the specific details generally required to reproduce the underlying failures.**

In this paper, we perform an in-depth analysis of 180 reproducible bug reports systematically mined from Android apps on GitHub and investigate how the information contained in the reports relates to the task of reproducing the described bugs. In our analysis, we focus on three pieces of information: the environment needed to reproduce the bug report, the steps to reproduce (S2Rs), and the observed behavior. Focusing on this information, we characterize failure types, identify the modality used to report the information, and characterize the quality of the information within the reports. We find that bugs are reported in a multi-modal fashion, the environment is not always provided, and S2Rs often contain missing or non-specific enough information. These findings carry with them important implications on automated bug reproduction techniques as well as automated bug report management approaches more generally.

I. Introduction

The importance of the quality of mobile applications (colloquially referred to as apps) has grown in recent years as smartphones and tablets have become deeply integrated into users' daily lives. Once an application has been released to users, its quality is largely ensured by continuing maintenance activities, which have been shown to consume considerable amounts of engineering effort [1]. These important maintenance activities are typically centered around *bug report management* and include activities related to understanding, reproducing, and resolving bug reports.

A number of unique development constraints related to mobile apps, such as pressure for frequent releases [2], [3], the need to cope with constantly evolving platform APIs [4], [5], a large volume of user feedback [6], [7], [8], [9], [10], and testing challenges [11] complicate the bug report management process. Software engineering researchers have recognized these domain-specific challenges and have worked toward providing automated solutions across several bug report management activities for mobile apps, including bug report quality assessment [12], reproduction [13], [14], triaging [15], and bug localization [16], [17].

One common thread among these various automated solutions is that they operate directly upon the information contained within bug reports and, as such, are directly affected by the characteristics and quality of various report components, such as environmental information (e.g., device, software version), reproduction steps (S2Rs), and observed behavior (OB). Thus, researchers and practitioners require a solid empirical foundation that delineates common characteristics of mobile app bug reports to build effective automated techniques.

In prior work, researchers have examined high-level statistics (e.g., number and type of report, fix rates, fix time) of large sets of bug reports. For example, Battacharya *et al.* [18] performed an empirical study on bugs submitted to the Android platform on 24 widely-used open source apps. Others have compared high-level bug characteristics between mobile apps and desktop apps [19]. However, to the best of our knowledge, no study has yet provided an in-depth characterization of how the information contained in mobile bug reports might impact the task of bug reproduction. One likely reason that past studies have not examined this relation is that as it requires *manually reproducing* real bug reports, which is a time-consuming and difficult task. Despite the difficulty of this analysis, understanding this information is critical as both developers and automated bug analysis techniques may need to (i) understand the type of reported failure, (ii) understand multiple modalities of information, such as text, images, or screen-recordings, and (iii) identify or infer information that is either vague or missing from the reports. In short, empirically analyzing both the *characteristics* and *quality* of the information reported in mobile app bugs is critical for both the practical and scientific advancement bug report management for mobile apps.

In this paper, we conduct and in-depth characterization of reproducible bug reports for Android apps. To this end, we significantly extend AndroR2 [20] – a dataset of reproducible bug reports for Android apps which contains bugs representing a range of failure types. We augmented the dataset with additional, manually verified and fully reproduced bug reports from open source Android apps hosted on GitHub [21] and available on the Google Play store [22], obtaining a dataset of 180 bug reports. In this work, we focus on bug reports for Android

apps as Android is the most widely used operating system for mobile apps [23]. To the best of our knowledge, ours is the largest dataset of (i) fully reproduced bug reports for Android apps, which (ii) contains both user-submitted and developer-submitted reports, and (iii) in contrast to related work, focuses on different types of failures beyond app crashes. Given this dataset, we focused our in-depth analysis on three sources of information: the description of the environment needed to reproduce the bug report, the steps to reproduce, and the observed behavior.

Leveraging the fact that our studied reports are considered fully reproducible, we perform an in-depth analysis of both the report *characteristics*—including the failure types and modalities of reported information—and the *quality* of reported information. In relation to the quality of reported information, we focus on three aspects: the types and prevalence of missing information, whether report discussion threads contain helpful information for reproducing the reports, and the specificity of reported information (which investigates whether reported information can be directly used for reproducing the reports). Although these aspects are only some of ones that describe the quality of reported information, we believe that the analysis of these aspects provides useful insights into the reproduction of bug reports and hence focus on them.

Our analysis shows that (i) reported failures can be grouped into four types, three of which are not yet considered by existing automated reproduction techniques, (ii) different information modalities are used to report the details related to the environment, steps to reproduce, and observed behavior, (iii) a large number of reports (74%) have at least one step to reproduce that requires multiple operations in the app indicating that the information provided for the step is not always specific enough, (iv) the great majority of reports (92%) have at least one missing reproduction step, illustrating that the operations required to reproduce the reports must often be inferred, and (v) bug report discussions can, in some cases (19%), provide additional information useful for the reproduction of the reports. Finally, we discuss implications of our findings, which can help guide future research on automated reproduction of bug reports and, more generally, bug report management activities.

In summary, the main contributions of this paper are:

- A large set of 180 manually mined and reproduced bug reports for Android apps that contains user- and developer-submitted bug reports of multiple failure types.
- A study that examines bug characteristics and information quality in reproducible mobile app bug reports. This advances upon prior studies which do not manually verify and collect reproducible bug reports.
- A discussion on the implications of our findings, which illustrates the need for future research on non-crashing oracles, multi-modal understanding of report information, mocking environments, and missing and non-specific reproduction steps.
- A replication package [24] that contains our dataset of bug reports, data analysis reports, and scripts to perform the study analyses, which can facilitate future replications and extensions of this work.

Bug Report

Title:
Bug: Long pressing the amount input brings up QWERTY keyboard
Content:
Software specifications:
• GnuCash Android version: 2.2.0
• System Android version: 6.0
• Device type: Motorola Moto G (2nd Generation)
Steps to reproduce the behaviour:
1. Navigate to Transactions screen
2. Tap the Add button
3. Enter Description (optional)
4. Focus the Amount input
5. Long press to bring up the context menu
Expected behaviour:
See the context menu
Actual behaviour:

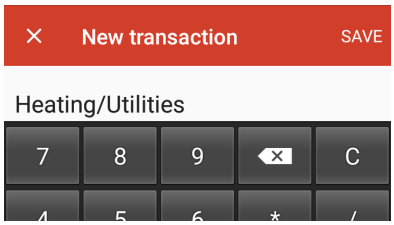


Fig. 1: Bug report for the GNUCASH app.

II. Background and Terminology

Given a bug report that describes a failure in an app, we use the term *reporter* to identify the person submitting the bug report. A reporter can be either a *user* or a *developer*. In this study, we consider a person who never contributed to the source code of an app to be a user and all other reporters to be developers.

