ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps

YU ZHAO, University of Central Missouri, USA
TING SU, East China Normal University, China
YANG LIU, Nanyang Technological University, Singapore
WEI ZHENG, Northwestern Polytechnical University, China
XIAOXUE WU, Yangzhou University, China
RAMAKANTH KAVULURU, University of Kentucky, USA
WILLIAM G. J. HALFOND, University of Southern California, USA
TINGTING YU, University of Cincinnati, USA

The large demand of mobile devices creates significant concerns about the quality of mobile applications (apps). Developers heavily rely on bug reports in issue tracking systems to reproduce failures (e.g., crashes). However, the process of crash reproduction is often manually done by developers, making the resolution of bugs inefficient, especially given that bug reports are often written in natural language. To improve the productivity of developers in resolving bug reports, in this paper, we introduce a novel approach, called ReCDroid+, that can automatically reproduce crashes from bug reports for Android apps. ReCDroid+ uses a combination of **natural language processing (NLP)**, deep learning, and dynamic GUI exploration to synthesize event sequences with the goal of reproducing the reported crash. We have evaluated ReCDroid+ on 66 original bug reports from 37 Android apps. The results show that ReCDroid+ successfully reproduced 42 crashes (63.6% success rate) directly from the textual description of the manually reproduced bug reports. A user study involving 12 participants demonstrates that ReCDroid+ can improve the productivity of developers when resolving crash bug reports.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Bug report, bug reproducing, Android GUI testing

Yu Zhao and Tingting Yu part of this work was completed at University of Kentucky.
This research is supported in part by the NSF grant CCF-1652149 and NTU research grant NGF-2017-03-033.
Authors' addresses: Y. Zhao, University of Central Missouri, 116 W South St, Warrensburg, MO, USA, 64093; email: yzhao@ucmo.edu; T. Su, East China Normal University, 3663 North Zhongshan Rd, Shanghai, China, 200062; email: tsuletgo@gmail.com; Y. Liu, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; email: yangliu@ntu.edu.sg; W. Zheng, Northwestern Polytechnical University, 127 West Youyi Road, Beilin District, Xi'an, Shannxi, China, 710072; email: wzheng@nwpu.edu.cn; X. Wu, Yangzhou University, 196 Huayangxi Road, Yangzhou, Jiangsu, China, 225127; email: xiaoxuewu@yzu.edu.cn; R. Kavuluru, University of Kentucky, 725 Rose St, Lexington, KY, USA, 40506; email: ramakanth.kavuluru@uky.edu; W. G. J. Halfond, University of Southern California, 941 Bloom Walk, Los Angeles, CA, USA, 90089; email: halfond@usc.edu; T. Yu (corresponding author), University of Cincinnati, 2600 Clifton Ave, Cincinnati, OH, USA, 45221; email: tingting.yu@uc.edu.

1049-331X/2022/03-ART36 $15.00
https://doi.org/10.1145/3488244

ACM Reference format:
Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William G. J. Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 36 (March 2022), 33 pages.
https://doi.org/10.1145/3488244

1 INTRODUCTION

Mobile applications (apps) have become extremely popular – in 2020 there were over 2.9 million apps in Google Play's app store [13]. As developers add more features and capabilities to their apps to make them more competitive, the corresponding increase in app complexity has made testing and maintenance activities more challenging. The competitive app marketplace [53] has also made these activities more important for an app's success. A recent study found that 88% of app users would abandon an app if they were to repeatedly encounter a functionality issue [11]. This motivates developers to rapidly identify and resolve issues, or risk losing users.

To track and expedite the process of resolving app issues, many modern software projects use bug-tracking systems (e.g., Bugzilla [24], Google Code Issue Tracker [4], and Github Issue Tracker [12]). These systems allow testers and users to report issues they have identified in an app. Reports involving app crashes are of particular concern to developers because it directly impacts an app's usability [57]. Once developers receive a crash/bug report, one of the first steps to debugging the issue is to reproduce the issue in the app. However, this step is challenging because the provided information is written in natural language. Natural language is inherently imprecise and incomplete [18]. Even assuming the developers can perfectly understand the bug report, the actual reproduction can be challenging since apps can have complex event-driven and GUI related behaviors, and there could be many GUI-based actions required to reproduce the crash.

The goal of our approach is to help developers reproduce issues reported for mobile apps. We propose a new technique, ReCDroid+, targeted at Android apps, that can *automatically* analyze bug reports and generate test scripts that will reproduce app crashes. Specifically, given a raw bug report, ReCDroid+ leverages HTML parser [2], **convolutional neural networks (CNN)** [46], and **long-short term memory (LSTM)** [35] to extract crash and **S2R (steps to reproduce)** sentences. Several **natural language processing (NLP)** techniques are then utilized to analyze the S2R sentences of the reports and automatically identify GUI components and related information (e.g., input values) that are necessary to reproduce the crashes. Next, ReCDroid+ employs a novel dynamic exploration guided by the information extracted from bug reports to fully reproduce the crashes. ReCDroid+ takes as input a bug report and an APK and outputs a script containing a sequence of GUI events leading to the crash, which can be replayed directly on an execution engine (e.g., UI Automator [6]).

ReCDroid+differs from prior work for analyzing the reproducibility of bug reports [26, 56] because most existing techniques focus on improving the quality of bug reports. One important related work, Yakusu [32], is translating a bug report into a test case that has the highest number of matching steps with the bug report. None of them have considered using information from bug reports to automatically guide bug reproduction. For example, FUSION [56] enables the auto-completion of Android bug reports in order to provide more actionable information to developers. In contrast, ReCDroid+ takes crash description of the report as input, regardless of its quality, and extracts the information necessary to reproduce crashes. ReCDroid+ also differs from techniques on synthesizing information from bug reports [26, 32, 36, 61] because they focus on extracting useful information (e.g., test cases [32]) without directly targeting at reproducing crashes. In addition, ReCDroid+ provides an accurate solution to automatically extract sentences describing steps-to-reproduce (S2R) by using CNN to model sentence features and LSTM to

model sentence dependence. A set of rules are developed to overcome the false positives and false negatives of the extraction.

ReCDroid+ has been implemented as a software tool on top of two execution engines—Robotium [79] and UI Automator [6]. To determine the effectiveness of our approach, we ran ReCDroid+ on 66 bug reports from 37 popular Android apps. ReCDroid+ was able to successfully reproduce 42 (63.6%) of the crashes that can be reproduced by a professional human tester. Furthermore, 12 out of the 24 remaining crashes could have been reproduced by ReCDroid+ if limitations in the implementation of the execution engines (e.g., fail to click certain buttons) were to be removed.

To determine the usefulness of our tool, we conducted a light-weighted user study that showed that ReCDroid+ can reproduce 21 crashes not reproduced by at least one developer and was highly preferred by developers in comparison to a manual process. To assess the effectiveness of ReCDroid+ in handling bug reports written by different users, we asked another four participants to re-write the bug description for the 42 reproduced crashes. ReCDroid+ can detect 93% of the 168 (4×42) bug reports. To evaluate the robustness of ReCDroid+ in handling low-quality bug reports, we randomly removed 10%, 20%, and 50% content from the 42 original reproduced bug reports. Among all 630 mutated bug reports, ReCDroid+ can reproduce 88% of them. Overall, we consider these results to be very strong and they indicate that ReCDroid+ could be a useful approach for helping developers to automatically reproduce bug crashes.

ReCDroid+ provides several benefits for developers. First, because ReCDroid+ is fully automated, developers can simply push a button and work on other tasks instead of waiting for the results or manually reproducing bugs. Second, ReCDroid+ can be used with continuous integration server to enable automated/fast feedback, such that whenever a new issue is submitted, ReCDroid+ will automatically provide a reproducing sequence for developers. Third, users can use ReCDroid+ to assess the quality of a bug report before submitting it.

In summary, our paper makes the following contributions:

- The development of a tool that can automatically reproduce crash failures for Android apps directly from the textual description of bug reports.
- A novel text analysis technique that uses natural language processing (NLP) and deep learning to derive a set of heuristics and grammar rules that can automatically capture events and input values relevant to the bug reports.
- A novel algorithm that leverages the information extracted from bug reports to guide an exploration of the app's GUI to search for sequences of events leading to the reported crash.
- An empirical study showing that ReCDroid+ is effective at reproducing Android crashes and likely to improve the productivity of bug resolution.
- The implementation of our approach as a publicly available tool, ReCDroid+, along with all experiment data (e.g., apk and bug report datasets, user study) [8].

Further, we made a large number of changes related to the presentation of the article and included more detailed descriptions of the algorithm, evaluation and related work.[1]

In the next section we present the motivation and challenges of our approach. We then describe ReCDroid+ in Section 3. Our evaluation follows in Sections 4–5, followed by discussion in Section 6. We present related work in Section 7, and end with conclusions in Section 8.

2 OVERVIEW

In this section we describe our observations from studying hundreds of bug reports. We then discuss the design challenges of ReCDroid+.

[1]Our cover letter provides more details on the extension and revision over the original paper.

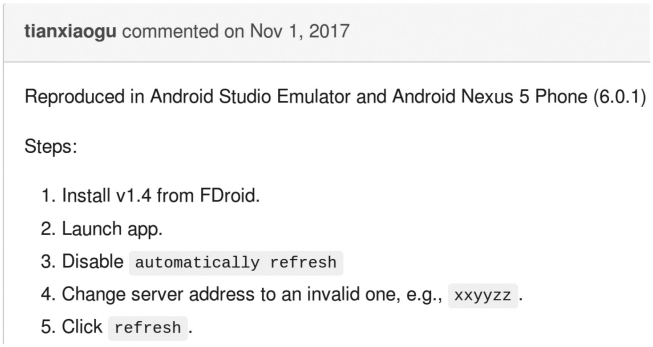

Fig. 1. Bug Report for LibreNews issue#22.

2.1 Observations

As the first step, we spent a month studying a large number of Android bug reports to understand their characteristics for guiding the design and implementation of ReCDroid+.

We collected Android apps from both Google Code Archive [5] and GitHub [12]. We crawled the bug reports from the first 50 pages in Google Code, resulting in 7,666 bug reports. We then searched Android apps from GitHub by using the keyworld "Android", resulting in 3,233 bug reports. Among all 10,899 bug reports, we used four case-insensitive keywords: "crash", "exception", "failure", and "error" involving app crashes. This yielded a total number of 1,038 bug reports. The result indicates that *a non-negligible number (9.5%) of bug reports involve app crashes.*

ReCDroid+ focuses on reproducing app *crashes* from bug reports *containing textual description of reproducing steps*, so we analyze the 1,038 crash bug reports and summarize the following findings: (1) 813 bug reports (78.3%) contain reproducing steps—the maximum is 11 steps, the minimum is 1 step, and the average is 2.3 steps; (2) only 3 out of 813 crashes are related to rotate action—they all occur 1–2 steps right after the rotate; (3) 398 of the 813 crash bug reports (49%) require specific user inputs on the editable GUI components to manifest the crashes—29 (3.5%) of them involve special symbols (e.g., apostrophe, hyphen); (4) 127 crashes (15.6%) involve generic click actions, including `OK` (79), `Done` (9), and `Cancel` (2).

