Automated Bug Reproduction from User Reviews for Android Applications*

Shuyue Li
Xi'an Jiaotong University
lishuyue1221@stu.xjtu.edu.cn

Jiaqi Guo†
Xi'an Jiaotong University
jasperguo2013@stu.xjtu.edu.cn

Ming Fan
Xi'an Jiaotong University
mingfan@mail.xjtu.edu.cn

Jian-Guang Lou
Microsoft Research Asia
jlou@microsoft.com

Qinghua Zheng
Xi'an Jiaotong University
qhzheng@mail.xjtu.edu.cn

Ting Liu‡
Xi'an Jiaotong University
tingliu@mail.xjtu.edu.cn

ABSTRACT

Bug-related user reviews of mobile applications have negative influence on their reputation and competence, and thus these reviews are highly regarded by developers. Before bug fixing, developers need to manually reproduce the bugs reported in user reviews, which is an extremely time-consuming and tedious task. Hence, it is highly expected to automate this process. However, it is challenging to do so since user reviews are hard to understand and poorly informative for bug reproduction (especially lack of reproduction steps). In this paper, we propose RepRev to automatically **Rep**roduce Android application bugs from user **Rev**iews. Specifically, RepRev leverages natural language processing techniques to extract valuable information for bug reproduction. Then, it ranks GUI components by semantic similarity with the user review and dynamically searches on apps with a novel one-step exploration technique. To evaluate RepRev, we construct a benchmark including 63 crash-related user reviews from Google Play, which have been reproduced successfully by three graduate students. On this benchmark, RepRev presents comparable performance with humans, which successfully reproduces 44 user reviews in our benchmark (about 70%) with 432.2 seconds average time. We make the implementation of our approach publicly available, along with the artifacts and experimental data we used [4].

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Bug Reproduction, Android Applications, User Review Analysis

ACM Reference Format:
Shuyue Li, Jiaqi Guo, Ming Fan, Jian-Guang Lou, Qinghua Zheng, and Ting Liu. 2020. Automated Bug Reproduction from User Reviews for Android Applications. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381355

1 INTRODUCTION

Negative user reviews, especially bug-related ones, may influence the reputation of mobile applications and prevent users from downloading it. As shown in a survey, 5% of users will abandon the app after encountering bugs [9]. Thus, the app developers have to fix the bugs mentioned in user reviews as soon as possible. After receiving a bug-related user review, the first step is to compose test cases for bug reproduction. This process involves many human efforts, including natural language comprehension, app usage knowledge and test case composition. It is expected to relieve developers from this time-consuming and tedious task with some automated approaches.

There is much previous work [9, 10, 21] on mining or categorizing user reviews from app stores to help developers understand users' requests in time. Some work also tries to link bug-related user reviews to relevant code snippets [34] or stack traces [15]. These work demonstrates the possibility of utilizing user reviews in app quality assessment and assurance.

In this work, we aim to design an automated approach to reproduce bugs from user reviews. To this end, we first conduct an empirical study to understand: 1) how end-users describe crash bugs and 2) the challenges in reproducing crashes from user reviews. To answer the first question, we inspect 3,497 bug-related reviews (Dataset I) from Google Play, and summarize three characteristics of user reviews in perspective of bug reproduction (discussed in detail in Section 2). To answer the second question, we construct a benchmark including 63 bug-related user reviews, which have been reproduced manually. By examining these reviews, we find out the following challenges: 1) Extracting useful information from noisy user reviews. User reviews usually contain irrelevant contents to the crash, e.g., users' complaints. It is challenging to distinguish useful information for bug reproduction from the noise. 2) Leveraging limited useful information to reproduce crashes. In most cases, user reviews do not contain concrete reproduction steps and are

*This work was supported by National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61632015, 61772408, U1766215, 61721002, 61532015, 61833015, 61902306), Ministry of Education Innovation Research Team (IRT_17R86), and Project of China Knowledge Centre for Engineering Science and Technology.
†Work done during an internship at Microsoft Research Asia.
‡Corresponding author.

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7123-0/20/05…$15.00
https://doi.org/10.1145/3377813.3381355

poorly informative for bug reproduction. Considering the huge and unbounded GUI space of an app, it is challenging to effectively reproduce the crash with such limited useful information.

To tackle the challenges and achieve our goal, we propose RepRev to automatically **Rep**roduce Android application bugs from user **Rev**iews. Firstly, RepRev reduces the noise and extracts the most useful information from user reviews by exploiting several natural language processing techniques. Then, a guided search strategy with one-step exploration is designed to effectively trigger the bugs. The search strategy mainly consists of two techniques: 1) GUI component ranking. It ranks current GUI components on screen by its semantic similarity with bug description in the user review. 2) one-step exploration. It enhances information about a GUI component by interacting with it when the components on the current screen lack of enough information to guide the exploration. We implement RepRev and evaluate it on the benchmark aforementioned. It successfully reproduces 44 out of 63 user reviews (about 70%) with an average time of 432.2 seconds. The experimental results indicate that RepRev could be useful in assisting developers with automated bug reproduction from user reviews.

This paper makes the following contributions:

(1) Two datasets of user reviews are collected and open to the research community [4]. Through an empirical study on them, the characteristics of bug-related user reviews are summarized in the perspective of bug reproduction. These results can help the developers to understand how non-technical end-users describe the crashes they encountered.
(2) We propose a novel approach RepRev to automatically reproducing crashes with a depth-first search guided by user reviews. Specifically, RepRev extracts informative words from user reviews and uses these words to rank each GUI component in the search. A one-step exploration technique is proposed to further improve the efficiency of our search.
(3) RepRev is implemented and evaluated on the benchmark. It presents comparable performance as human developers, which successfully reproduces 44 user reviews in our benchmark (about 70%) with an average time of 432.2 seconds.

The rest of this paper is organized as follows. Section 2 presents an empirical study on bug-related user reviews. Section 3 introduces our approach in detail. In Section 4, we present the experimental results of RepRev and discuss some limitations in Section 5. Eventually, we introduce the related work about our research in Section 6 and conclude the paper in Section 7.

2 EMPIRICAL STUDY

In this section, we conduct an empirical study on bug-related user reviews to explore: (i) how users describe the crash bugs; and (ii) the challenges to reproduce crash bugs from user reviews.