We conceptually group the information contained in a bug report into multiple parts, each of which detail a particular aspect of the report. The parts and aspects of interest in this study are the ones providing details on how to reproduce the failure described in a report. These aspects are: the *environment*, the *steps to reproduce* (S2Rs), and the *observed behavior* (OB). The environment includes information on the software and hardware necessary to reproduce the failure described in a report. This part can contain information such as the app version, the operating system (OS) version, and the device where the failure occurred. The S2Rs provide details on the operations that should be performed on a device in order to reproduce the failure. We use the terms *GUI action* (or simply *action*) and *GUI interaction* (or simply *interaction*) interchangeably to indicate the operations performed on the GUI of a device. An S2R (which are the unit of information composing the S2Rs) can be mapped to one or more GUI actions. The OB describes the failure and can be used to check that the failure was successfully reproduced. In practice, the information from these conceptual parts can be interleaved across the paragraphs and sections of a bug report. Bug reports can also have a *discussion thread*. A discussion thread contains *discussion messages* and these messages can provide

additional information on the environment, the S2Rs, and the OB associated with the report.

Figure 1 provides an example of a user-submitted bug report [25]. This bug report is taken from the report management system of GNUCASH, an app for finance tracking, and is slightly modified for presentation purposes. The bug report contains information related to the environment, the S2Rs, and the OB, which are located in the *Software specifications*, *Steps to reproduce the behaviour*, and *Actual behaviour* sections of the report, respectively.

To exercise the bug, the user navigated to the transactions screen, started adding a new transaction, and long-clicked on the GUI element representing the amount of the transaction. The failure manifests as a wrong screen being displayed to the user: screen with a keyboard view instead of the context menu. The OB describing the failure is reported using text (in the title) and using an image (in the *Actual behaviour* section). We refer to the way in which a piece of information is reported as the *reporting modality* (or *modality* in short) and reporters can provide the same information multiple times using different modalities. Because the user did not reach the desired screen, we identify this failure as a *navigation failure*. We use the terms *failure type* and *failure category* interchangeably to refer to the categorization of the failure.

The report has five S2Rs (numbered items under the *Steps to reproduce the behaviour* section) and 13 GUI actions are necessary to reproduce the failure. An example of GUI action is performing a click on the add button in the transaction screen of the app as indicated by *2. Tap the Add button*. An S2R can map to one or more GUI actions. In this example, the first S2R (*1. Navigate to Transactions screen*) maps to three GUI actions. We refer to S2Rs that map to multiple GUI actions as *non-specific S2Rs*. Of the remaining four S2Rs, three map to one GUI action and one S2R is optional (*3. Enter Description (optional)*.) This optional S2R is not included in 13 GUI actions necessary to reproduce the failure. Seven (13-3-3) of the GUI actions in this example are not described by any of the S2Rs. We refer to such GUI actions as *unmapped GUI actions* and say that they correspond to *missing S2Rs*. We refer to the remaining actions as *mapped GUI actions*. If an unmapped GUI action occurs before the first mapped GUI action, we call the missing S2R that corresponds to the unmapped action a *missing context S2R*, indicating that some contextual information is missing from the bug report. Otherwise, if a missing S2R is associated with a GUI action occurring after the first mapped GUI action, we refer to the S2R as a *missing inline S2R*.

III. METHODOLOGY

To characterize reproducible bug reports, inform research on automated bug reproduction, and, more generally, provide insights for research on bug report management, we formulated and answered the following research questions (RQs):

- **RQ$_1$: What are the failure types associated with reproducible bug reports?** In this RQ, we analyzed and categorized failures associated with reproducible bug reports. With the findings from this RQ we aim to inform research on automatic failure recognition.
- **RQ$_2$: What information modalities are used to report the information contained in reproducible bug reports?** This RQ categorizes the modalities used to report environment, S2Rs, and OB information. The findings from this RQ aim to inform research in bug triaging, report reproduction, and report quality assessment.
- **RQ$_3$: Do reproducible bug reports have missing information?** We answer this question by analyzing the information contained in reproducible bug reports w.r.t. operations required to reproduce the failures described in the reports. This RQ aims to direct efforts on research for identifying and inferring missing information in bug reports, necessary for bug report reproduction.
- **RQ$_4$: Do discussion threads of reproducible bug reports contain helpful information for reproducing the reports?** In this RQ, we analyzed the information gain obtained by interpreting the bug report discussions. This RQ aims to evaluate the need for approaches that combine content from bug reports and their discussions.
- **RQ$_5$: How specific is the information reported in reproducible bug reports?** In this RQ, we investigated whether the information contained in reproducible bug reports can be directly mapped onto the operations need to reproduce the reports. This RQ aims to provide insights on how to leverage the information in bug reports for reproducing the failures.

Figure 2 provides a high-level outline of the methodology we used to answer the RQs. In a nutshell, we first assembled a dataset of reproducible bug reports and then analyzed the characteristics of the bug reports through qualitative and quantitative analyses. We describe these steps in detail next.

A. Dataset Creation

The *Dataset Creation* component of Figure 2 provides an overview of our data collection workflow, which consisted of two phases: *bug reports filtering* and *failure reproduction*.

1) Bug Reports Filtering: The objective of this phase was to identify a set of bug reports that we could try to reproduce and ultimately include in our dataset. In this study, we are interested in both user-submitted and developer-submitted bug reports that are reproducible and describe different types of failures. To the best of our knowledge, ANDROR2 [20] is the largest dataset of reproducible bug reports for Android apps that does not exclusively focus on crashes. This dataset contains 90 user-submitted bug reports, which are associated with apps available on the Google Play store [22] and hosted on GitHub [21]. The 90 bug reports are GitHub issues [26] and are associated with reproduction scripts created by the ANDROR2's authors. This set of 90 bug reports was extracted from a larger set of 6,365 issues that was systematically mined from GitHub. The set of 6,365 issues contains issues that: (i) are part of repositories that use Java, (ii) have the label "bug", (iii) are in repositories that contain an `AndroidManifest.xml` file (as Android apps require this

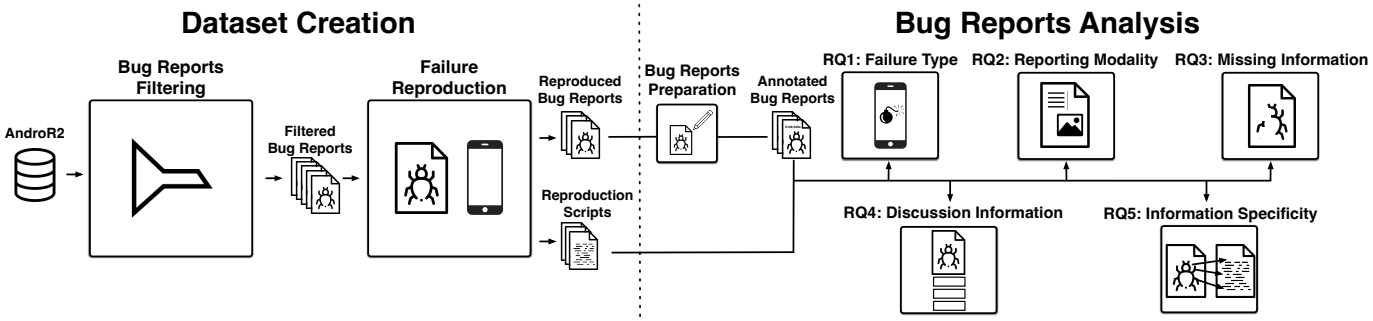


Fig. 2: Overview on the methodology used in the study.

file to properly compile [27]), (iv) contain the word "step" in them, and (v) are associated with apps also available on the Google Play store.