2.2 Design Challenges

An example bug report is shown in Figure 1. In this example, the reporter describes the steps to reproduce the crash in five sentences. The goal of ReCDroid+ is to translate this sort of description to the event sequence shown in Figure 7 for triggering the crash. To achieve this goal, our approach contains two general steps: (1) Extracting the needed information from bug reports, and (2) Using this information to guide the reproduction of the crash. In the remainder of this section, we describe the design challenges of our approach and how we address these challenges. Details and algorithms of our approach are presented in Section 3.

2.2.1 Extracting Information from Bug Reports. As the first step, we need to identify the types of information that are broadly useful for guiding crash reproduction. From the examination of the 813 bug reports containing reproducing steps, our *insight* was that events that trigger new activities, interact with GUI controls, or provide values are the key parts of the steps provided by bug reporters. More broadly, these actions involve performing "a type of user action" on "a particular GUI component" with "specific values" (if the component is editable). Therefore, *action*, *target GUI component*, and *input values* are the main elements to be extracted from bug reports. These

elements are often contained in the steps-to-reproduce (S2R) sentences. To illustrate, consider the fourth step in Figure 1. Here, "change" is the user action, "Server" is the target GUI component, and "xxyyzz" is the input value.

There are two challenges in identifying the needed information for crash reproducing. First, we need to determine whether a given bug report involves a crashed output because our goal is to reproduce crashes. Second, we need to identify S2R sentences and extract the semantic representation of the S2R from the bug reports, defined by a tuple {action, GUI component, input}.

Challenge 1: Identifying crash report and extracting S2R and crash sentences. A bug report often contains mixed types of information, such as comments, code, status of the issue, and information unrelated to the bug. To automate the process of identifying crash bug reports and extracting S2R, we designed a novel deep learning model that can automatically identify S2R and crash sentences. A crash sentence contains information involving the symptom of the crash failure. The same model is used to handle both S2R and crash sentences.

We invited three students to label 4,000 bug reports randomly selected from 11,399 bug reports crawled from bug tracking systems. On each bug report, the students labeled whether a sentence is a crash sentence or S2R. Unlike traditional text classification methods [15, 54, 82], ReCDroid+'s deep learning model targets the bug report sentences and takes into account their relations. For example, a sentence right after the text "steps to reproduce:" may have a high possibility of being an S2R. A sentence sitting between two S2Rs has a high possibility to be an S2R.

It is challenging to create a robust deep learning model because of reasons such as incorrect labeling, unbalanced dataset, and unpredictable issues during the training process. To improve the accuracy of our model, we designed 12 rules to model the context of the bug report. Therefore, the extracted S2R from deep neural models is refined by these rules in order to increase the success rate of bug reproduction. For example, a rule may suggest that S2R should be extracted from the user comment with the largest number of S2R sentences among all user comments. The rationale behind this is that S2R sentences often appear together in one user comment of a bug report.

Challenge 2: Mapping bug report into semantic representations of events. The second design challenge is the extraction of the semantic representation of the reproducing steps from the bug reports, defined by a tuple {action, GUI component, input}. A seemingly straightforward solution to this challenge is to use a simple keyword search to match each sentence in the bug report against the name (i.e., the displayed text) of the GUI components from the app. However, keyword search cannot reliably detect input values or the multitude of syntactical relationships that may exist among user actions, GUI components, and inputs. For example, consider a sentence "I click the *help* button to *show* the word." If both *help* and *show* happen to be the names of app buttons, a keyword search could identify both *help* and *show* to be the target GUI components, whereas only *help* has a relationship with the action `click`. Moreover, reporters may use new words that do not match the name of the GUI component of the app. For example, a reporter may use "play the film" to describe the "movie" button.

Our insight is that the extraction process can be formulated as a slot filling problem [45, 55] in natural language processing (NLP). With this formulation each element of the event tuple is represented as a semantic slot and the goal of the approach then becomes to fill the slots with concrete values from the bug report. Our approach uses a mixture of NLP techniques and heuristics to carry out the slot filling. Specifically, we use the SpaCy dependency parser [38] to identify typical grammatical structures that were used in bug reports to describe the relevant user action, target GUI component, and input values. These were codified into 15 typical patterns, which we summarize and describe in Section 3. The patterns are used to detect event tuples of a new bug report and fill their slots with values.

To help bridge the lexical gap between the terminology in the bug report and the actual GUI components, our approach uses word embeddings computed from a Word2vec model [47] to determine whether two words are semantically related. For example, the words "movie" and "film" have a fairly high similarity.

2.2.2 Reproducing Crashes Guided by Bug Reports. In the second step, we will leverage the event representations extracted from bug reports to guide the reproduction of crashes. To make our approach practical, there are two major challenges in this step. First, we need to generate complete and correct sequences. Second, we want the reproducing process to be fast.

Challenge 3: Creating complete and correct sequences for bug reproduction. A key challenge for our approach is that even good bug reports may be incomplete or inaccurate. For example, steps that are considered obvious may be omitted or forgotten by the reporter. Therefore, our approach must be able to fill in these missing steps. Ideally, information already extracted from the report can be used to provide "hints" to identify and fill in the missing actions.

Existing GUI crawling tools [17, 20, 37, 52, 76] are not a good fit for this particular need. For example, many existing tools (e.g., A3E [20]) use a **depth-first search (DFS)** to systematically explore the GUI components of an app. That is, the procedure executes the full sequence of events until there are no more to click before searching for the next sequence. In our experience, this is sub optimal because if an interaction with an incorrect GUI component is chosen (due to a missing step), then the subsequent exploration of sub-paths following that step will be wasted.

For our problem domain, a guided DFS with backtracking is more appropriate. Using this strategy, our approach can check at each search level whether GUI components that are more relevant (i.e., match the bug report) to the target step are appearing and use this information to identify the next component to explore. If none of the components are relevant to the bug report, instead of deepening the exploration, ReCDroid+ can backtrack to a relevant component in a previous search level. This process continues until all relevant components in previous levels are explored before navigating to the subsequent levels.

Challenge 4: Making the reproduction efficient. Efficiency in the reproduction process is important for developer acceptance. An approach that takes too long may not seem worth the wait to developers, and an approach that generates a needlessly long sequence of actions may be overwhelming to developers. These two goals represent a tradeoff for our approach: identifying the minimal set of actions necessary to reproduce a crash can require more analysis time.

To achieve a reasonable balance between these two efficiency goals, we designed a set of optimization strategies and heuristics for our approach. For the guided crawl, we utilized strategies that included checking the equivalence of screens and detecting loops to avoid redundant backtracking, and prioritizing GUI components to be explored based on their likelihood of causing bugs. For minimizing the size of the sequence of GUI actions, whenever a backtrack was needed, our approach restarted the search from the home screen of the app and reset the state of the app. This avoids a common source of inefficiency present in other approaches (e.g., [17, 20, 76]) that add backtracking steps to their crawling sequence, which results in an overall much longer sequence of reproducing actions.

3 RECDROID+ APPROACH

The architecture of ReCDroid+ is shown in Figure 2. ReCDroid+ consists of three major phases—preprocessing, bug report analysis, and dynamic exploration. The preprocessing phase employs HTML parsing to extract the title and comments from the downloaded HTML bug reports. In the meantime, sentences are segmented by NLP. Then deep learning techniques are utilized to identify crash sentences and S2R sentences. The crash sentences are used to verify whether the bug report

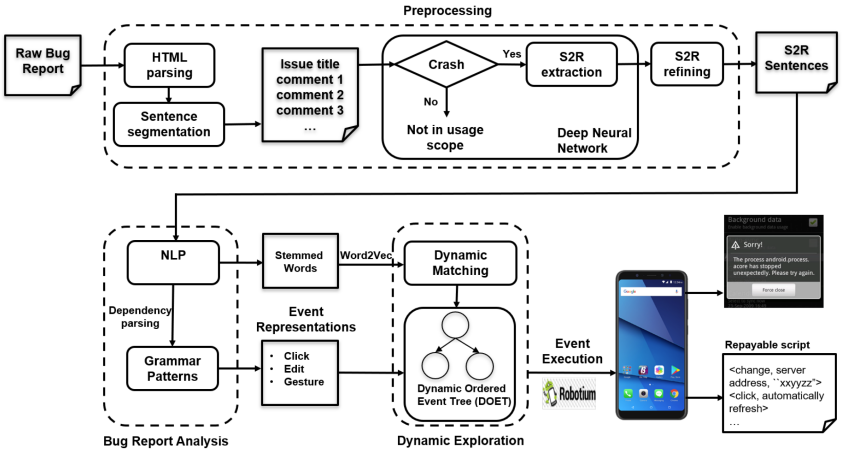

Fig. 2. Overview of the ReCDroid+ Framework.

is in the usage scope (i.e., handling only crash bug reports). The S2R refining rules reduce the false positives and false negatives of the extracted S2R.

To complete the sequence of extracted steps, the second phase employs a novel dynamic exploration of an app's GUI. This exploration is performed based on a **dynamic ordered event tree (DOET)** representation of the GUI's events, and searches for sequences of events that fill in missing steps and lead to the reported crash. ReCDroid+ saves the event sequences into a script that can be automatically replayed on the execution engine.

3.1 Preprocessing Bug Reports

In the preprocessing phase, ReCDroid+ first leverages HTML parsing to extract the actual content of bug report from files in HTML format. It then uses NLP techniques, combined with CNN and LSTM to models identify crash bug reports and extract S2R sentences. A set of modeling rules are derived to improve the accuracy of learning.

3.1.1 HTML Parsing. Inorder to perform analysis on bug reports, we will need to download the bug report files from bug tracking systems. These are often created in HTML format, which contain mixed types of information, such as CSS/HTML tags, navigation tags, status tags, and ads. Such noisy information can be overwhelming but is irrelevant to the actual content of bug reports. For example, as shown in Figure 3, in a bug report with only 10 lines of bug description [1], the associated noisy information contains more than a thousand lines. As the very first step, ReCDroid+ needs to eliminate the noises and extracts only texts that are relevant to the bug (a.k.a. *relevant content*).

ReCDroid+ employs a parsing technique to extract the relevant content of bug reports directly downloaded from the bug tracking systems. Specifically, the *title* and the *comments* are considered to be relevant and need to be extracted. The insight is that some of the titles provide information related to bug descriptions and symptoms, which may be used to identify S2R and crash reports. The first comment is often written by the report providing a detailed bug description and the followup comments are often discussions related to the bug.

For bug reports from the same bug tracking system, there is a unique HTML tag standard to label the title and comments position. For example, in Github, the title element is labeled by a particular HTML tag "//span[@class="js-issue-tittle"]" and comment element is labeled by HTML

Fig. 3. HTML parsing.

tag "//td". ReCDroid+ utilizes an HTML parsing tool called `lxml` [2] to identify the HTML tag and extract the texts under the title and comment elements. On a different bug tracking system, the names of HTML tags may be different. For example, in Google code, the HTML tag for the title element is "div[@id="gca-project-header"]". ReCDroid+ saves different tag names in a dictionary for each bug tracking system and selects the right one to use when needed. ReCDroid+ currently supports GitHub, Google Code, Bitbucket, and GitLab. It can be extended to support other bug tracking systems by creating a dictionary with system-specific tags.