2.1 Data Collection

Two datasets of bug-related user reviews are used in our empirical study. Dataset I is taken from the public resource [16], which contains 288,065 user reviews from Google Play, involving 395 open source apps. Following the data collection practice in recent work [21], we filter with keyword "crash" and collect 3,497 (about 1.2%) crash bug-related reviews in total.

Table 1: Details of Dataset II. #All, #Cra and #Rep are the number of all reviews, the crash-related reviews, and the reproduced crash-related reviews

App	Category	Downloads	Reviews		
			#All	#Cra	#Rep
Amaze	Tools	1,000,000+	680	19	2
Ankidroid	Education	1,000,000+	1124	19	19
Brave	Communication	10,000,000+	3400	28	0
Cgeo	Entertainment	1,000,000+	1529	9	1
Duckduckgo	Tools	1,000,000+	1640	15	0
Gnucash	Finance	100,000+	852	15	5
K9	Communication	5,000,000+	1002	29	3
Kiwix	Books & Reference	500,000+	521	25	9
Lightning	Communication	500,000+	365	21	3
Materialistic	News & Magazines	100,000+	463	15	3
Mysplash	Personalization	50,000+	272	23	7
Omninotes	Productivity	100,000+	520	5	0
Phonograph	Music & Audio	1,000,000+	1000	16	6
Redreader	News & Magazines	50,000+	560	14	4
Shuttle	Music & Audio	1,000,000+	680	12	1
Twidere	Social	500,000+	360	6	0
Wordpress	Productivity	10,000,000+	1000	27	0
Total			16,383	292	63

Dataset II, including 16,383 user reviews, is collected by ourselves. Table 1 shows the details of the apps we choose and the number of user reviews of each app. We choose two most popular open source apps from each category in Google Play. We then collect the reviews between 2015 and 2019 for each app and filter 292 bug-related (about 1.78%) user reviews with the same method on Dataset I. After that, we recruit three graduate students to reproduce the bugs manually. As a result, we successfully reproduce at least one bug for 17 apps. Thus, our benchmark is established on the user reviews from these 17 apps.

We collect open source apps since we need to inspect both the stack trace and source code to validate whether the crash is exactly the one reported in the user review. But RepRev can also apply to the apps without source code.

2.2 Characteristics of Bug-Related User Reviews

To understand how users describe the bugs they encounter, we carefully inspect the bug-related reviews. Specifically, three graduate students are invited to this task. Each participant independently read each review and label what characteristics are exhibited by the review. After labeling, we manually go through the labels to find out the conflicts. Once the conflict of a label is detected, the participants are asked to discuss the results with the aim of reaching a consensus. Afterwards, we update characteristics for each review according to the decisions taken during the discussion. The characteristics are summarized into three categories:

Characteristic 1: Containing A Lot of Noises. Noise words, such as misspelled, repetitive, and garbled characters, are frequently observed in user reviews since most of them are casually written by users with mobile interfaces. Moreover, users may describe not only the bugs they encounter but also their complaints, suggestions, and other topics like the price. These bug-irrelevant contents can also be regarded as noise. Considering Case 1 in Fig. 1, the user

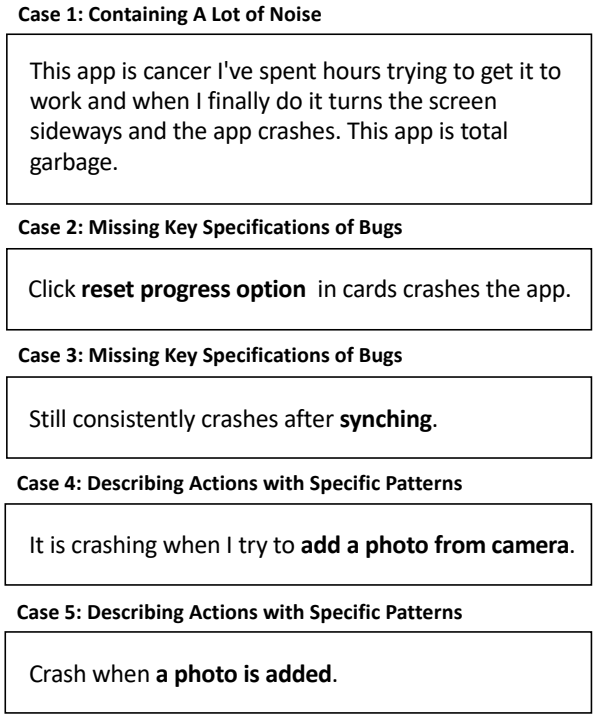

Figure 1: Examples of bug-related user reviews.

provides little and even ambiguous information about the crash but uses most of the words to express his/her subjective opinion.

Characteristic 2: Missing Key Specifications of Bugs. A detailed specification of actions that a user takes before a crash happens is critical for crash reproduction. However, we observe that there are only 0.29% of bug-related reviews describing the complete actions to reproduce a crash. About 20% of bug-related user reviews only complain about the poor user experience caused by crashes but do not provide any information about how crashes happen. In most cases, users tend to describe the last action that they take or the last state of the app before a crash happens. For example, in Case 2 of Fig. 1, the user only says that he/she clicks the reset progress option before a crash happens, but how to reach the reset progress option is never mentioned. In Case 3 of Fig. 1, the user complains that the app crashes after finishing a synchronization. Such characteristic significantly sets bug-related user reviews apart from bug reports in which detailed reproduction steps are required.

Additionally, due to the critical Android fragmentation problem, the device and Android OS on which the apps run are essential for crash reproduction. Unfortunately, there are only about 6% of bug-related user reviews reporting information about the device, Android OS, and OS version.

Characteristic 3: Describing Actions with Specific Patterns. We observe that the user tend to refer to the GUI component, when they describe the action. It can be observed in about 60% of bug-related user reviews in our dataset. Considering the Case 2 example shown in Fig. 1, there is a button with text "Reset Progress" shown in the screen, and the user refers to the button with its text. We also notice there are 65% of bug-related user reviews using specific grammar patterns like verb-object and passive nominal subject to describe actions, such as the Case 4 and Case 5 in Fig. 1.