Because we are also interested in developer-submitted bug reports, we started from the set of 6,365 GitHub issues provided by ANDROR2 and identified 90 reproducible, developer-submitted bug reports (to match the number of already available user-submitted bug reports). To identify the 90 developer-submitted bug reports, we used a methodology similar to that of ANDROR2. Specifically, we first refined the set of 6,365 issues to only contain those created by GitHub users that had contributed to the repositories associated with the issues, resulting in 2,523 issues. Second, we selected issues that were closed at the time the issues were mined (November 2020) so that we could more easily identify whether the issues were also originally reproduced by the developers. This filtering resulted in 2,045 reports. Third, after analyzing the set of issues, we found that some repositories had a much larger number of issues compared to others. To avoid overfitting the bug report dataset to a specific app, we considered at most ten issues per repository. When a repository had more than ten issues, we randomly selected ten from this set resulting in 645 bug reports for 164 apps.

2) Failure Reproduction Phase: In the second phase of our dataset creation process, we randomly selected bug reports from the set of 645 developer-submitted bug reports until we reproduced 90 of them. In this process, we disregarded trivially reproducible bug reports, i.e., those we could reproduce by simply opening the app.

Two authors tried to reproduce the failures described in the bug reports. To reproduce a failure, the authors followed the S2Rs contained in the bug report by mapping the steps to GUI actions on the screen of the device running the app associated with the report. If a report had missing S2Rs, the authors manually explored the functionality of the app to identify the minimal sequence of GUI actions that would account for those missing steps, using a trial-and-error approach. When a bug report could be successfully reproduced by one of the two authors, the other author also tried to reproduced the same report to ensure that the reproduced failure was the same as the one described in the report. For all 90 bug reports, the authors also encoded the GUI actions in reproduction scripts using the UIAutomator framework [28].

To validate whether user-submitted bug reports were still reproducible, we ran the scripts associated with these reports in the ANDROR2 dataset. Four reports were not reproducible as the servers associated with the apps were no longer running. To replace these bug reports, we identified and reproduced four additional user-submitted reports from the set of 6,365 GitHub issues provided by ANDROR2. At the end of this process, we obtained a set of 90 user-submitted and 90 developer-submitted reproducible bug reports, which we considered for the rest of the study.

B. Bug Reports Analysis

In this section, we present the analyses we performed to characterize aspects related to the reproducibility of Android bug reports. The *Bug Reports Analysis Creation* part of Figure 2 provides a summary of the analyses we performed. The analyses were driven by two of the paper's authors and were performed one at a time to reduce cognitive load.

1) Bug Reports Preparation: Before performing the analyses associated with the RQs, we annotated the information contained in the bug reports and their discussion threads, to identify the portions of each report that provide information about the environment, S2Rs, and OB. This step was performed by the two authors together and in multiple sessions; the authors associated each sentence in the report's textual description, as well as each link, image, recording, and execution logs, with it designated purpose: to describe environment, S2Rs, and OB. Some elements received multiple annotations, e.g., a sentence can provide both S2Rs and OB.

2) Analysis for RQ_1 (What are the failure types associated with reproducible bug reports?): To answer RQ_1, we performed a qualitative analysis that combines inductive and axial coding [29], [30]. Inductive coding is a systematic approach for categorizing data by manually coding (i.e., labeling) the data. Axial coding relates codes to one another and finds higher-level codes that represent abstractions of the original codes. In our analysis, a code is a label that categorizes the type of a failure and we assigned the code to the bug report describing the failure.

The analysis was performed by two raters, who analyzed the description of the failure in the bug report and used the reproduction scripts to observe how the failure manifested. The analysis was divided into two parts. In the first part, the two raters analyzed a sample of the bug reports to define the analysis codebook – a document detailing the rules for assigning a specific code to a failure. For each code, the set

of rules specified the characteristics required for assigning a code to a failure.

This part of the analysis was performed in six iterations. In each iteration, the raters independently analyzed 18 bug reports (10% of the report considered in the study). The set contained the same bug reports for both raters and was selected randomly from the set of not-yet-analyzed bug reports. At the end of each iteration, the raters used negotiated agreement [31] to resolve inconsistencies among created and assigned codes, and to insure the reliability of the coding process. We used this method due to it is advantages in research like ours, where generating new insights is the primary concern [32]. Because we used negotiated agreement, measures such as inter-rater agreement are not applicable in our context. To resolve disagreements, the raters reproduced the failures together and then decided on the final classification. For example, for one of the reports considered in the study [33], one of the raters categorized the failure as a crash and the other rater categorized the failure as a navigation issue. When the two raters met, they discussed the disagreement and decided to classify the failure as a crash because the app displayed an exception before bringing the user back to a different screen.

At the sixth iteration, the raters did not create new codes and had assigned the same codes to all reports. From that point, the raters split the remaining 72 bug reports equally and coded the bug reports independently. At the end of the coding process, the raters also performed axial coding. This step led to four main categories of failures, which we present in Section IV.

3) Analysis for RQ_2 (What information modalities are used to report the details contained in reproducible bug reports?): The analysis to answer RQ_2 was also based on inductive and axial coding. Two raters analyzed the environment, S2Rs, and OB information annotated during the bug reports preparation step. The raters created the analysis codebook in two iterations, analyzing in each iteration a sample of 18 bug reports (10% of all bug reports). The raters used negotiated agreement to address the reliability of the coding process. After finalizing the codebook, the authors split the remaining 144 bug reports equally and coded them independently.

The raters performed axial coding at the end of the coding process. This process led to six main reporting modalities, detailed in Section IV.

4) Analysis for RQ_3 (Do reproducible bug reports have missing information?): To answer RQ_3, we performed two types of analysis. First, we leveraged the annotations created in the bug reports preparation step to identify whether environment, S2Rs, and OB information was completely missing from the reports. Second, when the S2Rs information was provided, we performed an in-depth analysis of S2Rs. Specifically, for each bug report, we compared the S2Rs information from the bug report with the GUI actions in our reproduction scripts, in order to identify missing S2Rs. Once we identified missing S2Rs, we categorized them into missing context S2Rs and missing inline S2Rs (see definitions in Section II). Two authors analyzed each bug report independently and then met to discuss and finalize the classification.