ReCDroid+ employs a special mechanism to translate all numbers and <li> in the HTML bug report into a special string "numDot" to process texts that are related to S2R. The intuition is that sentences beginning with list symbols (e.g., bullets, numbers), transformed from the tag in HTML, are more likely to be S2R. Among the 4,000 bug reports, 2014/4322=46.6% of <li> tags are manually labeled as S2R.

3.1.2 Extract S2R and Crash Sentences. As the first step, ReCDroid+ needs to split the relevant text extracted from the HTML file into sentences. To do this, ReCDroid+ uses SpaCy [38] to detect the segmentation of sentences based on punctuation (e.g., "." , "!", "?"). Given a bug report with a sequence of sentences, ReCDroid+ builds a deep learning model to identify S2R and crash sentences. This is a typical binary classification problem. While there has been much work on addressing different kinds of text classification problem [43, 85], texts involving S2R and crash have their unique characteristics that should be considered.

Specifically, S2R and crash sentences tend to have adjacent context. In the example of Figure 1, a sentence right before or right after a S2R sentence is more likely to be an S2R sentence than the others. In addition, sentences that follow the text indicating steps to reproduce (e.g., a word "steps") has a higher chance to be S2R. Also, sentences that begin with listing symbols (e.g., bullet points, numbers) tend to be S2R. The crash sentences may also depend on the adjacent context, especially in the Java error message written in multiple adjacent sentences.

Given the unique characteristics, we need to build a model that is more suitable to handle the text classification program for detecting S2R and crash sentences in a bug report. To do this, ReCDroid+ first transfers a word to a word feature vector by using word embedding method, as shown in Figure 4. Next, based on the extracted *word* feature vector, a CNN and max layer are used to

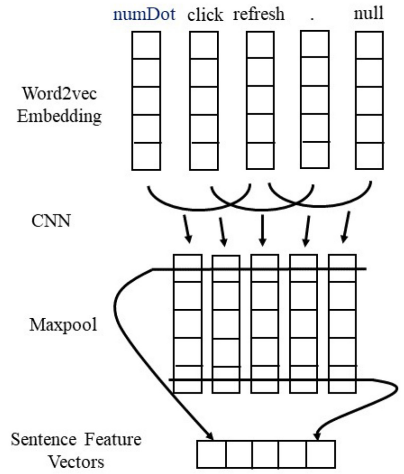

Fig. 4. The convolutional neural network extracts sentence features from each word. The word embedding per-trained through Word2vec.

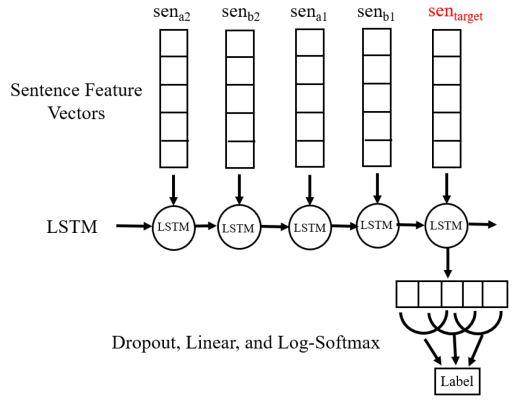

Fig. 5. The LSTM classifies the sentence with the dependence information from neighbor sentences.

generate a *sentence* feature vector, which can be used to predict at the sentence level. Finally, we use an LSTM to model the inter-sentence sequential dependencies and their role in the eventual label prediction for the target sentence, as shown in Figure 5. We next describe the three steps used to detect S2R and crash sentences.

Word embedding. ReCDroid+ uses Word2vec [7] to build pre-trained word vectors that are used as the input of the deep learning model. Word2vec builds word vector space using a large corpus of text as input. Every word in the corpus will be represented as real vector ($\mathbb{R}^d$) in the word vector space. In the pre-training process, 10,899 Android bug reports are crawled from GitHub and Google Code and the sentences extracted by the HTML parser are fed into Word2vec. To encourage robust learning, we set the number of epochs [2] to 2000, which is empirically determined in our evaluation.

[2]The number of epochs is a hyper-parameter that defines the number of times the learning algorithm will iterate through the entire training dataset [63].

Sentence feature vector extraction. A CNN and a maxpool layer (i.e., an output layer in a neural network [14]) are used to extract the sentence feature vector from the pre-trained word vectors. Figure 4 shows the overview of the model used to extract sentence feature vector. The input to the CNN model is a list of word vectors computed by Word2vec from each sentence and the output is a 2-D matrix, which represents the convolutional layer with multiple filters of CNN. A max layer is used to reduce the spatial dimension of input volume from 2-D matrix to 1-D matrix representing a sentence feature vector, which is later used as the input to LSTM. This sentence feature extraction model is inspired by the classical CNN sentence classification method [46]. However, in addition to the orignal approach, we leverage LSTM to model the dependency among sentences of bug reports.

We take the sentence "5. Click refresh" in Figure 4 as an example. After pre-processing, the sentence is transformed into a list of tokens: {"numDot", "Click", "refresh", "."}. There are three words and one punctuation in it. By applying Word2vec word embedding, each token is represented as a vector. The four vectors from four tokens can be combined into a 2-D matrix with 4 rows. Next, the CNN model processed this 2-D matrix and output an 2-D matrix as the convolution result to the max layer. Finally, the max layer reduce it to a 1-D matrix as the sentence feature vector of the original sentence "5. Click refresh".

Sentence dependence modeling. ReCDroid+ uses an LSTM to model sentence dependence, as shown in Figure 5. The insight is that the non-target sentences with shorter distance to the target sentence are more likely to be an S2R or crash sentence. In each step, the LSTM learns what to omit and what to retain from the previous inputs of the input sequence. Information from sentences that are farther away from the target sentence is less likely to be retained when compared to those that are closer to it. [70]. ReCDroid+'s LSTM model orders the non-target sentences by their distances to the target sentence. By default, ReCDroid+ selects four neighbor sentences as input to the LSTM model.

In the Figure 5, the target sentence and its four neighbor sentences are represented as $\{sen_{b_2}, sen_{b_1}, sen_{target}, sen_{a_1}, sen_{a_2}\}$, where sen_{target} is the target sentence, sen_{b_2} and sen_{b_1} are the second and first sentences right before sen_{target}, and sen_{a_1}, sen_{a_2} are the first and second sentences right after sen_{target}. When there are no sentences before or after the target sentence, i,e., the dependence sentences are missing, we use an all zero padding vector to represent the missing dependence sentence feature.

Following the idea of **Named Entity Recognition (NER)** [28], the well-known dropout method is used to prevent overfitting of the LSTM model [64]. The *linear layer* followed by the softmax layer decodes the label of each target sentence. The last cell used as input to the LSTM model is the target sentence's feature. This cell outputs the feature involving sentence dependence information to the dropout, linear, and softmax layer. The final output is the decoded label. If the output >= 0.5, the target sentence is an S2R (label = 1), otherwise, it is not (label = 0). This decoded label rule is utilized for both crash sentences and S2R extraction.

3.1.3 Policy based S2R Sentences Selection. The extracted S2R sentences by the deep learning model may not be ready to use immediately because of the potential high false positive and false negative rates. A false positive occurs when ReCDroid+ mistakenly labels a non-S2R sentence as an S2R sentence, which may bring noises to the exploration. A false negative indicates an S2R sentence is identified as not-S2R, which may cause ReCDroid+ to fail to reproduce the crash due to a missing reproducing step. The deep learning model may also output duplicated S2R sentences. For example, the same reproducing step may be mentioned multiple times in the report. Such duplicate S2R sentences may misguide bug reproduction.

To address the above challenges, we designed a set of S2R refining rules to refine the S2R labels of the bug report sentences. The input to the refining rules are the whole bug report, as well

as the S2R sentences output by the the deep model. The refining rules may extract additional sentences from the bug report as S2R to handle the model's false negatives, or eliminate certain S2R sentences identified as positive by the model to address its false positives. The final output of S2R after applying the refining rules is denoted as $S2R_r$.

Table 2 lists the eleven rules, their description, and the rationale behind each rule. These rules are derived from the 4,000 human labeled bug reports from GitHub and Google Code (Section 4.1). Among the 4,000 bug reports, 1,997 (50%) bug reports involve S2R and 189 (9.4%) of them contain S2R on title. Reporters write S2R in the first comment in 1,954 (98.3%) bug reports. There are 848 bug reports containing S2R right after the key word "reproduce". Among 1,343 bug reports containing at least two S2R sentences, 977 (72%) bug reports have two consecutive S2R sentences. Among 953 bug reports that contain more than three S2R sentences, 456 (47.8%) bug reports have three consecutive S2R sentences. In the equation representation of each rule, "cmt_i" suggests sentences in the i^{th} comment, "$M()$" suggests the S2R extraction deep learning model, and "title" suggests the title of a bug report. For example, by applying the first rule, among all labeled S2R sentences, only the ones in the comment containing the most number of S2R sentences are used for bug reproduction.

ReCDroid+ applies one rule at a time for reproducing a crash. If a rule does not take effect, it will be ignored and ReCDroid+ will move to the next rule. For example, when applying rule 11, if the bug report does not have any "to produce" text, this rule will be ignored. In the optimum scenario, developers may use multiple machines (e.g., devices, VMs) to reproduce the same crash in parallel, where each machine is applied a different rule. The reproduction process terminates if at least one device successfully reproduces the crash or a timeout occurs. However, in the cases where only a limited number of machines are available, ReCDroid+ needs to decide the order of rules to be applied. For example, if a rule does not take effect or fails to reproduce the crash, ReCDroid+ will decide which rule to apply next. In this case, the choice of rule order may substantially affect how much time it takes to reproduce a crash.

ReCDroid+ employs a clustering-based prioritization strategy to prioritize rules in terms of their likelihood of successfully reproducing crashes in a timely manner. The clustering-based strategy considers the relationships among different rules. Our assumption is that if multiple rules have similar effects in reproducing the same crash, i.e., successfully reproduce the same crash in a similar amount of time, they tend to expose similar behaviors. Hence, such rules ought to belong to the same cluster.

The clustering algorithm is based on the dataset of historical bug reports used for reproducing crashes (i.e., training dataset). Each bug report in the dataset is recorded as to whether it was successfully reproduced and the time cost for reproducing it on each refining rule. Each rule is associated with a vector, where each element in the vector is associated with a bug report, indicating the time spent (in seconds) on reproducing the bug report. The length of the vector is equal to the number of bug reports in the dataset. Given a rule, if the bug report failed to reproduce the crash, the element associated with the bug report is represented as a large negative number (i.e., -1000), which is used to distinguish it from the successful cases.

We use mean-shift [60], an unsupervised clustering algorithm to cluster the eleven vectors. Mean-shift is a centroid-based clustering algorithm. Within a given region in a coordinate system, it updates candidates for centroids to be the mean of the points. While other clustering algorithms, such as k-mean can also be used, we found that mean-shift performed better in our evaluation.