2.3 Challenges in Reproducing Reviews

To better understand the challenges in reproducing crashes from user reviews, we try to manually reproduce crashes in the 292 bug-related reviews in Dataset II. Since users rarely describe the device, Android OS, and OS version (Characteristic 2), we choose the most recent 4 versions of the app before the date of the user review posted. In terms of OS, we only consider the official Android OS released by Google. Since the reviews we collect range between 2015 to 2019, we choose the 5 dominated OS versions in this period as our target, including Android 6.0, 7.0, 7.1, 8.0 and 8.1 [26]. As a result, we have 20 different configurations for each app. Three graduate students were recruited to manually reproduce crashes from these bug-related reviews. Each of the participants had more than half a year experience in Android development. For each bug-related user review, we allow a participant to explore a configuration in 30 minutes to reproduce the crash. If the participant cannot reproduce the crash in all configurations, we consider the crash reported in the review as unreproducible. Overall, the reproduction of the 292 bug-related user reviews costs us about 2 man-months.

As a result, 63 out of 292 (23.2%) bug-related user reviews are successfully reproduced by at least one participant in one configuration, as shown in Table 1. We notice that the success rate is not very high and some failures may be due to hardware specific issues or other unknown reasons. Thus, we mainly focus on these 63 reviews to summarize the challenges of user review reproduction:

Challenge 1: Extracting Useful Information from Noisy User Reviews. Intuitively, to reproduce the crash reported in a user review, we need to extract the actions that users take and the states of the app before it crashes. In the remainder of this paper, we define the actions and the states as useful information for crash reproduction. However, as we have summarized in the Characteristic 1, bug-related user reviews have a lot of noises, which inevitably makes the extraction of useful information challenging.

Challenge 2: Leveraging limited useful information to reproduce bugs. As discussed in Characteristic 2, in most cases, users only describe the last action they take or the last state of an app before a crash happens. In other words, the useful information in user reviews that can be leveraged for crash reproduction is limited. Considering the huge and unbounded GUI space of an app, how to effectively leverage such limited useful information for reproduction becomes challenging. Also, it is well-known that there exists a lexical gap between natural language and software artifacts [18, 27, 32, 34]. In this crash reproduction scenario, the lexical gap also exists between user reviews and the artifacts of an app, as users do not have a professional Android development background. The lexical gap makes the task more challenging.

3 APPROACH

In this section, we present our approach, named RepRev, to reproduce crashes from bug-related user reviews. RepRev takes a bug-related user review as input, and outputs a test case script that can reproduce the bug and its related stack trace. In the following, we first provide an overview of RepRev and elaborate on

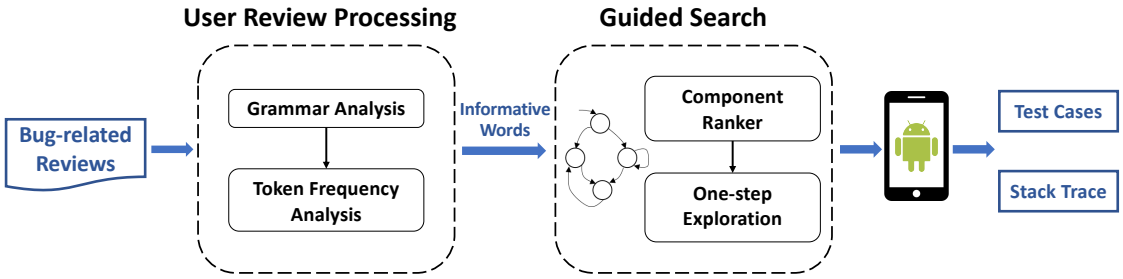

Figure 2: Workflow of RepRev

how RepRev addresses the two challenges discussed in section 2. Then, we illustrate the techniques and algorithms in detail.

3.1 Overview

The workflow of RepRev is shown in Fig. 2. Taken a bug-related user review as input, RepRev operates in two phases to reproduce a crash: *user review processing* and *guided search*.

In the user review processing phase, a two-pronged approach is designed to address the first Challenge. **1) Grammar Analysis.** There are grammar patterns on how the bugs are described (Characteristic 3). Therefore, we leverage the dependency parsing technique to identify typical patterns. However, the state-of-the-art dependency parsing technique is not robust to noisy inputs like user reviews. To this end, we further design a novel normalization method to clean the user review before adopting dependency parsing techniques. Details are presented in section 3.2. **2) Token Frequency Analysis.** Solely relying on grammar pattern is not enough, as we may miss some informative words due to the limitation of dependency parsing techniques. Hence, we design a simple yet effective token frequency analysis technique to identify more informative words. The outputs of these two steps are merged as a set of words, serving as guidance to reproduce the crash.

In the second phase, a guided search algorithm is proposed to leverage the extracted information from a user review to reproduce the bugs described in user review. The algorithm is built upon the depth-first search strategy, which in essence mimics how users interact with an app [8, 33]. At each step of the search, unlike existing test input generation techniques that choose the next action randomly or with the aim at pursuing a high code coverage [13, 25, 30], we choose the next action with the guidance of extracted information from the user review. That is, the informative words extracted from the last phase are used to rank GUI components, and we first interact with those GUI components with higher rankings. To rank GUI components with the extracted informative words, we propose a method that uses the word2vec technique [26, 28] to bridge the lexical gap between the user review and GUI components. Additionally, as we have shown in Characteristic 2, bug-related user reviews may only describe the last action or last state of an app before a crash happens. Such limited useful information makes it difficult to correctly rank GUI components in the very first stage of the search. In these circumstances, we propose a novel one-step exploration technique to enhance the information of each GUI component before we decide the next action.

Algorithm 1: User Review Preprocess

Input : User Review r, Target App **A**,
All Reviews of Target App R
Output: Informative Words W

```
// Grammar Analysis
1 let n = normalize(r, A);
2 let patterns = parseDependencyTree(n);
3 let W1 = extract(patterns) ;
// Token Frequency Analysis
4 let f = analyzeFrequency(r, R);
5 let W2 = filterWithThreshold(f);
6 W = W1 ∪ W2;
7 return W;
```

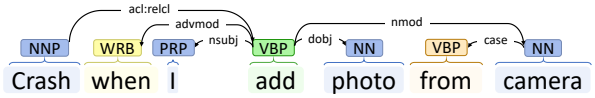

Figure 3: Example of Dependency Tree.

To further improve the search efficiency, we design two optimization strategies: 1) state abstraction which avoids searching equivalent GUIs, and 2) loop breaking which avoids too many invalid searches in the same window. The search terminates when the bug reported in the user review is reproduced or the time limit is reached. If RepRev successfully reproduces a bug, it outputs the test case to trigger the bug and the corresponding stack trace.