5) Analysis for RQ_4 (Do discussion threads of reproducible bug reports contain helpful information for reproducing the reports?): In RQ_4, two authors manually analyzed the messages in the bug report discussions, to identify whether they added information relevant to understanding and reproducing the bug reports. The authors leveraged the annotations from the bug reports preparation step to focus on messages providing environment, S2Rs, and OB information. The authors analyzed each bug report independently and labeled with the word *additional* the data from discussion messages that provided additional information. The two authors met and discussed the final classification also in this case.

6) Analysis for RQ_5 (How specific is the information reported in reproducible bug reports?): To answer RQ_5, we analyzed whether the information provided in the bug reports could be directly used for reproducing the bug reports. For the environment-related information, two authors checked whether the provided information was sufficient to define the environment where to reproduce the failure. If no additional information was needed, we considered the provided information to be of specific (and non-specific otherwise). For S2Rs, two authors mapped each of the S2Rs defined in a bug report to corresponding GUI actions from the reproduction script. If an S2R mapped to multiple GUI actions, we labeled that S2R as a non-specific S2R. We considered the other S2Rs to be specific. For the OB information, the authors checked whether the information was sufficient to verify the failure. If no additional information was needed (i.e., no need to check discussion messages), we considered the provided information to be specific (and non-specific otherwise).

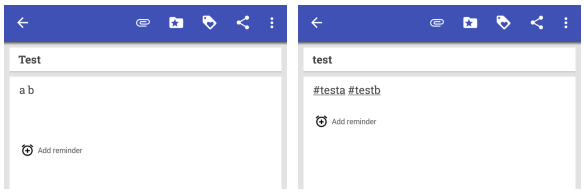
IV. Results

In this section, we present the results of our study on analyzing and characterizing reproducible Android bug reports.

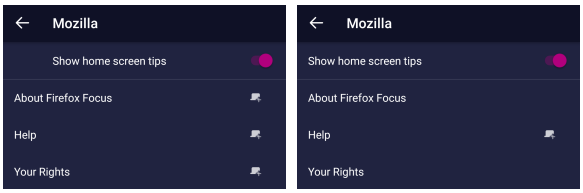
A. RQ_1: What are the failure types associated with reproducible bug reports?

Our analysis identified four failures types: *output*, *cosmetic*, *navigation*, and *crash*. Output failures reveal issues in the output provided by the app. Cosmetic failures identify issues in the app that do not affect the functionality of the app. Navigation failures display the wrong screen to the user. Crashes abruptly terminate the execution of the app. Across the bug reports considered, we identify 33% of reports reporting output failures, 31% reporting cosmetic failures, 8% reporting navigation failures, and 28% reporting crashes. This finding is notable, as many current bug report analysis techniques focus solely on crashes. We discuss the implications of these findings further in Section V.

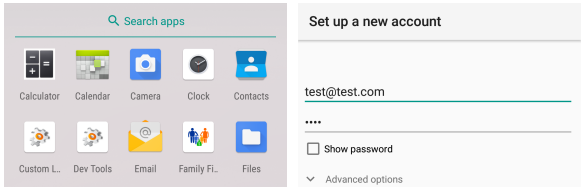
This distribution reveals a comparable amount of failures between the output, cosmetic, and crash categories and a significantly lower number of navigation failures. The distribution is similar across both developer- and user-submitted bug reports. Specifically, among the user-submitted bug reports, there are 33% output failures, 31% cosmetic failures, 7% navigation failures, and 28% crashes. Among developer-submitted bug



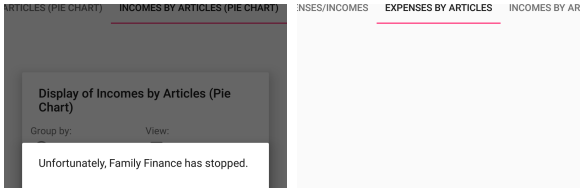
(a) Example of output failure on the left and fix on the right.



(b) Example of cosmetic failure on the left and fix on the right.



(c) Example of navigation failure on the left and fix on the right.



(d) Example of crash failure on the left and fix on the right.

Fig. 3: Screenshot examples for the four failure types identified in the bug reports considered.

reports, there are 32% output failures, 29% cosmetic failures, 9% navigation failures, and 29% crashes.

Our analysis categorized the 60 output failures into two subcategories: *incorrect output* (32) and *missing output* (28). *Incorrect output* identifies failures in which some computation of the app is displayed incorrectly or improperly saved to a file, and *missing output* describes failures where the result of some computation is not displayed or saved to a file. A vast majority of these cases affect the GUI of the app (56 cases) whereas a smaller number impact generated files (4 cases).

The screenshot on the left of Figure 3a shows an example of a failure under the incorrect output subcategory. The example is taken from a bug report [34] of OMNI NOTES, a note-taking app. The app has a failure as it does not display the right values for the tags associated with the notes in the app.

As part of our analysis, we further classified the 55 cosmetic failures into eight subcategories: *incorrect color* (10), *incorrect cursor placement* (3), *content cut* (3), *image rendering issue* (4), *missing GUI element* (9), *incorrect orientation* (2), *incorrect placement* (4), and *incorrect text* (18). We provide details for each of these subcategories in our online appendix [24]. The screenshot on the left of Figure 3b illustrates an example of a cosmetic failure from the *incorrect placement* subcategory. This example is taken from a report [35] submitted for FIREFOX FOCUS, a browser app. In this example, the text `Show home screen tips` has additional padding w.r.t other text elements (e.g., `About Firefox Focus`) on the screen.

Our analysis of the navigation failures did not produce any further subcategories. The screenshot on the left of Figure 3c reports an example of a navigation failure. This failure was reported [36] for K-9 MAIL, an email client app. In this example, the user started setting up a new email account, went into the manual configuration settings, and, upon pressing the back button, the user was brought out of the app instead of the previous app screen. The screenshot in the right part of Figure 3c illustrates the correct app behavior where the user navigates to the sign-up screen after pressing the back button.