The algorithm clusters the rules into different groups. Rules in one group share similar behaviors. Specifically, ReCDroid+ iterates through each group. It selects and removes the rule with the lowest reproducing time in each group. The selected rules are then sorted from lowest reproducing time

Table 1. S2R Refining Rules

ID	Rule	Description	Rationale
1	$i = \arg\max_i len(M(cmt_i))$ $S2R_r = M(cmt_i)$	(1) Find the comment cmt_i with the most extracted S2R sentences. (2) Extract all S2R from cmt_i.	The comment with the most extracted S2R sentences is likely to describe S2R.
2	$i = \arg\max_i len(M(cmt_i))$ $S2R_r = extra1Neib(M(cmt_i), cmt_i))$	(1) Find the comment cmt_i with the most extracted S2R sentences. (2) Extract neighboring sentences of the S2R extracted from cmt_i.	S2R sentences may come from cmt_i's neighboring sentences.
3	$i = \arg\max_i len(M(cmt_i))$ $S2R_r = extra2Neib(M(cmt_i), cmt_i))$	(1) Find the comment cmt_i with the most extracted S2R sentences. (2) Extract two neighboring sentences of the S2R extracted from cmt_i.	To deal with false negatives.
4	$i = \arg\max_i len(M(cmt_i))$ $S2R_r = cmt_i$	(1) Find the comment cmt_i with the most extracted S2R sentences. (2) Use all sentences in cmt_i as S2R.	To deal with false negatives.
5	$S2R_r = cmt[0]$	Use the first comment as S2R.	Reporter tends to report S2R in the first comment.
6	$i = \arg\max_i len(cmt_i)$ $S2R_r = cmt_i$	Use the comment with the largest number of sentences as S2R.	This comment may contain more information than others.
7	$S2R_r = title$	Use title as S2R.	Reporters may write S2R on the title of the bug report.
8	$S2R_r = \sum_i cmt_i$	Use all comments as S2R.	S2R may spread across multiple comments.
9	$S2R_r = \sum_i cmt_i + title$	Use all texts of the bug report as S2R.	To deal with false negative.
10	$S2R_r = title$ $if \quad len(M(title)) > 0$	Use the title only when it is an extracted S2R sentence.	Similar to rule 7, but it saves time when the title is not S2R.
11	$sentIndex, line_i =$ $\quad findFirst(\text{“}toreproduce\text{”}, line)$ $S2R_r = extafter(extra2Neib($ $\quad M(line_i), line_i)), sentIndex)$	(1) Find the text line $line_i$ containing “to reproduce”. (2) Use the two neighboring sentences from extracted S2R right after $line_i$.	Sentences after “to reproduce” is likely to be S2R. This rule reduces duplicated S2R.

to the highest reproducing time and added to a list L. The rationale is that rules in the same group tend to have same capability in reproducing crashes. This process continues until all rules are removed from the groups. During the crash reproducing process, a rule is iteratively removed from the head of the list L and used to extract S2Rs until the crash is successfully reproduced.

3.2 Bug Report Analysis

ReCDroid+ uses 15 grammar patterns (summarized from the 22 patterns [10]) to extract the the semantic representations of events (i.e., the tuple {action, GUI component, input}) described in a bug report.

3.2.1 Grammar Patterns. The 15 grammar patterns were derived from the corpus of 813 Android bug reports described in Section 2.1. These patterns are broadly applicable and can be reused (e.g., by compiling them into a library) for new Android bug reports. Specifically, for each bug report we analyzed the dependencies among words and phrases in the sentences describing reproducing steps. Specifically, we use SpaCy's grammar dependency analysis to identify the **part-of-speech (POS)** tag (e.g., noun, verb) of each word within a sentence, parse the sentence into clauses (e.g., noun phrase), and label semantic roles, such as direct objects. Figure 6 shows an example of the results of the spaCy dependency analysis on two sentences with different structures.

Broadly, the grammar patterns could be grouped into three types of interactions with an app: click events (e.g., click buttons, check checkboxes), edit events (e.g., enter a text box with a number), and gesture events (e.g., rotate). Table 2 lists the eight typical grammar patterns (the full list can be found in our artifacts [8]). Column 3 shows the percentage of the 813 bug reports in which each grammar pattern applies. We next describe these patterns.

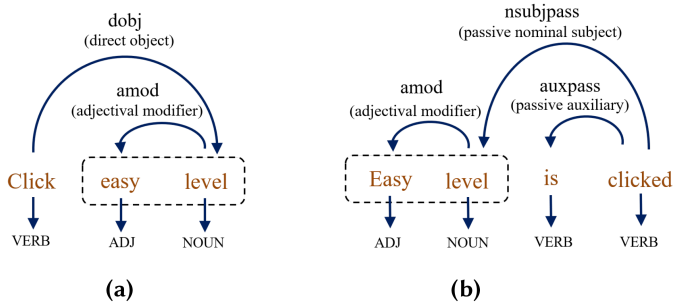

Fig. 6. Examples of Dependency Trees.

Table 2. Summary of Grammar Patterns

Category	*ID*	*Pct.*	*Grammar Pattern*	*Example*	Event Tuple
Click	CR1	22.7%	action → dobj (→NP)	Click$_{[action]}$ {easy level$_{[dobj]}$}$_{[NP]}$	<click, (easy) level>
	CR2	1.3%	action → nsubjapss (→ NP)	{Easy level$_{[dobj]}$}$_{[NP]}$ is clicked$_{[action]}$	<click, (easy) level>
	CR3	14%	action → pobj (→ NP)	I made a click$_{[action]}$ on {easy level$_{[pobj]}$}$_{[NP]}$	<click, (easy) level>
	CR4	0.2%	CR* + ‘‘multiple’’/‘‘twice’’/‘‘times’’	Click$_{[action]}$ {easy level$_{[pobj]}$}$_{[NP]}$ many times	<multi-click, (easy) level>
	CR5	0.7%	CR* + ‘‘long’’	Long click$_{[action]}$ {easy level$_{[pobj]}$}$_{[NP]}$	<long-click, (easy) level>
	CR6	6.3%	action → amod/compound (→ NP)	Clicking$_{[action]}$ {easy level$_{[amod]}$}$_{[NP]}$	<click, (easy) level>
Edit	TR1	13.1%	action→dobj|obj|attr→prep→pobj (→NP) prep ∈ {on,in,to}	Input$_{[action]}$ xxyyzz$_{[dobj]}$ to$_{[prep]}$ {server address$_{[pobj]}$}$_{[NP]}$	<input, (server) address, xxyyzz>
	TR2	3.3%	action→dobj|obj|attr(→NP)→prep→pobj prep ∈ {with, by}	Input$_{[action]}$ {server address$_{[dobj]}$}$_{NP}$ with$_{[prep]}$ xxyyzz$_{[pobj]}$	<input, (server) address, xxyyzz>
	TR3	0.9%	action→dobj|obj|attr(→NP)→prep→pobj prep ∈ {to, with}, action ∈ {change}	Change$_{[action]}$ {server address$_{[dobj]}$}$_{NP}$ to$_{[prep]}$ xxyyzz$_{[pobj]}$	<change, (server) address, xxyyzz>
	TR4	2%	TR1|TR2|T3 + EG(→NP)	Input$_{[action]}$ a number$_{[dobj]}$ to kilometer$_{[pobj]}$, e.g., {10}	<input, kilometer, 10>
	TR5	7.3%	TR1|TR2|T3|T4 + NUM → UNIT	I input 10$_{[NUM]}$ km$_{[UNIT]}$,	<input, km, 10>
	TR6	16.5%	action→dobj|pobj(→NP)	I fill$_{[action]}$ xxyyzz$_{[dobj]}$,	<input, *, xxyyzz>
	TR7	1.6%	action(leave)→dobj|obj(→NP)	I leave$_{[action]}$ address$_{[dobj]}$,	<input, address, “ ”>
Gesture	NR1	0.7%	action(rotate)	Rotate$_{[action]}$ the screen	<rotate>
	NR2	7.8%	action(back)	Back$_{[action]}$ to before page.	<back>

Click Events. ReCDroid+ uses six grammar patterns to extract the click event tuple. The “input” element in the tuple is not applicable to click events. In Table 2, CR1 specifies that the direct object (i.e., dobj) of the click action is the target GUI component. Also, the **noun phase (NP)** of the direct object corresponds to the target GUI component. The **second pattern (CR2)** identifies the GUI component that has an nsubjpass (i.e., passive nominal subject) relation with the action word. The **third pattern (CR3)** specifies that the **object of a preposition (pobj)** of the click action is the target GUI component.

Edit Events. We identified seven grammar patterns for extracting edit events. In Table 2, the first grammar pattern (TR1) specifies that if the preposition is a word in {on, in, to}, the **direct object (dobj)** is the input value and the **preposition object (pobj)** is the target GUI component. On the other hand, in the second pattern (TR2), if the preposition is with or by, the direct object (dobj) is the GUI component and the preposition object (pobj) is the input value. The change action requires a special grammar pattern to handle (TR3) because the preposition object is often preceded by a target GUI component and followed by an input value.

As for the fourth grammar pattern (TR4), we observe that words happening after the phrase (EG) containing an introducing example (e.g., e.g., example, say), especially NOUN, often involve input values. Therefore, TR4 specifies that if the sentence prior to EG contains a user action and a GUI component detected by a grammar pattern (TR1, TR2, or TR3), then EG contains an input value associated with the GUI component. To extract the input value, ReCDroid+ first extracts the NOUN from EG and the NP is identified as an input value. For TR5, ReCDroid+ searches for the word right after the number and if the word is a unit (UNIT∈{kg, cm, litter}), it is considered to be a target GUI component. The number is identified as an input value.

Gesture Events. The grammar patterns for gesture events involve only the "action" element in the event tuple. The current implementation of ReCDroid+ supports only the `rotate` event. Nevertheless, our grammar patterns can be extended by incorporating other events, such as zoom and swipe.

3.2.2 Extracting Event Representations. Given a bug report, ReCDroid+ uses the grammar patterns to extract event representations (i.e., event tuples) relevant for reproducing bugs. ReCDroid+ first splits the crash description into sentences, where sentence boundaries are detected by syntactic dependency parsing from SpaCy [38]. It then applies *stemming* [44][3] to the words in each sentence with each word assigned a sentence ID (used for the guided exploration).

Next, ReCDroid+ determines if a sentence describes a specific type of event. To do this, we construct a vocabulary containing words that are commonly used to describe the three types of actions (e.g., "click", "enter", "rotate"). This vocabulary was manually constructed by manually analyzing the corpus of 813 bug reports. The frequency distribution of the words in the vocabulary can be found in our artifacts [8]. ReCDroid+ then matches each sentence (using the stemmed words) against the vocabulary and if any match is found, the grammar patterns associated with the event type are applied to the sentence for extracting the target GUI components and/or input values. For example, the 4th step in Figure 1 contains a word "change", so the grammar pattern TR3 is applied.