3.2 User Review Processing

In this phase, RepRev takes a user review as input, and extracts informative words from the user review. The process algorithm is outlined in Algorithm 1. There are two steps in this phase: Grammar Analysis (line 1-3) and Token Frequency Analysis (line 4-5).

3.2.1 Grammar Analysis. As we have discussed in Section 2, user reviews are rather noisy (Characteristic 1), which inevitably hurts the performance of any dependency parsing technique. In order to overcome this, we normalize the input user review (line 1), before adopting a dependency parsing technique. Specifically, given an input app, we first extract a complete list of its pre-defined GUI components text from the UI configuration file of the app, i.e.,

res/values/public.xml. Note that when the source code of the app is not available, the UI configuration file can also be obtained with Android reverse engineering tools, e.g., apktool [3]. Then, for each GUI component, we check whether it is mentioned in the user review. If so, we replace the words in the user review with a single word "button". It is important to note that such a replacement would not change the grammatical structure of the user review. Instead, it facilitates the dependency parsing techniques to identify grammar patterns in the user review accurately. In addition, we also try to identify user's actions, such as *click*, *tap* and *scroll*. We manually establish a corpus of commonly used action words by examining hundreds of user reviews. During normalization, we also enumerate each action keyword in the corpus and check whether it is mentioned in the user review. If so, we replace it with its standard form in the corpus. In this way, both the actions and GUI components are normalized to facilitate further analysis. Take a user review saying "crash when access /r/all" as an example, where "/r/all" is the name of a pre-defined component on the app. After normalization, it is transformed into "crash when *click button*". Obviously, it is much easier for a dependency parser to parse the normalized version than the original one.

After the normalization of the user review, we identify specific grammar patterns in it with a state-of-the-art dependency parsing technique, and extract informative words from the patterns (line 2-3). Firstly, we parse the user review into a dependency tree. The dependency tree depicts the grammatical structure of a sentence and defines the relation between each pair of words. Fig. 3 shows a dependency tree of a user review, where a directed edge between two words indicates that the source word depends on the target word with a specific relation. We extract the two grammar patterns that we discuss in Characteristic 3: verb-object structure and passive nominal subject structure. In addition to these two patterns, we also observe that some informative words are represented by the modifier relation. Considering the example in Fig. 3, while "add" and "photo" can be extracted by verb-object structure, another informative word "camera" needs to be extracted by its modifier relation with the word "add". To obtain informative words altogether, we also extract these modifiers relations in our approach. Finally, we collect all the words that are involved in the extracted patterns.

While this method works for most user reviews, there are still some cases where informative words are missed. For example, in Case 3 in Fig 1, the word "synching" cannot be extracted by this method. To overcome this limitation, we further design the following token frequency analysis step.

3.2.2 Token Frequency Analysis. Intuitively, words that frequently occur in an app's user reviews but rarely occur in the other apps are likely to be valuable. We design a statistical method to identify informative words in the bug-related user review based on this intuition, which is inspired by the term frequency inverse-document frequency (TF-IDF) technique [29, 32]. Specifically, we consider all user reviews of an app as a document (including non-bug-related user reviews). Given the data we collect in Section 2, we establish a collection of documents. Taking a bug-related user review as input, we assign each word in the review a score with the following equation:

$$Score(w) = \frac{freq_{doc}(w)}{freq_{corpus}(w)}, \quad (1)$$

where $freq_{doc}$ and $freq_{corpus}$ denote the frequency of word w in the document (all the user reviews of an app) and in the corpus (a collection of documents), respectively. For those words that solely appear in an app's user reviews, we assign their scores as 1. Finally, those words that have scores larger than a threshold are considered informative. Considering the Case 3 in Fig 1, the word "synching" refers to a synchronization function of the app, and it is frequently mentioned in the user reviews of this app. As a result, this word has the highest score in the review and consequently, it is regarded as an informative word.

The outputs of the Grammar Analysis and Token Frequency Analysis are merged, serving as guidance in the next phase.

Algorithm 2: Guided Search with One-step Exploration

```
1  Procedure GuidedSearch(S, W)
       // S: a GUI; W: informative words
2      let C = ExtractComponents(S);
3      let Scores = CalculateScore(C, W);
4      C = Rank (C, Scores);
5      if Scores are all zeros then
6          C = OneStepExploration (C, S);
7          Scores = CalculateScore(C, W);
8          C = Rank (C, Scores);
9      foreach GUI component c ∈ C do
10         let act = GenerateAction (c);
11         let newS = Execute (act);
12         if A crashes then
13             return stack trace;
14         if newS is seen before then
15             continue;
16         GuidedSearch(newS, W);
17         RestoreGUI(S);
18     return none;
19 Procedure OneStepExploration(C, S)
20     foreach GUI component c ∈ C do
21         let act = GenerateAction (c);
22         let newS = Execute(act);
23         if newS = S then
24             continue;
25         c = UpdateRepresentation(c, newS);
26         NtopC = getNewRanking (S, NS);
27         RestoreGUI(S);
28     return C;
```

3.3 Guided Search

In the guided search phase, given the extracted informative words as input, we propose a guided search algorithm to effectively explore

the GUI space of the app to reproduce a crash. The algorithm is outlined in Algorithm 2.

3.3.1 Depth-First Search. The guided search algorithm is built on the depth-first strategy, which aims to explore an app's functionalities thoroughly. The depth-first search starts from the initial page of an app. At each step of the search, we extract the GUI components on current GUI. Then, the components will be recursively explored in a depth-first manner through interacting with a component which makes transition to a new GUI. The process ends when we cannot reach any new GUI. To this end, the search can systematically explore the app functionalities. Interested readers can refer to [8] for more details about the depth-first search strategy.

3.3.2 Rank GUI Components by User Review. The original depth-first search only randomly selects a GUI component to explore at each step of the search, which is not efficient. In our scenario, user reviews can be leveraged to guide the search and improve the search efficiency, because they provide useful information for crash reproduction. We propose to rank GUI components based on the extracted informative words from a user review and explore those GUI components with higher rankings first.