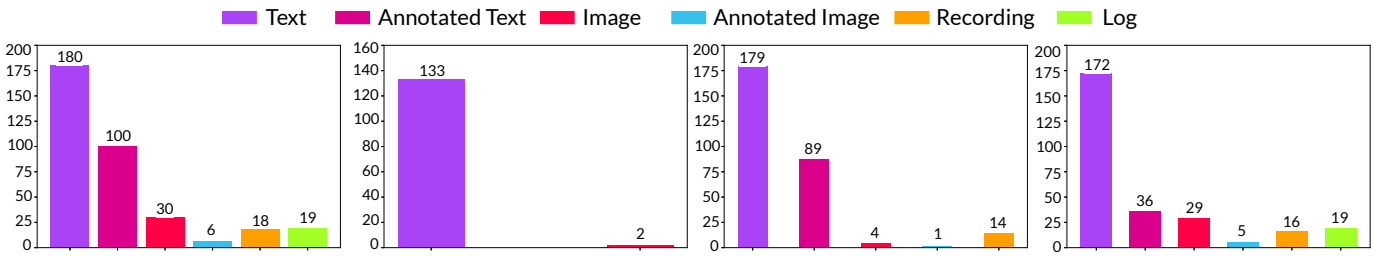
For the 50 failures leading to a crash, we identified two main subcategories, *immediate crash* (46) and *app freeze* (4). Immediate crash identifies failures in which the app crashes as soon an operation is performed in the app. App freeze includes failures in which the app first becomes unresponsive after an operation is performed in the app, and then the crash appears after a certain amount of time. The screenshot in the left portion of Figure 3d reports an example of an immediate crash failure reported [37] for FAMILY FINANCE, a household finance app. The right part of the Figure 3d reports the screen of the app after the bug in the app was fixed.

> **RQ_1 answer:** Our categorization identified four failure types: output (33%), cosmetic (31%), navigation (8%), and crash (28%). We also identified subcategories for output (2), cosmetic (8), and crash (2). Finally, the failure distribution does not differ dramatically when user- and developer-submitted reports are considered individually.

B. RQ_2: What information modalities are used to report the details contained in reproducible bug reports?

In our analysis of RQ2, we identified six modalities used to report bug information: *text*, *annotated text*, *image*, *annotated image*, *recording*, and *log*. Text identifies information reported in plain text. Annotated text is a sentence containing text within quotes or text with casing or capitalization [38], which represent either app inputs or GUI elements. Image identifies device screenshots. Annotated image is associated with device screenshots that have been edited to highlight parts of their content. Recording refers to any animated image or video providing a recording of the device screen. Finally, log identifies reporter-provided stack traces extracted from either app or system logs. Figure 4 reports the distribution of the modalities, for reports as a whole (Figure 4-a), the environment (Figure 4-b), S2Rs (Figure 4-c), and OB (Figure 4-d).

As expected, *text* is the most commonly used modality, with all 180 bug reports using text to convey some piece of information. *Annotated text* is the second most recurring modality and appeared in 100 bug reports. In our analysis, we also further categorized the annotated text modality into *annotated GUI text* and *annotated input text*. Annotated GUI text identifies bug reports in which the reporter used text within quotes or latter casing to identify an element in the GUI of the



a) Modalities for bug reports. b) Modalities for environment. c) Modalities for S2Rs. d) Modalities for OB.

Fig. 4: Reporting modalities for bug reports and bug report components.

relevant app. An example of this case appears in the bug report associated with Figure 3b in which the user wrote *"Show homescreen tips" is indented* in the report to describe the report's OB. The annotated input text subcategory contains cases in which the reporter provided a textual app input using text within quotes. An example of this case appears in a bug report [39] for K-9 MAIL, where the reporter mentioned *Add new email account with "foo@b.2"* as one of the S2Rs in the report. In total, we identified 100 bug reports with annotated text (85 annotated GUI text, six annotated input text, and nine in which both categories appeared). The remaining modalities, while less common, were still present and, in a large number of cases, provided information that would have been more cumbersome to convey otherwise. Among the bug reports considered, reporters used the image, annotated image, recording, and log modalities in 30, 6, 18, and 19 bug reports, respectively. Furthermore, image-based modalities (i.e., image, annotated image, and recording) appeared more frequently in user-submitted (35) than developer-submitted bug reports (14). Finally, we noticed a slight trend of increasing use of image data over the years, with image-based information being present in only 14% of reports in 2016 to 36% in 2019.

Figures 4-b, 4-c, and 4-d report the modalities used for specific sections of the bug reports. Figure 4-b reports the modalities used for the environment sections. The great majority of the reports (133) use the text modality to report environment information, and only a few use the image modality (2). Figure 4-c provides the modalities used for the S2Rs. Text is the most commonly used modality (present in 179 bug reports). Annotated text also appears in a considerable number of bug reports (89). The remaining modalities are less common but provide relevant information for reproducing the bug reports. Nineteen bug reports had multiple S2R modalities other than text or annotated text, 16 of these bug reports were user-submitted and 3 were developer-submitted. These 19 bug reports also used the recording (14), the image (4), and annotated image (1) modalities. Finally, Figure 4-d report the modalities used for the OB sections. Once more, the text modality is the most recurring one (172 cases). However, for OB, the image and recording modalities were used more frequently (29 and 16 cases, respectively) as compared to environment and S2Rs. Sixty bug reports had multiple OB modalities other than text or annotated text, 38 of these bug reports were user-submitted and 22 were developer-submitted. These 60 bug reports also used the image (29), the annotated image (5), recording (16), and log (19) modalities (with some bug reports having multiple modalities). Overall, user-submitted bug reports used reporting modalities other than text more frequently that developer-submitted bug reports.

Examining the relationship between reporting modalities and failure types, we found that bug reports with cosmetic and navigation failures have a higher proportion of cases in which the information is reported using image-based modalities as compared to output and crash failures. Specifically, 45% of the bug reports describing cosmetic failures and 43% of the bug reports discussing navigation failures use image-based modalities, while these modalities appear in only 16% and 18% of the bug reports describing crash and output failures, respectively. Focusing on specific bug reports sections, we find a similar result for OB descriptions. Additionally, the log modality was used exclusively to report the OB of bug reports describing crashes. These results highlight how certain modalities might be preferable particular failure types.

RQ_2 answer: Our categorization identified six main reporting modalities. Overall, text and annotated text are the most recurring modalities. Certain modalities occur more frequently when considering specific failure types, e.g., images for cosmetic and navigation failures.

C. *RQ_3: Do reproducible bug reports have missing information?*

Our analysis identified that 54 bug reports did not contain any environment information, one bug report did not have any S2Rs, and four bug reports did not contain OB information. (Missing information is computed with respect to the bug reports initially submitted and does not consider the information contained in their discussions, as that is the focus of RQ4.)