3.2.3 Limitations of Using Grammar Patterns. The grammar patterns can be used to extract event tuples from well-structured sentences. However, in the case of complicated or ambiguous sentences, NLP techniques are likely to render incorrect **part-of-speech (POS)**, dependency tags, or sentence segmentation. While this problem can be mitigated by training the tags [68], it comes with an additional cost. Moreover, the extracted target GUI components from the bug report may not match their actual names in the app. Such inaccuracy and incompleteness may negatively impact the efficiency of the dynamic exploration. Section 3.3.2 illustrates how ReCDroid+ obtains additional information from unstructured texts to address the mismatch between bug reports and target apps.

3.3 Guided Exploration for Reproducing Crashes

The goal of the second phase is to identify short sequences of events that complete the sequence identified in the first phase and allow it to fully and automatically reproduce the reported crash. To do this, ReCDroid+ builds and uses a *Dynamic Ordered Event Tree* $\mathcal{T} = (V, E)$ to guide an exploration of the app's GUI. The set of nodes, V, represents the app's GUI components, and the set of edges, E, represents event transitions (i.e., from one screen to another by exercising the component) observed at runtime. The tree nodes of each level (i.e., screen) are ordered (shown as left to right) according to the descending order of their relevance to the bug report. The details of determining the relevancy of GUI components is described in Section 3.3.1.

During the exploration, ReCDroid+ iteratively selects, for each screen, the most relevant component (the leftmost node of the subtree associated with the screen). If none of the GUI components match the bug report, ReCDroid+ traverses the tree leaves to select another matching but unexplored GUI component to execute. This process continues until all matching components in previous levels (i.e., screens) are explored before navigating to the subsequent screens to expand tree levels. Compared to conventional DFS, our search strategy can avoid potential traps. The advantage of using the DOET is that by prioritizing the GUI components, the leaf traversal would

[3]Stemming is the process of removing the ending of a derived word to get its root form. For example, "clicked" becomes "click".

ALGORITHM 1: Guided Dynamic Exploration

Require: App, stemmed words from bug report: W, Eg
Ensure: Script $\mathcal{R}$ /*sequence of events leading to the reported crash*/
1: $\mathcal{S} \leftarrow <Launch>$
2: $\mathcal{T}.root \leftarrow Launch$
3: **while** time < LIMIT **do**
4: $P \leftarrow$ Execute($\mathcal{S}$, App)
5: **if** P triggers BR's crash **then**
6: $\mathcal{R} \leftarrow$ Save ($\mathcal{S}$)
7: **return**
8: **if** IsAddLeafNodes ($\mathcal{T}$, S.last) is true **then**
9: $U \leftarrow$ GetAllElem (P)
10: **for** each GUI element $u \in U$ **do** /*current screen*/
11: **if** IsMatch(u, Eg, W) is true **then**
12: u.status $\leftarrow ready$ /*can be explored*/
13: **end for**
14: $\mathcal{T} \leftarrow$ AddOrderedNodes (U, $OrderCriteria$)
15: **if** for all $LeafNodes \in \mathcal{T}$ is explored **then**
16: **return**
17: **if** for all $LeafNodes \in \mathcal{T}$ is not $ready$ **then**
18: $LeafNodes \leftarrow ready$ /*need backtrack*/
19: $\mathcal{S} \leftarrow$ FindSequence($\mathcal{T}$) /*select a GUI component to explore*/

always select the leftmost relevant tree leaf to explore without iterating through all components on the screen.

3.3.1 ReCDroid+'s Guided Exploration Algorithm. Algorithm 1 outlines the algorithm of ReCDroid+'s dynamic exploration. The algorithm begins by launching the app (Line 1) and then enters a loop to iteratively construct a **dynamic ordered event tree (DOET)** (Lines 3 – 19). At each iteration, ReCDroid+ uses the tree to compute an event sequence $\mathcal{S}$ (Line 19) to be executed in the next iteration (Line 4). The algorithm terminates when (1) the reported crash is successfully reproduced (Lines 5–7), (2) all paths in the tree are executed (Lines 15–16), or (3) a timeout occurs (Line 3). During the exploration, ReCDroid+ may accidentally trigger crashes different from the one described in the bug report. ReCDroid+ prompts the user when a crash is detected and lets the user decide if it is the correct crash for the purpose of terminating the search.

After exercising the last GUI component from the event sequence $\mathcal{S}$, ReCDroid+ determines whether the DOET should be expanded (Line 8). If a loop or an equivalent screen is detected (discussed in Section 3.3.4), ReCDroid+ stops exploring the GUI components in the current screen. Otherwise, ReCDroid+ obtains all GUI components from the current screen and matches them against the bug report (Algorithm 2). It then orders these components and adds them as the leaf nodes of the last exercised GUI component (Lines 9–14).

A GUI component is considered to be *relevant* to the bug report and ordered on the left of the tree level when the following conditions are met: (1) it matches the bug report and was not explored in previous levels; (2) upon meeting the first condition, it appears earlier in the bug report according to its associated sentence ID; (3) it is a clickable component and does not meet the first condition, but its associated editable component matches the bug report (because only by exercising the clickable component can the exploration bring the app to a new screen); (4) upon meeting any of the above conditions, it is naturally more dangerous. Our current implementation considers `OK` and `Done` as naturally more dangerous components (Finding 4), because the former component is more likely to bring the app to a new screen. The conditions (1) and (2) consider the order of S2R during the exploration, so that ReCDroid+ can avoid duplicate and incorrect matching.

The routine `FindSequence` (Line 19) determines which GUI component to explore next to find an event sequence to execute in the next iteration. If any components in the current tree level are relevant to the bug report, it selects the leftmost leaf and appends it to $\mathcal{S}$. If none of these components are relevant, ReCDroid+ traverses the tree leaves from left to right until finding a

ALGORITHM 2: `IsMatch`

Require: GUI component in app: u, Events detected by grammar patterns: E_g, A set of bug report sentences: S
Ensure: A boolean value
20: **for** each event $g \in E_g$ **do**
21: **if** u.similar (g.u) > 0.8 **then** /*use Word2vec*/
22: **if** e.action is edit **then**
23: u.setText ($g.input$)
24: **return** $true$
25: **end for**
26: $W_b \leftarrow$ GenerateNGram (S - $E_g.S$)
27: $W_u \leftarrow$ GenerateNGram (u)
28: **for** each $w_u \in W_u$ **do**
29: **for** each $w_b \in W_b$ **do**
30: **if** w_u.similar (w_b) > 0.8 **then**
31: **if** e.action is edit **then**
32: u.setText (D)
33: **return** $true$
34: **end for**
35: **end for**
36: **return** $false$

leaf node that is relevant to the bug report. Instead of adding backtracking steps to $\mathcal{S}$, ReCDroid+ finds the suffix path from the leaf to root to be executed in the next iteration. The goal of this is to minimize the size of the event sequence. If the algorithm detects that none of the leaf nodes are relevant to the bug report, it means that we may need to deepen the exploration to discover more matching GUI components. Therefore, ReCDroid+ resets all leaf nodes to *ready* in order to continue the search (Line 19–20).

DOET does not capture the `rotate` action because it is not a GUI component. In addition, because of the possible missing information in the bug report, it is hard to determine the location of the rotate action. Therefore, we need to find the right locations in an event sequence to insert the `rotate` action (Line 4). We use a threshold R to specify the maximum number of steps to the last event at which `rotate` was exercised. Finding 2 shows that a crash often occurs 1–2 steps after the rotate. Therefore, by default, $R = 2$.

3.3.2 Dynamic Matching. To determine whether a GUI component matches a bug report (Line 11), ReCDroid+ utilizes `Word2vec` [47], a word embedding technique, to check if the content of the GUI component is semantically similar with any of the extracted event representations or the words from sentences in which grammar patterns cannot be used. Word2vec uses a neural network model to learn word embedding from a large corpus of text. Word2vec represents each word by a numerical vector. Cosine similarity score in the range of [0, 1] between vectors of two words suggests the semantic similarity between words (1 indicates an exact match). The `Word2vec` model is trained from a public dataset *text8* containing 16 million words and is provided along with the source code of `Word2vec` [7]. The model uses a score in the range of [0, 1] to indicate the degree of semantic similarity between words (1 indicates an exact match). ReCDroid+ uses a relatively high score, 0.8, as the threshold. We observed that using a low threshold may misguide the search toward an incorrect GUI component. For example, the similarity score of "start" and "stop" is 0.51 but the two words are not synonymous.

Algorithm 2 outlines the process of matching a GUI component observed at runtime. ReCDroid+ first compares the observed GUI component (u) with the event tuples (E_g) to detect if there is a match. If u is an editable component, the corresponding input values from e are filled into the text field (Lines 21–24). If no matches are found from the previous step, ReCDroid+ analyzes the sentences in which grammar patterns do not apply (Lines 26 – 35). It generates n-grams[4], from both

[4]An n-gram is a contiguous sequence of n items from a given sequence of text, which has been widely used in information retrieval [72] and natural language processing [23].

Fig. 7. The steps of reproducing the crash described in Figure 1.

the bug report description and the GUI component u (Lines 26 – 27). ReCDroid+ then compares the content of the GUI component against the bug report based their generated grams (Lines 28 – 30). We consider unigrams (single word tokens) and bigrams (two consecutive word tokens) that are commonly used in existing work [19, 59, 67].

If an editable GUI component does not match any events extracted from grammar patterns, ReCDroid+ associates the component with the following values (D in Line 32): 1) input values for other editable components extracted by grammar patterns that match the data type (e.g., digit, string) of the editable component, and 2) special symbols appearing in the bug report, such as "apostrophe", "comma", and "quote" because we observed that such symbols are likely to cause problems (Finding 3). If neither of the two types of values can be found in the bug report, ReCDroid+ randomly generates one.

3.3.3 A Running Example. Figure 8 shows a partial DOET for the example in Figure 7. The shaded nodes indicate the GUI components leading to the reported crash. ReCDroid+ first launches the app and brings the app to the screen in Figure 7(a). There is one clickable GUI component G in the screen, which is not relevant to the bug report. Since by traversing the leaf nodes (only G) ReCDroid+ does not find any relevant component, it sets the status of component

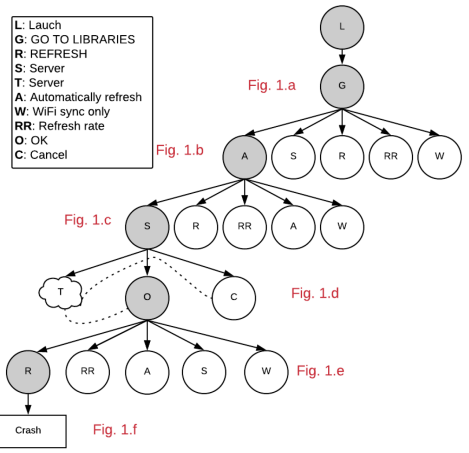

Fig. 8. Dynamic Ordered Event Tree (DOET) for Figure 7.