To rank GUI components, we calculate a score for each of them with respect to the informative words. Typically, a GUI component has 3 attributes: *Text*, *Description* and *Resource Id*. We represent a GUI component as a set of words. Since the *Text* attribute is shown on the screen, the users usually mention a GUI component with its *Text* attribute. If the *Text* attribute of it is not empty, all words in *Text* are added into the word set to present this GUI component; else all words in both *Description* and *Resource Id* are added. Then, the score of a GUI component is calculated by summing over its similarity scores to each informative word:

$$Score(c) = \sum_{w_i \in w} Similarity(w_i, c), \tag{2}$$

where c denotes the word set of a GUI component, w denotes the set of informative words of a user review, and $Similarity(w_i, c)$ measures the similarity between an informative word and the GUI component. To bridge the lexical gap between user reviews and GUI components, we leverage the word2vec techniques to measure the similarity between an informative word and a GUI component. In addition, we observe that the word sets of GUI components are usually short in length and different GUI components share many words. Thus, the more frequently a word appears in different components, the less capable it is of distinguishing a GUI component from others. Based on this observation, we measure the similarity between an informative word and a GUI component as follows:

$$Similarity(w_i, c) = \max_{w_j \in c}\left(\frac{cos(\mathbf{w_i}, \mathbf{w_j}))}{freq_{app}(w_j)}\right), \tag{3}$$

where w_j denotes the j-th word in the GUI component, $\mathbf{w_i}$ and $\mathbf{w_j}$ denote the word embeddings of word w_i and w_j, $freq_{app}$ denotes the frequency of w_j appearing in all the GUI components of the app. Moreover, we set a threshold (0.6) for $Similarity$ and set it as 0 when it is lower than the threshold.

3.3.3 One-Step Exploration. As we have discussed in Section 2, bug-related user reviews usually only describe the last action or the last state before a crash happens. Such limited information makes it difficult to rank GUI components accurately in the very early stage of the search. Take a bug-related user review from an app Ankidroid [2] as an example, there are partial steps to reproduce the crash reported here, which we present in Fig. 4. In the left GUI in Fig. 4, the scores of GUI components calculated with equation 2 are all zeros, as none of them are similar to the information from user review ("add a photo from camera"). In these circumstances, instead of randomly interacting with GUI components, we design a novel one-step exploration technique to enhance the information of each GUI component in the current GUI and decide on the next action based on the enhanced information.

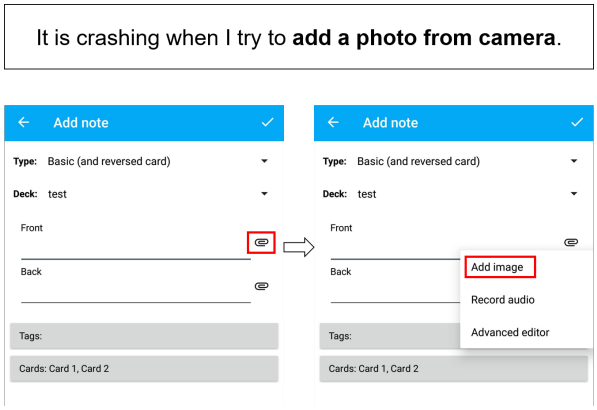

Figure 4: A user review of Ankidroid and partial steps (the second one and the third one) to reproduce the crash.

Specifically, at each step of the search, when the scores of GUI components C are all zeros, RepRev interacts with each of them in an arbitrary order. If an interaction of a GUI component c leads to a previously unseen GUI, we augment the representation of c with the representations of all the GUI components C_{new} in the new GUI. Then, we re-calculate the score of the GUI component c based on its updated representation. If an interaction does not lead to a previously unseen GUI, we keep the score of the GUI component unchanged. Having interacted with all the GUI components C, we re-rank them based on their updated scores. Considering the example in Fig. 4, the GUI component outlined in red on the left, is ranked to the first one through the one-step exploration.

3.3.4 Optimizations. The GUI space of an Android app is known to be huge and unbounded. To improve the search efficiency, we further propose two optimization techniques, presented as follows.
GUI Abstraction. We propose the GUI abstraction technique to merge similar GUIs, aiming to reduce the number of different GUIs. First, following the practice in [30], RepRev omits minor changes in GUI [30], including content change of TextViews/EditTexts and UI property changes (e.g., the checked property of RatioButtons and CheckBoxs). Second, RepRev conducts abstraction on dynamic list views, one of the most common components in Android applications. A dynamic list view is a set of items containing texts or images, and users are allowed to add items or delete items on the fly. Heuristically, users interact with the items within a dynamic list view in the same way. Hence, list views are abstracted to containing only one item within them whenever they are not empty.

Loop Breaking in same *window*. Typically, an Android app comprises lots of *Activities*. During execution, each activity is assigned a *window* in which the activity draws its GUI, and different GUIs can be drawn in the same *window*. A typical example is music player applications. Usually, when playing a song and jumping to the next, the view hierarchy of the GUI screen will update, but the *window* object stays the same (if the *window stack* is not refreshed at the moment). Since the dynamic exploration identifies each new screen by its view hierarchy, it will create an infinite loop on the same *window* (i.e., the same scenario), and dramatically reduce the search efficiency. In our experience, staying in the same *window* for five actions is long enough in normal usage on apps. Hence, when the search loops in the same window over five times, we iteratively interact with components on the screen (and back button) until jumping out of the window. Specifically, we differentiate *windows* according to their addresses in *window* stack which can be obtained with a command *adb dumpsys top activity*.

Additionally, we demote a component that leads to a side menu. Side menus, officially named navigation drawers [14], are used for quick navigation between top-level functionalities. To avoid too many transitions between different functionalities, we delay the interaction with side menus.

4 EVALUATION

We implement RepRev as a publicly available tool [4]. In the user review processing phase, the Stanford CoreNLP framework [24] is used to parse user reviews into dependency trees. The threshold in the token frequency analysis is set to 0.3. In the guided search phase, the python wrapper of Android uiautomator2 [5] is applied to inject click, scroll and input events into apps. Word embedding is implemented in Python with Gensim library [1] and is pre-trained on the Google News dataset [12]. The dimension of word embedding is set to 100. The ADB command is used to dump the stack traces (*adb logcat*) and to monitor the state of emulators. We conduct evaluations on Android emulators running on Ubuntu 14.04. To evaluate the effectiveness of RepRev, we explore the following research questions:

RQ1. Can RepRev reproduce crashes from user reviews?
RQ2. How efficient is RepRev for crash reproduction?
RQ3. How effective are the techniques we proposed in RepRev?