Although only one bug report did not have any S2Rs, 92.2% of the bug reports had at least one missing S2R. As mentioned in Section II, missing S2Rs include missing context S2Rs and missing inline S2Rs. 88.3% of bug reports had at least one missing context S2R and 37.7% of bug reports had at least one missing inline S2R. Figure 5 associates missing S2Rs to unmapped GUI actions. More precisely, for each bug report, the figure reports the percentage of unmapped GUI actions with respect to the number of GUI actions necessary to reproduce the report. The figure reports the percentage for missing S2Rs, missing context S2Rs, and missing inline S2Rs. The figure reveals that 75% of the bug reports have at least 20% unmapped GUI actions due to missing S2Rs. Across

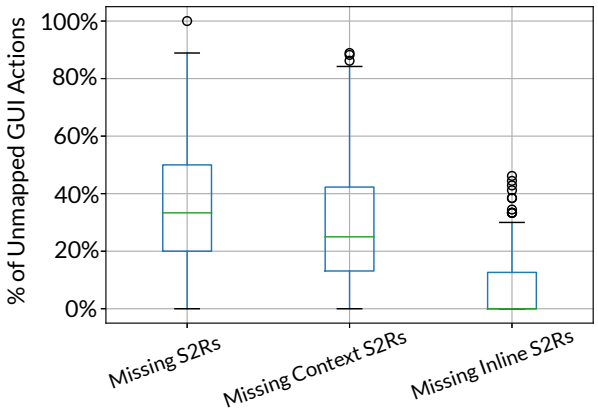


Fig. 5: Pct. of unmapped GUI actions due to missing S2Rs.

all bug reports, missing S2Rs led to 43.2% of GUI actions being unmapped. 33.4% of unmapped GUI actions are due to missing context S2Rs and 9.8% are due to missing inline S2Rs. These results illustrate that reproducing bug reports also requires inferring a large number of GUI actions that are not specified in the description of the bug reports.

Comparing missing S2Rs from user-submitted bug reports with respect to missing S2Rs from developer-submitted bug reports, users submitted reports that have a lower percentage of unmatched GUI actions due to missing context S2Rs (22.5%) with respect to developer-submitted reports (43.5%). This difference does not appear for unmatched GUI actions due to missing inline S2Rs (9.5% for user-submitted and 10% for developer-submitted bug reports). We did not observe a difference in missing information across failure types.

RQ_3 answer: The environment section of a bug report is the most likely to be missing from submitted bug reports among the sections considered. A large percentage of bug reports (92%) had at least one missing S2R. Missing S2Rs equate to 43.2% unmapped GUI actions necessary to reproduce the failures described in the reports.

D. RQ_4: Do discussion threads of reproducible bug reports contain helpful information for reproducing the reports?

To answer this RQ, we analyzed the discussions associated with the bug reports in our dataset and identified information added as part of the conversations that was relevant for reproducing the bugs. In total, 35 of the bug reports contained additional information detailing either the environment, the S2Rs, or the OB of the bug reports. Among these 35 bug reports, 25 were user-submitted and 10 were developer-submitted. Additionally, in 22 of the 35 bug reports, a developer explicitly requested for the information to be added to the discussion.

In the discussions, there were 20 instances of environment information added to the report, 11 instances of S2Rs, and 9 instances of OB. The sum of these numbers is higher than the total number of bug reports with additional information because some discussions (five in total, four with two messages and one with three) contained multiple messages that provided additional information. Although added information does not appear in a large number of cases, these results show that follow up conversations can be leveraged to reproduce reported bugs. Furthermore, considering the high number of reports with missing environment information and unmatched GUI actions identified in RQ_3, automated techniques can try to identify and automatically seek this information through iterative or interactive bug reproduction approaches.

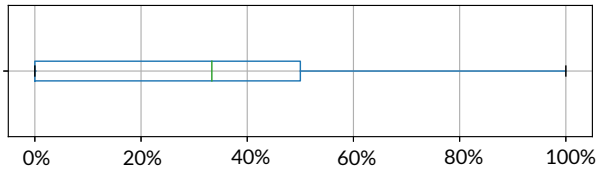


Fig. 6: Percentage of non-specific S2Rs by bug report.

Looking at different failure types, bug reports describing output failures were the ones with the highest number of added information in their discussions. Among the 35 bug reports with added information, 17 described output failures, 15 reported crashes, 2 described cosmetic failures, and 1 discussed a navigation failure.

RQ_4 answer: Among the bug reports considered, 35 had additional information relevant for reproducing the reports derived from follow-up, message-based discussions. In 22 reports, the information was explicitly requested by a developer. Finally, of the 35 reports, 20 had added environment info, 11 had added S2Rs, and 10 had added OB.

E. RQ_5: How specific is the information reported in reproducible bug reports?

When the environment was reported, the information could be directly mapped into actions for reproducing the failure. That is, it was possible to select the right app version, Android version, and device for reproducing the failure. In the case of OB, we had to look at the bug report discussion of six reports to better understand the problem associated with the reported failures, meaning that, in our analysis, the OB described in those bug reports was not specific enough for reproducing the failures. Considering S2Rs, 73.9% of the bug reports had at least one reported S2Rs that could not be directly mapped into a single GUI action but, instead, required multiple GUI actions. Based on the terminology defined in Section II, this means that those bug reports had at least one non-specific S2R. Figure 6 reports the percentage of non-specific S2Rs in each bug report of our dataset. Across all reports, the S2Rs section had an average of 36% of S2Rs that were non-specific. This results shows that there is the need to fill a gap to map S2Rs into corresponding GUI actions when reproducing reports.

Considering failure types, bug reports describing navigation failures had the highest average percentage of non-specific S2Rs (40%), while output failures had the lowest (34%). This result shows a minor difference in the specificity of S2Rs between reported failure types. There was also little difference in the average percentage of non-specific S2Rs reported by users (34.6%) and developers (35.8%).

RQ_5 answer: Environment and OB information was specific enough to reproduce reported failures in the great majority of cases. A large percentage of reports (73.9%) had at least one non-specific S2R, and the average percentage of non-specific S2Rs across all reports was 36%.

V. Discussion and Implications

1) New automated techniques are needed for understanding non-crashing oracles. Most existing automated bug reproduction approaches for mobile apps focus on reproducing bugs leading to a crash [13], [14]. This is likely because failures related to crashes are easier to recognize, for example through detection of a crash dialog, and thus detect when a a crashing bug has been reproduced. However, our analysis shows that more than 70% of the bug reports describe failures *other than crashes* and thus require more sophisticated oracle definitions and detection. For example, automated techniques for bug report reproduction might benefit from techniques that can define visual oracles using computer vision, such as detecting an incorrect color theme through color histogram analysis. Similarly, navigation failures might require analysis of statically computed program state graphs, to determine feasibility of navigation paths. Extending recent work on defining oracles through the derivation of program invariants (e.g., [40]) could further aid in oracle construction.

2) There is a need for automated multi-modal understanding of bug report information. Our analysis has illustrated that bug reports can mix multiple modalities of information together in form of text, images, and recordings, which capture disparate pieces of information about a given bug. However, most recent work on automated bug report reproduction and analysis only considers the textual modality [12], [13], [14]. Given the amount of prevalence of missing information, even in reproducible reports, revealed through our analysis of RQ_3, automated report analysis should strive to analyze *all* types of reported information for a more robust and complete analysis. As such, new techniques for multi-modal understanding of bugs is needed. For example, deep learning techniques that connect images and natural language (e.g., dense image captioning [41]) could be used to link textual information to visual information for more complete report analysis. Furthermore, in the case of S2Rs, automated techniques would also need to identify how to suitably order the information and this could be achieved by leveraging window transition graphs computed statically or dynamically from the apps [42], [43].