G to *ready* and continues the search (Lines 17–18). In the second iteration, ReCDroid+ clicks component G and brings the app to Figure 7(b). ReCDroid+ ranks the GUI components in the current screen and adds them to the tree (Lines 8–16). Specifically, the first four components (i.e., A, S, R, RR) match the bug report description and are ordered on the left of the tree level. Internally, the four components are ranked in terms of the orders of their appearance in the bug report. ReCDroid+ then checks all nodes in the current level (Figure 7(b)) and selects the leftmost leaf (A) to execute, which brings the app to the screen of Figure 7(c). At this tree level, A is placed on the right because it has been explored before. In the fourth iteration, exercising the leftmost leaf node S brings the app to Figure 7(d), since the editable component `Server` matches the bug report description, its corresponding input value is filled in and the associated clickable components are considered to be relevant. Because `OK` is more likely to bring the app to a new screen, it is ordered before `Cancel`. In the last iteration (Figure 7(d)), both A and S are placed on the right because they have been explored. Lastly, R is executed and the crash is triggered.

We next illustrate how ReCDroid+ backtracks. Suppose in Figure 7(c), none of the components are relevant to the bug report, ReCDroid+ would traverse the leaf nodes of the whole DOET from left to right until finding a matching and unexplored GUI component. Therefore, component S in the screen of Figure 7(b) would be selected. So in the next iteration, ReCDroid+ restarts the search and executes the sequence $L \rightarrow G \rightarrow S$.

3.3.4 Optimization Strategies. ReCDroid+ employs several optimization strategies to improve the efficiency of the algorithm by avoiding exploring irrelevant GUI components (Line 8). For example, ReCDroid+ checks if the current screen is the same as the previous screen. A same screen may suggest either an invalid GUI component was clicked (e.g., a broken button) or the component always brings the app to the same screen (e.g., refresh). In this case, creating children nodes for the current screen can potentially cause the algorithm to explore the same screen again and again. To address this problem, ReCDroid+ sets the status of the last exercised GUI component G to *dead* to avoid expanding the tree level from G. We also develop an algorithm to detect loops in each tree path. For example, in a path *DABCABCABC*, the subsequence *ABC* is visited three times in a row. In this case, ReCDroid+ keeps only one subsequence and the leaf node is set to *dead*, so the loop will not be explored in the future.

4 EMPIRICAL STUDY

To evaluate ReCDroid+, we consider five research questions:

RQ1: How effective and efficient is ReCDroid+ at reproducing crashes in bug reports?
RQ2: How effective and efficient is ReCDroid+ at extracting S2R sentences and crash sentences?
RQ3: To what extent do the NLP techniques in ReCDroid+ affect its effectiveness and efficiency?
RQ4: Does ReCDroid+ benefit developers compared to manual reproduction?
RQ5: Can ReCDroid+ reproduce crashes from different levels of low-quality bug reports and bug reports written by other reporters?

4.1 Datasets

We need to prepare datasets for evaluating our approach. To avoid overfitting, we do not consider the 813 Android bug reports that we used to identify the grammar patterns. Instead, we randomly crawled an additional 360 bug reports containing the "keywords crash and exception" from GitHub. We next included all 15 bug reports from the FUSION paper [56] and 25 bug reports from a recent paper on translating Android bug reports into test cases [32]. FUSION considers the quality of these bug reports as low, so we aim to evaluate whether ReCDroid+ is capable of handling low-quality bug reports. This process yields a total of 400 bug reports.

We then manually filtered the 400 collected bug reports to get the final set that can be used in our experiments. This filtering was performed independently by three graduate students, who have 2-4 years of industrial software development experience. We first filtered bug reports involving actual app crashes, because ReCDroid+ focuses on crash failures. This yielded 320 bug reports. We then filtered bug reports that could be reproduced manually by at least one inspector, because some bugs could not be reproduced due to lack of apks, failed-to-compile apks, environment issues, and other unknown issues. These bug reports cannot assess ReCDroid+ itself and thus was excluded from the dataset. In total, we evaluated ReCDroid+ on 66 bug reports from 37 apks. The cost of the manual process is quite high: the preparation of the dataset required around 500 hours of researcher time.

To train the deep learning model for extracting S2R and crash sentences, we crawled 3,233 bug reports from Github and 7,666 bug reports from Google Code and randomly selected 4,000 bug reports to build the dataset. These bug reports are different from the 66 bug reports in the testing set. During the manual inspection, we read the reports with sufficient details in the bug descriptions and examined the discussions posted by commentators to decide if a sentence is an S2R or not. To ensure the correctness of our results, the manual labeling process was performed independently by two graduate students. We measured the agreement between the two graduates by using Cohen's Kappa (k) [22], which can be used to assess the agreement between two raters. Cohen's Kappa produces values between 0 and 1, where 0 indicates poor agreement and 1 perfect agreement. For extracting S2R sentences, k = 0.86. For the extraction of crash sentences, k = 0.84. The results suggest that the labeling process is reliable. Any time there was dissension, a third graduate student was involved and they discussed until reaching a consensus. Note that we spent about 20 hours training the three students on how to analyze the bug reports and label S2R sentences. To make sure they understood the process, we asked the students to start with a small number of bug reports from the dataset.

4.2 Implementation

We conducted our experiment on a physical x86 machine running with Ubuntu 16.04. This machine has i7-4790 CPU @ 3.60GHz and 32 GB memory with no GPU. The NLP techniques of ReCDroid+ was implemented based on the SpaCy dependency parser [38]. The dynamic exploration compo-

nent was implemented on top of two execution engines, Robotium [79] and UI Automator [6], for handling apps compiled by a wide range of Android SDK versions. An apk compiled by a lower version Android SDK (<6.0) can be handled by Robotium and that by a higher version SDK (>5.0) can be handled by UI Automator.

4.3 Experiment Design

4.3.1 RQ1: Effectiveness and Efficiency of ReCDroid+. We measure the effectiveness and efficiency of ReCDroid+ in terms of whether it can successfully reproduce crashes described in the bug reports within a time limit (i.e., 22 hours) and efficiency in terms of the time it took to reproduce each crash. The default setting is 22 hours because ReCDroid+ spends two hours trying out every rule with a total cost of 22 hours (11 rules*2 hours). In fact, we found that ReCDroid+ can reproduce all reproducible bug reports in three hours. To ensure the crash found by ReCDroid+ is the same as the one described in the bug report, for each crash found by ReCDroid+, we manually inspect the bug report to determine if it is the reported one. If an error stack trace is provided in the bug report, we manually compared the stack trace with the stack trace generated by ReCDroid+ to see if they are identical.

4.3.2 RQ2: Effectiveness and Efficiency of ReCDroid+ in extracting S2R and crash sentences. We performed a five cross-validation on the 4,000 labeled bug reports. Of these five folds, four folds are used to train the deep learning model while the 5th fold is used to evaluate the performance of the model. We used precision, recall, and F-measure to evaluate the effectiveness of ReCDroid+ in extracting S2R and crash sentences. We consider F-measures over 0.7 to be good [26]. When measuring the efficiency, we calculated the time (in seconds) spent on the extraction.

In addition, we evaluated the performance refining rules, i.e., whether they can increase the success rate of crash reproduction. Specifically, we calculated the time of crash reproduction with the refining rules. The refining rules were applied to ReCDroid+ one by one with the clustering-based prioritization strategy described in Section 3.1.3. We showed the results on reproducing success and time on each single refining rule.

4.3.3 RQ3: The Role of NLP in ReCDroid+. Within ReCDroid+, we assess whether the use of the NLP techniques can affect ReCDroid+'s effectiveness and efficiency. We consider two "vanilla" versions of ReCDroid+. The first version, ReCDroid+$_N$, is used to evaluate the effects of using grammar patterns. ReCDroid+$_N$ does not apply grammar patterns, but only enables the second phase on dynamic matching. The second version is ReCDroid+$_D$, which evaluates the effects of applying both grammar patterns and dynamic matching. The comparison between ReCDroid+$_D$ and ReCDroid+$_N$ can assess the effects of using dynamic matching. ReCDroid+$_D$ is a non-guided systematic GUI exploration technique (discussed in Section 6). The time limits for running ReCDroid+$_N$ and ReCDroid+$_D$ were also set to three hours.

4.3.4 RQ4: Usefulness of ReCDroid+. The goal of RQ4 is to evaluate the experience developer had using ReCDroid+ to reproduce bugs compared to using manual reproduction. We are concerned about whether ReCDroid+ can reproduce crashes faster than manual reproduction and save developers' effort because of its automation. We recruited 12 graduate students as the participants. All had at least six months of Android development experience and three were real Android developers working in companies for three years before entering graduate school. Each participant read the 54 bug reports and tried to manually reproduce the crashes. All apps were preinstalled. For each bug report, the 12 participants (the three graduate students who built the dataset are not included) timed how long it took for them to understand the bug report and reproduce the bug. If a participant was not able to reproduce a bug after 30 minutes, that bug was marked as not repro-

duced. After the participants attempted to reproduce all bugs, they were asked to use ReCDroid+ on the 54 bug reports. This was followed by a survey question: would you prefer to use ReCDroid+ to reproduce bugs from bug reports over manual reproduction? We also let the participants write down their thoughts about using ReCDroid+. Note that to avoid bias, the participants were not aware of the purpose of this user study.

4.3.5 RQ5: Handling Low-Quality Bug Reports. The same bug may be reported in different quality levels by different reporters. The goal of RQ5 is to assess the ability of ReCDroid+ to handle different levels of low-quality bug reports for the same bug. Since judging the quality of a bug report is often subjective, we created low-quality bug reports by randomly removing a set of words from the original bug reports. Some words in the original bug reports are not related to S2R or exist in duplicated S2R, so removing them may not reduce the quality of bug reports . Therefore, we focused on removing words from texts containing unduplicated S2R sentences.

Randomly removing words from S2R can simulate two scenarios. The first scenario is missing S2R because removing certain words may cause the loss of information in the entire S2R, such as which button to click or what text to enter. The second scenario is the low-quality S2R, in which the grammar is too poor to be understood by humans or effectively handled by ReCDroid+. For example, the bug report `olam-2` contains a S2R sentence "Force Close when enter word with apostrophe". If we remove the word "enter", the sentence is not easy to be understood. If we remove key word "apostrophe", the bug report would be very difficult to reproduce.

Specifically, we considered three variations for each of the 42 bug report reports reproduced by ReCDroid+ in order to mimic different levels of quality: (1) removing 10% of the words, (2) removing 20% of the words, and (3) removing 50% of the words. Due to the randomization of removing words from bug reports, we repeated the removal operation five times for each bug report across the three quality levels. We evaluate the effectiveness and efficiency of ReCDroid+ in reproducing crashes in the 630 ($42 \times 3 \times 5$) bug reports. Again, the time limit was set to 3 hours.

4.3.6 RQ6: Handling Bug Reports Generated by Different Users. Given a bug, different reporters may describe it in different language styles. To assess the ability of ReCDroid+ to handle bug reports written by different users, we recruited another four participants to write bug reports using a template similar to Figure 1 for the 42 crashes reproduced by ReCDroid+. To avoid introducing bias from the original bug reports, we recorded videos of the steps needed to manually reproduce the crash for every bug report. After viewing the video, each participant was asked to write bug reports for the 42 crashes. In total, the participants constructed 168 bug reports. We then evaluated the effectiveness and efficiency of ReCDroid+ in reproducing the 168 bug reports.