4.1 RQ1: Review Reproduction

To answer the RQ1, we evaluate RepRev on user reviews from Dataset II. Given a user review in the benchmark, RepRev runs on 20 configurations (aforementioned in Section 2.3) of the app until the crash is reproduced, or the time limit is reached. Then, we manually examine the stack trace and the action sequences of a crash to validate whether it is the one reported in the user review. In practice, we first set up the app, e.g., user authentication, before running RepRev. The time limit is set as two hours for each configuration.

As shown in Table 2, RepRev successfully reproduces 44 crashes posted in the user reviews out of the total 63 ones(about 70%). It is noticed that RepRev reproduces all 16 reviews that could be reproduced with no more than 2 actions, 24 out of 38 reviews (63%) that with 3 or 4 actions, and 4 out of 9 reviews (44%) that need more

Table 2: Experiment Result. "#Word" is the number of words in user review. "#Actions" indicates the number of actions to reproduce the bug. "Time_Man" and "Time_Rev" are the time to reproduce the review by human or RepRev. "<1" means App is crashed without any action, and "Failed" means RepRev fails to reproduce the review.

App	#Words	#Actions	Time_Man(s)	Time_Rev(s)
Amaze	33	3	100.0	174.5
Amaze	12	3	106.7	192.3
Ankidroid	45	4	186.7	275.2
Ankidroid	17	5	316.7	56.1
Ankidroid	39	4	160.0	90.4
Ankidroid	11	4	376.7	1103.4
Ankidroid	9	4	220.0	992.3
Ankidroid	5	1	170.0	8.7
Ankidroid	21	4	126.7	54.42
Ankidroid	12	4	136.7	68.5
Ankidroid	8	4	126.7	35.5
Ankidroid	7	4	120.0	44.4
Ankidroid	29	2	200.0	1205.4
Ankidroid	20	2	160.0	135.2
Ankidroid	21	4	90.0	8.7
Ankidroid	14	4	266.3	Failed
Ankidroid	8	3	195.3	Failed
Ankidroid	15	6	260.7	Failed
Ankidroid	10	4	253.3	88.4
Ankidroid	10	4	110.0	64.2
Ankidroid	13	2	116.7	58.87
Cgeo	67	2	186.7	350.4
Gnucash	6	0	<1	<1
Gnucash	6	0	<1	<1
Gnucash	5	0	<1	<1
Gnucash	20	6	263.0	Failed
Gnucash	12	3	250.3	Failed
K9	12	3	330.3	Failed
K9	19	3	255.0	Failed
K9	13	3	230.3	Failed
Kiwix	45	3	310.0	1430.2
Kiwix	15	3	166.7	1113.2
Kiwix	20	5	166.7	1223.1
Kiwix	147	5	286.7	1102.3
Kiwix	6	3	120.0	32.3
Kiwix	15	3	296.7	1124.5
Kiwix	213	2	483.3	979.4
Kiwix	60	2	266.7	668.4
Kiwix	58	0	<1	<1
Lightning	67	3	580.7	Failed
Lightning	18	3	610.0	Failed
Lightning	15	3	670.3	Failed
Materialistic	12	4	310.0	Failed
Materialistic	12	4	322.3	Failed
Materialistic	54	4	240.3	Failed
Mysplash	16	3	90.0	625.3
Mysplash	7	3	110.0	612.2
Mysplash	14	3	120.0	620.4
Mysplash	5	3	113.3	632.2
Mysplash	7	3	156.7	714.8
Mysplash	14	3	183.3	Failed
Mysplash	12	4	188.3	Failed
Phonograph	54	4	96.7	320.8
Phonograph	21	5	226.7	120.2
Phonograph	24	0	<1	<1
Phonograph	27	5	312.3	Failed
Phonograph	9	5	412.0	Failed
Phonograph	5	5	355.3	Failed
Redreader	23	1	43.3	15.2
Redreader	10	1	48.3	13.4
Redreader	17	1	60.0	59.7
Redreader	24	1	50.0	14.2
Shuttle	13	3	133.3	77.2
Aver.	27.7	3.4	221.5	423.2

than 4 actions. To understand the source of failures, we further analyze all the failure cases and summarize their causes:

Comprehension of natural language in user review. There are 8 cases where our approach missed correct event sequences because of its limitation on natural language comprehension. For example, a review says "...crash when seeing the transaction detail if you have more then 4 level of sub account"[1]. It seems easy for human to perceive the description "more than 4 level", but RepRev is unable to generate the correct actions. Our approach fails on these highly abstract descriptions resulting from its lack of understanding of natural language. Another example is a review from a web browser application saying that "This app is useless wait crashes every time I load a heavy website...". To reproduce the crash, human tried many "heavy" websites, e.g., the website with many images. However, generating this kind of input is non-trivial for our approach without some domain knowledge. Actually, this case is out of the scope of our work in this paper.

Unsupported Operations. There are two cases where it requires some actions outside the app to reproduce the crash. Specifically, in these two cases, it is required to choose local files from the file browser in Android system but our approach does not support these system actions. There are three cases caused by the limitation of the depth-first search. The depth-first search only explores a functionality of an app once. However, if a bug can only be reproduced with repetitive actions, the depth-first search can never reproduce it. For example, a user review says that "My app gets crashed when trying to record audio". To reproduce the crash, two conditions need to be satisfied: (1) tab the Play button right after Record button, (2) repeat the "record-and-play" action twice. This example requires using the "record-and-replay" functionality twice.

Other causes. Non-deterministic bugs (3 cases). The non-deterministic bugs are common in Android applications and cause some failures of RepRev. We inspect the exploration process of these cases and find that RepRev generates correct event sequences, but the crash does not happen in our experiment for unknown environmental reasons, e.g., network state, hardware, etc. Additionally, we run these cases another two times but still fail to reproduce the crash.

Time interval (3 cases). There are two cases where the precise time interval is critical for bug reproduction. For example, a user review says that "Recently app crashes after download is finished". According to our manual reproduction experience, users need to wait on the same GUI until the download has finished in order to trigger the crash. In our approach, we set fixed time interval between actions so we fail to capture the precise timing to take the next action. Actually, the information about the time interval between actions is hardly mentioned in user reviews. This problem can be mitigated by defining some heuristic rules to predict time intervals; for example, if we recognize a download scenario, RepRev needs to wait until the download finishes.