3) Techniques for inferring and mocking app environments are essential. Historically, Android app developers have struggled to reign-in issues related to the fragmented platform and device ecosystem. These issues also surface in bug reporting. As identified while analyzing the bug reports considered, it is possible for bugs to manifest under specific combinations of device and platform versions. Considering, that this information is not always present in submitted bug reports (missing in 30% of the cases), techniques that are able to infer, prioritize environmental settings (e.g., device and platform versions) are needed to help drive research on more advanced automated mobile bug report analysis techniques. Furthermore, considering that apps are released frequently [44], [45] and bug reports do not always contain the associated app version, it would be beneficial to automatically infer the version of the app associated with a bug report. This task could be done by automatically by mapping bug report information into GUI components or code entities in the app.

3) Reasoning about missing S2Rs is required. Our analysis illustrated that a large majority (92%) of our studied bug reports have at least one missing S2R. This represents a notable challenge for automated report analysis techniques which will likely need to infer this missing information in order to provide robust analyses. Current techniques do offer advanced solutions (i.e., they are based on random exploration) to help fill in certain missing gaps [13], [14], [12]. However, additional techniques are likely needed that allow for fine-grained inference of missing steps. For instance, future techniques could examine existing corpora of bug reports (such the artifacts associated with this research) and attempt to infer missing steps via patterns learned from a corpus of complete bug reports.

4) Handling non-specific S2Rs in bug report data is a major challenge. In addition to a high prevalence of missing S2Rs, our analysis also revealed that 36% of the S2Rs were mapped to multiple GUI actions. These S2Rs identified "high-level" operations, in which the actions or target GUI elements were not explicitly delineated. This situation represents a challenging reasoning problem for automated reproduction and report analysis techniques. Current techniques attempt to overcome such ambiguities through the use of ontological matching [13] or neural representations of text [12] in addition to random exploration. However, additional techniques for performing mapping of non-specific actions or targets are likely needed. For example future techniques may benefit from inferring descriptions of app controls or functionality through multi-modal image captioning models that allow for better mapping of text to runtime app information. Automated "repair" of ambiguous bug report steps based on patterns learned form well-formed sets of reproduction steps may also be a worthwhile direction of exploration. Additionally, S2R descriptions could be extracted from sequences of GUI actions in existing test cases and be mapped to S2Rs in bug reports to facilitate their reproduction.

In summary, the analysis performed in this paper has revealed several notable implications that impact future work on automated bug report reproduction, reporting, analysis, and management. We believe that future work will benefit from these findings and the potential new directions of research that they point towards.

VI. Threats to Validity

While we follow a systematic methodology in collecting, analyzing, and reporting our results, it is important to discuss the threats to validity of our study to provide a comprehensive view of our findings. In terms of external validity, our results

may not generalize to bugs for other Android apps. However, given the number, diversity, and popularity of our subject applications and reports, we believe our studied reports should be reasonably representative of Android bug reports as a whole. We considered the most recent dataset of reproducible bug reports (with non-crashing bugs) and extended the dataset to also include developer-submitted bug reports. This dataset includes apps that vary in terms of their size and category. An additional threat could be posed by the fact that we only used open source apps. However, the evaluation includes apps such as FIREFOX FOCUS and SIMPLENOTE, which have complex functionality and millions of installs. In terms of construct validity, our results might be affected by errors in the tools we used to perform our analyses. To mitigate this threat, we extensively tested our tools and multiple authors manually inspected the results. Finally, we also performed qualitative analyses, which could be impacted by divergent understanding among evaluators. To mitigate this threat, we used open coding based on negotiated agreement [31].

VII. RELATED WORK

A. General Studies on Bug Reports

Related work investigated bug report properties to better understand multiple activities characterizing the bug report management process [46], [47], [48], [49], [50], [51], [52], [53], [54]. Among different topics, this line of research analyzed bug report content, developers' and users' participation in bug report discussions, triaging, and bug fixing. A prominent study carried out by Bettenburg et al. [46] identified desired aspects that should be contained in a bug report. In follow-up work, Bettenburg et al. [47] also showed that duplicated bug reports contain some additional helpful information that could be used for bug triaging. Sahoo et al. [48] identified the main components necessary for bug reproduction by performing an empirical study. Some prior studies focused primarily on user-submitted bug reports. This line of research investigated how users typically communicate software problems [51], the usefulness of the provided information by power users [52], and user communitys' expectations [55]. In this paper, we investigated key aspects related to both user-submitted and developer-submitted Android bug reports. Furthermore, we focused on the aspects related to the reproduction of bug reports and specifically investigated how the bug report information relates to the information needed to reproduce the reports.

B. Bug Report Studies for Mobile Apps

Most of the initial studies on bug reports focused on desktop applications. However, because of smartphone apps' availability, usability, and popularity in the last decade, researchers have also started focusing on studying characteristics of bug reports for mobile apps. Zhou et al. [19] performed a study to understand the bug management between desktop and mobile software. Bhattacharya et al. [18] studied mobile bug reports and the bug-fixing process. Aljedaani et al. [56] compared the bug reports between Android and iOS. Zhang et al. [57] studied mobile apps bug reports, labeled those reports, and computed similarities with the previously labeled ones. In our study we reproduced bug reports, characterized the failures associated with the reports, analyzed the usefulness of the information provided in the reports, and categorized the reporting modalities. Previous studies also produced datasets of Android bugs with associated bug reports. Wendland et al. [20] created a dataset of reproducible, user-submitted bug reports. Su et al. [58] created a dataset of crashing bugs based on GitHub issues. Fazzini et al. [13] and Zhao et al. [14] also assembled a dataset of crashing bugs for their research on automated reproduction of bug reports. Compared to these datasets, to the best of our knowledge, this paper is the first to create and consider in its study a dataset of non-crashing and reproducible bug reports that contains both user-submitted and developer-submitted reports.

VIII. CONCLUSION

We presented an empirical study that characterized reproducible Android bug reports. Specifically, we manually reproduced 180 bug reports systematically mined from Android apps on GitHub and investigated how the information contained in the bug report relates to the task of reproducing the reports. Our analysis identified that reported failures can be grouped into four categories, three of which are not yet considered by existing automated reproduction techniques, reporters use different modalities to report the information relevant for reproducing failures, a large number of reports (74%) have at least one non-specific S2R (i.e., multiple GUI action are necessary to perform the operation described by the S2R), the great majority of reports (92%) do not provide all the S2Rs that are necessary to reproduce the reports, and bug report discussions can, in some cases (19%), provide additional information useful for the reproduction of the reports.