5 RESULTS AND ANALYSIS

Table 3 summarizes the results of applying ReCDroid+, ReCDroid+$_N$, and ReCDroid+$_D$ in 54 out of the 66 bug reports. We did not include the remaining 12 crashes because they failed to be reproduced due to the technical limitations of the two execution engines rather than ReCDroid+. For example, Robotium failed to click certain buttons (e.g., [3]). Column 2 shows the number of reproducing steps in each bug report. Columns 3–20 show whether the technique successfully reproduced the crash, the size of the event sequence, and the time each technique took.

5.1 RQ1: Effectiveness and Efficiency of ReCDroid+

As Table 3 shows, ReCDroid+ reproduced 42 out of 54 crashes, a success rate of 77.7%. The time required to reproduce the crashes ranged from 16 to 7,331 seconds with an average time of 466.4 seconds. All four crash bug reports (marked with ⋆) from the FUSION paper [56] and eight

Table 3. RQ1, RQ4, RQ5: Different Techniques

#BR.	# steps	Reproduce Success						# Event in Sequence						Time (Seconds)						User
		RD	RD_N	RD_D	*Sap.*	*St.*	*Mon.*	*RD*	RD_N	RD_D	*Sap.*	*St.*	*Mon.*	*RD*	RD_N	RD_D	*Sap.*	*St.*	*Mon.*	(12)
newsblur-1053	5	Y	Y	Y	Y	Y	N	7	7	7	360	23	-	47	64	132.3	483	10	>	12
markor-194	3	Y	N	N	N	N	N	4	-	-	-	-	-	1222	>	>	>	>	>	12
birthdroid-13	1	Y	Y	Y	N	Y	N	8	8	8	-	10	-	351	351	1089	>	6600	>	9
car-report-43★	4	Y	Y	Y	N	N	N	18	18	16	-	-	-	600	602	101	>	>	>	8
opensudoku-173	8	Y	N	N	N	N	N	9	-	-	-	-	-	633	>	>	>	>	>	10
acv-11★	5	Y	Y	Y	N	N	N	8	8	5	-	-	-	479	467	2060	>	>	>	7
acv-12	4	Y	Y	Y	N	N	N	4	4	4	-	-	-	107	108	960	>	>	>	12
anymemo-18	1	Y	Y	Y	Y	Y	Y	3	3	3	204	6	7900	150	148	798	245	282	417	11
anymemo-422	3	Y	N	N	N	N	N	2	-	-	-	-	-	257	>	>	>	>	>	12
anymemo-440	4	Y	Y	N	N	N	N	8	8	-	-	-	-	1168	1185	>	>	>	>	12
notepad-23★	3	Y	Y	Y	N	N	N	6	6	6	-	-	-	186	194	1731	>	>	>	11
olam-2★	1	Y	N	N	Y	N	N	2	-	-	354	-	-	36	>	>	122	>	>	7
olam-1	1	Y	N	N	N	N	N	2	-	-	-	-	-	19	>	>	>	>	>	11
FastAdapter-394	1	Y	Y	Y	Y	Y	Y	1	1	1	364	13	6900	26	23	445	123	1860	385	9
LibreNews-22	4	Y	Y	Y	Y	Y	Y	6	6	5	394	198	30700	126	111	729	1203	120	1729	12
LibreNews-23	6	Y	N	N	N	Y	Y	3	-	-	-	516	46600	60	>	>	>	480	2669	12
LibreNews-27	4	Y	Y	Y	Y	Y	Y	6	6	5	394	198	30700	116	132	1075	1203	120	1729	11
SMSSync-464	2	Y	Y	Y	N	Y	N	4	4	4	-	12	-	787	740	5194	>	2258	>	10
transistor-63	5	Y	Y	Y	Y	N	Y	3	3	3	283	-	1200	28	27	65	120	>	74	12
zom-271	5	Y	Y	Y	Y	Y	Y	5	5	5	273	2230	1100	75	87	508	72	1800	74	11
PixART-125	3	Y	Y	Y	N	N	Y	5	5	5	-	-	58000	581	607	1032	>	>	3068	12
PixART-127∗	3	Y	Y	Y	N	N	N	5	5	5	-	-	-	146	146	992	>	>	>	12
ScreenCam-25∗	3	Y	Y	N	-	N	Y	6	6	-	-	-	14586	787	795	>	-	>	3600	11
ventriloid-1	3	Y	N	N	N	N	Y	9	-	-	-	-	700	56	>	>	>	>	36	11
Nextcloud-487	1	Y	Y	Y	N	N	N	2	2	2	-	-	-	69	62	944	>	>	>	11
obdreader-22	4	Y	Y	N	N	N	N	8	8	-	-	-	-	912	929	>	>	>	>	12
dagger-46∗	1	Y	Y	Y	-	Y	Y	1	1	1	-	419	2500	18	18	21	-	420	1155	12
ODK-1402	2	Y	N	N	N	N	N	2	-	-	-	-	-	73	>	>	>	>	>	10
ODK-2075	2	Y	Y	Y	N	N	N	3	3	3	-	-	-	91	90	2249	>	>	>	12
ODK-2086	2	Y	Y	Y	N	N	N	3	3	3	-	-	-	101	95	2982	>	>	>	12
ODK-2191	1	Y	Y	Y	N	N	N	3	3	3	-	-	-	231	227	2212	>	>	>	12
ODK-2525	2	Y	Y	Y	-	N	N	2	2	2	-	-	-	47	55	191	-	>	>	7
ODK-2601	2	Y	Y	N	-	N	N	3	4	-	-	-	-	193	190	>	-	>	>	10
k9-3255	2	Y	N	N	N	N	N	4	-	-	-	-	-	7331	>	>	>	>	>	12
k9-2612∗	4	Y	Y	Y	-	N	N	4	4	2	-	-	-	179	180	5731	-	>	>	10
k9-2019∗	1	Y	Y	Y	-	N	N	3	3	3	-	-	-	57	65	1352	-	>	>	11
Anki-4586∗	5	Y	Y	N	-	N	N	7	7	-	-	-	-	99	100	>	-	>	>	12
TagMo-12∗	1	Y	Y	Y	-	N	N	1	1	2	-	-	-	16	15	30	-	>	>	12
FlashCards-13∗	4	Y	Y	Y	-	N	N	3	3	3	-	-	-	68	70	94	-	>	>	12
Gnu-596	2	Y	N	N	-	N	N	1	-	-	-	-	-	18	>	>	-	>	>	12
Gnu-633	2	Y	N	N	-	N	Y	4	-	-	-	-	40400	72	>	>	-	>	1976	12
TimeTracker-35	2	Y	Y	Y	-	N	Y	4	4	4	-	-	126500	1974	1963	857	-	>	6989	10
TimeTracker-10	1	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	10
TimeTracker-138	4	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	10
FastAdaptor-113	2	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	7
Memento-169	3	N	N	N	N	N	N	-	-	-	-	-	-	>	>	>	>	>	>	2
ScreenCam-32	1	N	N	N	N	N	N	-	-	-	-	-	-	>	>	>	>	>	>	10
ODK-1796	2	N	N	N	Y	N	N	-	-	-	254	-	-	>	>	>	138	>	>	4
AIMSICD-816	3	N	N	N	N	N	N	-	-	-	-	-	-	>	>	>	>	>	>	1
materialistic-76	6	N	N	N	N	N	N	-	-	-	-	-	-	>	>	>	>	>	>	5
Gnu-663	2	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	11
Fdroid-1821	5	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	10
Shortyz-135	1	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	5
PdfViewer-33	4	N	N	N	-	N	N	-	-	-	-	-	-	>	>	>	-	>	>	5
total		42	31	26	8	10	13													

RD. = ReCDroid+. St. = Stoat Mon. = Monkey Sap. = Sapienz "✓" = Crash reproduced. "N" = Crash not reproduced. "-" = Not applicable. ">" = exceeded time limit (22 hours).

bug reports (marked with ∗) from Yakusu [32] were successfully reproduced. The results indicate that *ReCDroid+ is effective in reproducing crashes from bug reports*. The 12 cases where ReCDroid+ failed will be discussed in Section 6.

For six of the twelve cases where ReCDroid+ failed, we found that the failures were due to the following reasons. First, ReCDroid+ does not support scroll or swipe action (FastList-113, materialistic-1067, AIMSICD-816). Second, ReCDroid+ cannot deal with non-deterministic behaviors of the apps (Memento-169, Shortyz-135).

As a generic GUI exploration and testing tool, ReCDroid+$_D$ is similar to existing Android testing tools [9, 16, 20, 52, 65, 66], which detect crashes in an unguided manner. ReCDroid+$_D$ was shown to be competitive with Monkey [9], Sapienz [52], and the recent work Stoat [66] on our experiment subjects. Since Sapienz only works on the apps with android SDK version 4.4, the comparison between ReCDroid+ and Sapienz only uses apps with SDK version 4.4. Monkey, Stoat, and Sapienz are compared with ReCDroid+ in two hours for reproducing the crash in each bug report. Specifically, as Table 3 shows, ReCDroid+$_D$ reproduced 30 more crashes than Stoat, 19 more crashes than Sapienz, and 27 more crashes than Monkey. For the crashes successfully reproduced by all three techniques, the size of event sequence generated by ReCDroid+$_D$ was 98.8% smaller than Stoat, 98.8% smaller than Sapienz, and 99.9% smaller than Monkey. With regards to efficiency, ReCDroid+$_D$ required 87.8% less time than Stoat, 87.4% less time than Sapienz, and 87.7% less time than Monkey.

5.2 RQ2: Effectiveness and Efficiency of Extracting S2R and Crash Sentences

When performing cross-validation on the 4,000 labeled bug reports, for S2R extraction, the precision, recall, and F1 score is 0.683, 0.722, and 0.702, respectively. For crash sentence extraction, the precision, recall, and F1 score is 0.756, 0.849, and 0.789, respectively. When using all 4,000 bug reports as the training set and the 52 bug reports of the subject apps as the testing set, the precision, recall, and F1 scores of S2R sentences extraction are 0.852, 0.821, and 0.836. The crash sentence extraction results are 0.932, 0.835, and 0.88. ReCDroid+' reproducing dataset is more accurate than the random crawled dataset, because bug reports in the reproducing data set is manually filtered to ensure they are crash reports. These bug reports may have better written quality than the random crawled dataset, so it is easy to extract the needed information by the deep learning model.

ReCDroid+ failed to identify crash sentences in two apps: car-report-43 and obd-22. In car-report-43, the crash sentence contains a keyword "deadlock", which is uncommon in the training set (i.e., 4,000 bug reports). In obd-22, the crash sentence a word "live", which prevents the deep learning model from correctly labeling it as a crash sentence even if this sentence contains the word "crash". We hypothesize that a larger dataset may help to mitigate the inaccuracy problem. As a result, ReCDroid+ failed to automatically reproduce the crashes for these two apps due to missing oracles. Nevertheless, if the crashes sentences were correctly labeled, ReCDroid+ successfully reproduced them.

Regarding efficiency, ReCDroid+ spent four hours on training the deep learning model using the 4,000 bug reports. The model only needs to be trained once. When applied the model to our reproducing dataset, it took less than five seconds.