4.2 RQ2: Efficiency

To evaluate the efficiency of RepRev in reproducing crashes, we select 44 successfully reproduced reviews by RepRev. To establish the baseline, we also record the time that human participants spend on reproducing each crash.

[1]The user misspelled. "then" should be "than".

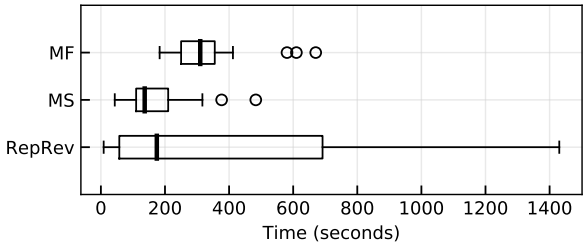

Figure 5: Reproduction time of manual process (*MF*: failure cases of RepRev, *MS*: successful cases of RepRev) and RepRev.

Table 3: The ablation study results. "#Reproduced" indicates the number of successfully reproduced reviews. "Averaged Time" indicates the averaged time to reproduce a review.

Approach	#Reproduced	Averaged Time(s)
RepRev	44	432.2
$RepRev_1$	39	547.7
$RepRev_2$	41	469.4

As shown in Table 2, RepRev is able to reproduce a crash in 423.2 seconds on average, while the averaged manual reproduction time is 169.4 seconds. Such results are expected since RepRev needs to explore a larger number of actions during reproduction. Hopefully, RepRev reproduces one-touch crash (22.2 seconds on average) faster than human developers (74.3 seconds) since it identifies the bug-related component more quickly (one-touch crash refers to the crash triggered by only one action). By the way, we find that humans also spend more time on the crashes that RepRev fails to reproduce. As shown in Fig. 5, manual reproduction time on failure cases of RepRev is larger than that on the successful cases of RepRev. It means that these cases are complex and need deep understanding of user reviews.

Although RepRev takes more time than human in average, it is still useful to developers. First, RepRev is fully automated and can relieve developers from the painstaking and tedious process of bug reproduction, especially when the correct configuration of the app is unknown. Secondly, RepRev can be integrated with a watchdog in app stores (e.g., Google Play, App Store). Whenever a bug-related review is posted, the watchdog can detect it and launch RepRev to generate a test case for reproducing the reported bug. Then, the user reviews along with the test case can be delivered to developers.

4.3 RQ3: Effectiveness

We conduct an ablation study on RepRev to evaluate the effectiveness of two techniques: one-step exploration and user review processing. When the one-step exploration is ablated, we randomly interacts with a GUI component when the scores are all zeros. We denote this variant of RepRev as $RepRev_1$. When the user review processing is ablated, RepRev simply tokenizes a user review, and all the resultant words serve as guidance words in the next guided search phase. We denote this variant as $RepRev_2$.

$RepRev_1$ and $RepRev_2$ are evaluated on our benchmark of user reviews. As shown in Table 3,$RepRev_1$ only reproduces 39 crashes. The cause for the five additional failures compared with RepRev (44) is that $RepRev_1$ takes more time in exploring the functionalities that are irrelevant to the crash when the scores are all zeros, and thus cannot reproduce the crash before the time limit is reached. Moreover, the average time for reproduction increases by 42.65% compared with that of RepRev. This result demonstrates the effectiveness of the one-step exploration technique in enhancing the representations of GUI components and guiding the exploration.

In terms of $RepRev_2$, it only reproduces 41 crashes and the average reproduction time increases by 30.0% compared with RepRev. One additional failure is caused by incorrect tokenization of user review. Specifically, a user reports that a crash is caused by "using OSM: Cyclemap", where "OSM: Cyclemap" is the name of a GUI component in the app. With a conventional tokenizer, the phrase is tokenized as three words, namely "OSM", ":", and "Cyclemap", which can mislead $RepRev_2$ to other components with similar names (e.g., "Cyclemap", "Cyclemap allowed", "of cyclemap"). Thus, $RepRev_2$ spends much time in exploring the irrelevant components and fail to reproduce the crash within time limit. With the user review processing technique, the keyword "OSM:Cyclemap" is retained (since we identify the name of components before any further processing) and guides the exploration effectively. In another two failures, the reviews contain lots of words (147 and 213 words, respectively) and most of the words describe bug-irrelevant contents. For example, a user mentions four irrelevant functionalities (i.e., "language", "settings", "category" and "the online tab") in one review, which misguides $RepRev_2$. The result indicates that the user review processing technique can improve the effectiveness of our approach.

5 DISCUSSION

Limitations & Future Work. While RepRev is effective in reproducing bugs reported in user reviews, it is still tied to certain limitations. First, RepRev currently only supports three basic types of interactions: click, swipe and text input. However, there exists some bugs where sophisticated GUI interactions like drag, system events interactions like incoming messages, and interactions with system applications are required. Given the general design of RepRev, we believe that it can be extended to support these interactions. Second, RepRev does not have an in-depth understanding of the given user review and the app under test. It would cause several failures that were discussed in Section 4.1. To mitigate this problem, the advanced techniques for Natural Language Processing based on deep learning can be leveraged to extract the useful information more accurately. In addition, recently proposed approaches to automatically recognize app's usage scenarios can also be adopted to improve the understanding of the app under test [20]. We leave the exploration of these techniques to improve RepRev as our future work. Third, RepRev currently focuses on reproducing crash bugs. In order to reproduce those non-crash bugs, we are suffering from the missing test oracles problem. One promising solution could be incorporating with developers to specify test oracles for bug-related reviews and running RepRev to reproduce the bug.

Threats to Validity. The primary internal threat arises from the empirical study in Section 2. It requires participants to summarize the characteristics of bug-related user reviews and challenges in automated bug reproduction. To mitigate this threat, we first ask each participant to work independently. Then, we open a discussion with the participants to reach a consensus on the characteristics and the challenges. After that, we re-label the bug-related user reviews based on the decisions made during discussion. When it comes to external threats which concern about the generalization of evaluation results, we only evaluate RepRev on 63 bug-related user reviews. This limitation is an artifact of complexity in constructing the benchmark, as we have to manually reproduce bugs from user reviews. To mitigate this threat, we sample popular open source Android apps. Most of them are downloaded over one million times. To this end, RepRev should generalize to the other Android apps. In terms of the construct validity, to make the evaluation and study replicable, we make RepRev's source code, Dataset I, Dataset II and all the experimental results publicly available [4].