In future work, we first plan to present our findings to Android developers and then develop techniques to aid automated reproduction of bug reports. To support automated reproduction of bug reports, we first plan to define an approach that leverages natural language processing and computer vision techniques to automatically encode OB information into oracles and so aid reproduction of output, cosmetic, and navigation failures. Second, we plan to define a technique that combines S2Rs information reported using different modalities. Third, we plan to define a technique that leverages the information contained in existing test cases to help mapping non-specific S2Rs to corresponding GUI actions. Finally, we believe that additional studies into the reproduction of bug reports for software in other domains are needed and those studies could inform techniques for bug report management in those domains.

ACKNOWLEDGMENT

This work was partially supported by a gift from Facebook and the NSF CCF-2007246 & CCF-1955853 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep., 2002.

[2] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in *Proceedings of the 9th European Conference on Computer Systems*, ser. EuroSys'14, New York, NY, USA, 2014, pp. 18:1–18:15.

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

[4] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, ser. WCRE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 83–92. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2012.18

[5] "Android fragmentation statistics http://opensignal.com/reports/2014/android-fragmentation/," 2014.

[6] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. Gall, "Analyzing reviews and code of mobile apps for better release planning," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER'17, Feb. 2017, pp. 91–102.

[7] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'16, Seattle, WA, USA, 2016, pp. 499–510.

[8] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE'17, Piscataway, NJ, USA, 2017, pp. 106–117.

[9] F. Palomba, M. Linares-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*, pp. 143–162, 2018.

[10] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME'15, Sept 2015, pp. 291–300.

[11] S. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2015, pp. 429–440. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE.2015.89

[12] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the Quality of the Steps to Reproduce in Bug Reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2019, p. 86–96.

[13] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018, pp. 141–152.

[14] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "ReCDroid: Automatically Reproducing Android Application Crashes From Bug Reports," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 128–139.

[15] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 03, pp. 272–297, mar 2017.

[16] M. Pradel, V. Murali, R. Qian, M. Machalica, E. Meijer, and S. Chandra, "Scaffle: Bug localization on millions of files," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2020, p. 225–236.

[17] T. Zhang, W. Hu, X. Luo, and X. Ma, "A commit messages-based bug localization for android applications," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 04, pp. 457–487, 2019.

[18] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source android apps," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, 2013, pp. 133–143.

[19] B. Zhou, I. Neamtiu, and R. Gupta, "A cross-platform analysis of bugs and bug-fixing in open source projects: desktop vs. android vs. ios," *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015.

[20] T. Wendland, J. Sun, J. Mahmud, S. M. H. Mansur, S. Huang, K. Moran, J. Rubin, and M. Fazzini, "Andror2: A dataset of manually-reproduced bug reports for android apps," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 600–604, 2021.

[21] (2021, Oct.) Github. [Online]. Available: https://github.com

[22] (2021, Oct.) Google play. [Online]. Available: https://play.google.com

[23] "Mobile operating system market share worldwide," StatCounter, Technical Report, 2021.

[24] A. Authors, "Online appendix https://sites.google.com/view/2021bugreportingstudy/home," 2021.

[25] (2021, Oct.) Bug: Long pressing the amount input brings up qwerty keyboard. [Online]. Available: https://github.com/codinguser/gnucash-android/issues/689

[26] (2021, Jan.) About issues. [Online]. Available: https://docs.github.com/en/github/managing-your-work-on-github/about-issues

[27] (2021, Jan.) App manifest overview. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro

[28] (2021, Jan.) Uiautomator. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[29] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[30] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative data analysis: A methods sourcebook*. Sage publications, 2018.

[31] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013.

[32] E. R. Morrissey, "Sources of error in the coding of questionnaire data," *Sociological Methods & Research*, vol. 3, no. 2, pp. 209–232, 1974.

[33] (2018, Aug.) Can't open the add/remove medicine stock dialog. [Online]. Available: https://github.com/citiususc/calendula/issues/134

[34] (2021, Oct.) Tag text removal bug by using checkbox. [Online]. Available: https://github.com/federicoiosue/Omni-Notes/issues/634

[35] (2021, Oct.) Home screen tips toggle improperly indented. [Online]. Available: https://github.com/mozilla-mobile/focus-android/issues/3304

[36] (2021, Oct.) App closes on pressing back button in manual setup. [Online]. Available: https://github.com/k9mail/k-9/issues/3971

[37] (2021, Oct.) app crash when change view by in report section. [Online]. Available: https://github.com/zwieback/FamilyFinance/issues/1

[38] (2021, Oct.) Title case vs sentence case in ux writing. [Online]. Available: https://uxdesign.cc/title-case-vs-sentence-case-in-ux-writing-212087192261

[39] (2021, Oct.) Crashes with invalid format email address. [Online]. Available: https://github.com/k9mail/k-9/issues/3255

[40] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485533

[41] J. Johnson, A. Karpathy, and L. Fei-Fei, "Densecap: Fully convolutional localization networks for dense captioning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[42] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 115–127.

[43] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for Android," *International Journal of Automated Software Engineering*, vol. 25, no. 4, pp. 833–873, Dec. 2018.

[44] J. Gao, L. Li, T. F. Bissyandé, and J. Klein, "On the evolution of mobile app complexity," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019, pp. 200–209.

[45] 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*, vol. 21, no. 3, pp. 1346–1370, 2016.

[46] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2008, pp. 308–318.

[47] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM'08, 2008, pp. 337–345.

[48] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 485–494. [Online]. Available: https://doi.org/10.1145/1806799.1806870

[49] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information Needs in Bug Reports: Improving Cooperation Between Developers and Users," in *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'10)*, 2010, pp. 301–310.

[50] S. Davies and M. Roper, "What's in a bug report?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2652524.2652541

[51] A. J. Ko, B. A. Myers, and D. H. Chau, "A Linguistic Analysis of How People Describe Software Problems," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06)*, 2006, pp. 127–134.

[52] A. J. Ko and P. K. Chilana, "How power users help and hinder open bug reporting," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1665–1674. [Online]. Available: https://doi.org/10.1145/1753326.1753576

[53] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*, 2012, pp. 205–214.

[54] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 495–504. [Online]. Available: https://doi.org/10.1145/1806799.1806871

[55] P. K. Chilana, A. J. Ko, and J. O. Wobbrock, "Understanding expressions of unwanted behaviors in open bug reporting," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010, pp. 203–206.

[56] W. Aljedaani, M. Nagappan, B. Adams, and M. Godfrey, "A comparison of bugs across the ios and android platforms of two open source cross platform browser apps," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 76–86.

[57] T. Zhang, H. Li, Z. Xu, J. Liu, R. Huang, and Y. Shen, "Labelling issue reports in mobile apps," *IET Softw.*, vol. 13, pp. 528–542, 2019.

[58] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs." New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130.