6 RELATED WORK

Mining User Reviews on App Stores. Many approaches have been proposed to mine and categorize user reviews in app stores to help developers know better about user requests or complaints in time [9, 10, 21]. Typically, they link bug-related user reviews to relevant code snippets [34] and stack traces [15]. For example, Ehsan et al.introduce the concept key topic of user review which are the topics that influence the app ratings most [10]. Yu et al. proposed *ReviewSolver* to locate problematic code of functional errors with user reviews [34]. Some works also leverage user reviews to prioritize static warnings [32], or link the crash-related information reported in both user feedback and stack traces[15]. While these work focuses on involving user review into app evolution process, none of them directly generate test cases from user review to automatically reproduce the crash reported by users. And it fundamentally differs RepRev from these prior works.

Automated Test Case Generation on Android. The dynamic search process in our approach shares some similarity with automated test case generation tools on Android. These tools aim for high coverage and the ability to detect bugs. They can be grouped into three categories based on their strategy to generate events: random exploration, model-based exploration and search-based exploration. Random test case generation tools [13, 22, 25], as its name shows, generate GUI actions randomly. They can generate events frequently, including invalid events, to test the robustness of the app. Model-based test case generation tools [6, 8, 17, 19, 30] explore the behavior of an app with the guidance of a GUI model, e.g., finite state machine, to generate only valid actions. Search-based test case generation tools [7, 23, 31] aim to synthesize some cases that can be hardly covered by two methods above, e.g, specific input, with specific techniques such as symbolic execution.

Recently, researchers have proposed automated approaches to reproduce crashes from bug reports [11, 35]. These approaches heavily rely on the concrete reproduction steps described in bug reports to reproduce bugs. However, as we have shown in Section 2, reproduction steps are usually missed in user review posted by non-technical end users. To overcome this problem, we propose various techniques, e.g., token frequency analysis and one-step exploration to effectively reproduce crashes reported in user reviews.

7 CONCLUSION

In this paper, we propose RepRev, an automated bug reproduction method, to help Android app developers reproduce the bugs described in user reviews. We conduct a large-scale empirical study on user reviews posted on Google Play. By reading and reproducing more than 3,000 bug-related user reviews, three characteristics and two challenges in bug reproduction are summarized. With these insights on user reviews, RepRev leverages various natural language processing techniques to extract useful information from user review, which is then used to guide the exploration on target app to reproduce the bug. The prototype system of RepRev is implemented, which successfully reproduces 44 out of 63 human-reproduced reviews. And, its time cost is comparable with manual bug reproduction process. It indicates that RepRev could be useful for developers to reproduce bug-related user reviews.

REFERENCES

[1] 2018. gensim. https://radimrehurek.com/gensim/models/word2vec.html.

[2] 2019. Ankidroid. https://play.google.com/store/apps/details?id=com.ichi2.anki.

[3] 2019. apktool. https://ibotpeaches.github.io/Apktool/.

[4] 2019. RepRev. https://drive.google.com/open?id=116JTG1jqzZB3JUXI9GJCrMMj-ZKy2wwE.

[5] 2019. uiautomator2. https://github.com/openatx/uiautomator2.

[6] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th International Conference on Automated Software Engineering*. ACM, 258–261. https://doi.org/10.1145/2351676.2351717

[7] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 1–11. https://doi.org/10.1145/2393596.2393666

[8] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 641–660. https://doi.org/10.1145/2509136.2509549

[9] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. 2014. AR-Miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 767–778. https://doi.org/10.1145/2568225.2568263

[10] Noei Ehsan, Zhang Feng, and Ying Zou. 2019. Too Many User-Reviews, What Should App Developers Look at First? *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2893171

[11] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 141–152. https://doi.org/10.1145/3213846.3213869

[12] Google. 2018. googlenews. https://code.google.com/archive/p/word2vec/.

[13] Google. 2019. androidmonkey. https://developer.android.com/studio/test/monkey.html.

[14] Google. 2019. navigationdrawer. https://material.io/components/navigation-drawer/.

[15] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald C Gall. 2018. Exploring the integration of user feedback in automated testing of android applications. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE Press, 72–83. https://doi.org/10.1109/SANER.2018.8330198

[16] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A Visaggio, Gerardo Canfora, and Sebastiano Panichella. 2017. Android apps and user feedback: a dataset for software evolution and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics*. ACM, 8–11. https://doi.org/10.1145/3121264.3121266

[17] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 269–280. https://doi.org/10.1109/ICSE.2019.00042

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 933–944. https://doi.org/10.1145/3180155.3180167

[19] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 204âĂŞ217. https://doi.org/10.1145/2594368.2594390

[20] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 269–282. https://doi.org/10.1145/3236024.3236055

[21] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. 2014. What do mobile app users complain about? *IEEE Software* 32, 3 (2014), 70–77. https://doi.org/10.1109/MS.2014.50

[22] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234. https://doi.org/10.1145/2491411.2491450

[23] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609. https://doi.org/10.1145/2635868.2635896

[24] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. ACL, 55–60. https://doi.org/10.3115/v1/P14-5010

[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105. https://doi.org/10.1145/2931037.2931054

[26] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 3111–3119. https://doi.org/10.5555/2999792.2999959

[27] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. 2017. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 106–117. https://doi.org/10.1109/ICSE.2017.18

[28] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[29] Juan Enrique Ramos. 2003. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the 1st Instructional Conference on Machine Learning*. Piscataway, NJ, 133–142.

[30] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256. https://doi.org/10.1145/3106237.3106298

[31] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2014. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes* 39, 1 (2014), 1–5. https://doi.org/10.1145/2557833.2560576

[32] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2017. OASIS: prioritizing static analysis warnings for Android apps based on app user reviews. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 672–682. https://doi.org/10.1145/3106237.3106294

[33] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 250âĂŞ265. https://doi.org/10.1007/978-3-642-37057-1_19

[34] Le Yu, Jiachi Chen, Hao Zhou, Xiapu Luo, and Kang Liu. 2018. Localizing Function Errors in Mobile Apps with User Reviews. In *Proceedings of the 48th International Conference on Dependable Systems and Networks*. IEEE Press, 418–429. https://doi.org/10.1109/DSN.2018.00051

[35] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 128–139. https://doi.org/10.1109/ICSE.2019.00030