The influences of refining rules. We evaluated the influences of the 12 refining rules on the effectiveness and efficiency of crash reproduction. As in Table 4 shows, for each bug report, there exists at least one rule that can successfully reproduce it. On the other hand, none of the rules were able to reproduce all bug reports. In summary, ReCDroid+ requires at least three rules to successfully reproduce all bug reports.

Table 4 shows the reproducing times when applying different rules for each bug report. The times vary significantly. For example, the crash in Nexcloud-487 was reproduced in 47 seconds with rule-1, but it took 67 minutes to reproduce the crash with rule-2, rule-3, and rule-4. On the other hand, rules 1–4 shared similar costs to reproduce other bug reports. Also, one rule may cost much less on one bug report than on another. For example, rule-7 cost less time on timeTrack-35 than rule-1, but the situation is opposite on ODK-2601. Some rules are critical to the success of reproducing certain bug reports. For example, olam-2 can be only reproduced by rule 7, 9, and 10. This is because the S2R "Force close when enter word with apostrophe" only occurred in the

Table 4. RQ3: Different Rules

#BR.	Time (sec)										
	r1	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*	*r9*	*r10*	*r11*
newsblur-1053	47	266	163	155	127	127	126	153	160	-	-
markor-194	1222	1250	1243	1253	1229	1230	>	1238	1191	-	-
birthdroid-13	-	-	-	-	83	84	351	83	351	-	-
car-report-43	600	593	592	611	271	260	259	594	593	-	-
opensudoku-173	633	556	577	582	571	541	>	686	602	>	-
acv-11	479	512	499	512	516	502	1512	525	502	1397	-
acv-12	107	107	107	123	108	110	294	111	95	-	110
anymemo-18	150	145	144	143	39	152	154	42	42	-	-
anymemo-422	257	258	259	250	250	250	250	249	249	250	250
anymemo-440	1168	1296	1260	2110	2060	1061	>	1059	1156	>	>
notepad-23	-	-	-	-	1497	1505	186	1497	186	186	-
olam-2	-	-	-	-	83 N	83 N	36	79 N	36	37	-
olam-1	-	-	-	-	86 N	84 N	18	88 N	19	-	-
FastAdapter-394	26	23	23	27	27	27	495	28	27	513	-
LibreNews-22	126	112	149	152	148	151	714	150	151	-	-
LibreNews-23	60	36	39	43	39	36	>	66	59	-	-
LibreNews-27	116	184	182	188	187	181	768	328	294	-	71
SMSSync-464	787	810	799	815	823	743	>	728	>	>	711
transistor-63	28	26	27	26	26	53	27	28	27	26	-
zom-271	75	73	74	89	90	90	392	90	89	-	73
PixART-125	581	574	578	585	584	588	1482	583	5633	1481	583
PixART-127	146	139	147	141	143	146	418	141	297	417	141
ScreenCam-25	787	724	727	724	728	775	769	953	962	791	>
ventriloid-1	56	52	54	54	54	53	>	61	54	>	-
Nextcloud-487	69	4039	4039	4024	4023	4015	52	4102	4114	64	-
obdreader-22	-	-	-	-	910	917	1119	910	912	-	-
dagger-46	18	18	18	18	18	19	17	18	17	-	18
ODK-1042	73	72	72	80	73	73	81	87	76	88	78
ODK-2075	91	90	89	89	114	164	243	90	89	232	116
ODK-2086	101	125	103	101	97	99	274	138	153	287	112
ODK-2191	231	225	227	228	227	225	227	229	229	227	-
ODK-2525	47	47	47	48	48	48	47	48	48	-	47
ODK-2601	193	193	192	194	193	194	4050	193	193	-	193
k9-3255	>	>	>	>	>	>	>	>	>	>	131
k9-2612	179	178	178	>	>	>	133	63	62	134	109
k9-2019	57	53	56	53	55	65	55	1728	3055	-	-
Anki-4586	99	96	96	96	>	100	243	121	96	306	115
TagMo-12	16	13	13	13	13	13	13	14	14	13	-
FlashCards-13	68	67	68	67	67	67	67	67	69	-	-
Gnu-596	18	21	23	29	30	21	19	20	17	19	18
Gnu-633	72	68	67	68	67	66	138	67	67	138	68
timeTracker-35	1974	1973	1973	1974	1974	2786	418	2788	326	418	-

"r1" = Rule 1. "N" = Crash not reproduced. "-" = Not applicable(empty extraction under the rule). ">" = Crash not reproduced in exceeded time limit (2 hours).

title of the bug report. In contrast, ventriloid-1 cannot be reproduced by rule 7 and 10 because the essential S2R information is not described in the title. While rule 9 takes all text involving title and comments as input to prevent false negatives in S2R extraction, it does not always successfully reproduce the crash. For example, SMSSync-464 spends much longer time on rule 9 than the other rules. This is because the second comment "Google" and "play" misguides the exploration since

there are two buttons named "google voice" and "message play". ReCDroid+ matches the two buttons and thus wastes the exploration time.

We also evaluated our clustering algorithm in generating the rule sequence. We used a random generation method (i.e., generated a sequence randomly) as a baseline. Specifically, we split the 42 successfully reproduced bug reports into a training set and testing set. In each iteration, we randomly selected 32 bug reports as a training set and the other 10 bug reports as a testing set, and then apply the clustering algorithm and the random method, respectively. The total number of iterations is set to 1,000. The results suggest that on average, we used U-test [58] to measure the significant difference between the average time cost of mean-shift and the random. The results that, on average, mean-shift took 854 seconds to reproduce a crash, which was almost half of the time taken by random (i.e., 1,594 seconds), and the difference is statistically significant.

5.3 RQ3: The Role of NLP in ReCDroid+

When compared, ReCDroid+ to ReCDroid+$_N$ and ReCDroid+$_D$, ReCDroid+ successfully reproduced 35.4% and 57.7% more crashes than ReCDroid+$_N$ and ReCDroid+$_D$.

For the crashes successfully reproduced by all three techniques, the size of event sequence generated by ReCDroid+ was 0.008% smaller than ReCDroid+$_N$ and 6.3% bigger than ReCDroid+$_D$. Both ReCDroid+$_N$ and ReCDroid+$_D$ generated short event sequences because like ReCDroid+, they do not backtrack. Instead, whenever a backtrack was needed, they restarted the search from the home screen of the app (Algorithm 1). With regards to efficiency, ReCDroid+ required 0.004% less time than ReCDroid+$_N$ and 90.2% less than ReCDroid+$_D$. Overall, these results indicate that *the use of NLP techniques, including both the grammar patterns and the dynamic word matching, contributed to enhancing the effectiveness and efficiency of ReCDroid+.*

We also examined the effects of false positives and false negatives reported when applying the 22 grammar patterns to each bug report, since false positives may misguide the search and false negatives may jeopardize the search efficiency (certain useful information is missing). In the 42 crashes successfully reproduced by ReCDroid+, we found that all false positives were discarded during the dynamic exploration because the identified false GUI components did not match with the actual GUI components of the apps. With regards to false negatives, we found that they were all captured by the dynamic word matching. Therefore, the false negatives and false positives of the grammar patterns did not negatively affect the performance of ReCDroid+, although our results may not generalize to other apps.

5.4 RQ4: Usefulness of ReCDroid+

The last column of Table 3 shows the number of participants (out of 12) that successfully reproduced the crashes. While all crashes were reproduced by the participants, among all 42 crashes reproduced by ReCDroid+, 21 of them failed to be reproduced by at least one participant. For the twelve bug reports that ReCDroid+ failed to reproduce, the success rate of human reproduction is also low. These results suggest that *ReCDroid+ is able to reproduce crashes that a small subset of developers in our study were unable to reproduce.* One reason for the failures was that developers need to manually search for the missing steps, which can be difficult due to the large number of GUI components. As columns 2 and 9 in Table 3 indicate, in 29 bug reports, the number of described steps is smaller than the number of events actually needed for reproducing the crashes. Another reason was because of the misunderstanding of reproducing steps.

We also compute the time required for each participant to successfully reproduce all 54 bug reports. The results show that the time for successful manual reproduction ranged from 3 seconds to 1,631 seconds, with an average 170.5 seconds—63.4% less than the time required for ReCDroid+ on the successfully reproduced crashes. Such results are expected as ReCDroid+ needs to explore

a number of events during the reproduction. However, *ReCDroid+ is fully automated and can thus reduce the painstaking effort of developers in reproducing crashes.* Among all 42 crashes successfully reproduced by ReCDroid+, the reproduction time required by individual participants ranged from 3 to 1,631 seconds.

It is worth noting that while it is possible the actual app developers could reproduce bugs faster than ReCDroid+, ReCDroid+ can still be useful in many cases. First, ReCDroid+ is fully automated, so developers can simply push a button and work on other tasks instead of waiting for the results or manually reproducing crashes. Second, ReCDroid+ can be used with a continuous integration server [33] to enable automated and fast feedback, such that whenever a new issue is submitted, ReCDroid+ will automatically provide a reproducing sequence for developers. Third, users can use ReCDroid+ to assess the quality of bug reports—a bug report may need improvement if the crash cannot be reproduced by ReCDroid+.

The 12 participants were then asked to use ReCDroid+ and indicate their preferences for the manual vs tool-based approach. We used the scale *very useful*, *useful*, and *not useful*. Our results indicated that 7 out of 12 participants found ReCDroid+ very useful and would always prefer ReCDroid+ to manual reproduction, 4 participants indicated ReCDroid+ is useful, and one participant indicated that ReCDroid+ is not useful. The participant who thought ReCDroid+ is not useful explained that, for some simple crashes, manual reproduction is more convenient. On the other hand, the participate agreed that ReCDroid+ is useful for handling complex apps (e.g., K-9). The 12 participants also suggested that ReCDroid+ is useful in the following cases: (1) bugs that require many steps to reproduce, (2) bugs that require entering specific inputs to reproduce, and (3) bug reports that contain too much information. The above results suggest that *developers generally feel ReCDroid+ is useful for reproducing crashes from bug reports and they prefer to use ReCDroid+ over manual reproduction.*

5.4.1 RQ5: Handling Low-Quality Bug Reports. Columns 4–9 of Table 5 reports the reproducibility of ReCDroid+ for the bug reports at the three different quality levels. The column *success* indicates the number of mutated bug reports (out of 5) that were successfully reproduced at each quality level. The column *time* indicates the average time (and the standard deviation) required for reproducing the crash. The results show that among all 630 mutated bug reports for the three quality levels, ReCDroid+ was able to reproduce 94.7%, 90%, and 80% of the bug crashes, respectively. Even when 50% of the words were removed, ReCDroid+ could still successfully reproduce 27 bug reports for all 5 crashes. The slowdowns caused by the missing information with respect to the original bug reports were only 1.6x, 1.9x, and 2.8x, respectively. These results suggest that *ReCDroid+ can be used to effectively handle low-quality bug reports with different levels of missing information.*

5.4.2 RQ6: Handling Bug Reports Generated by Different Reporters. The last two columns of Table 5 report the reproducibility and cost (and standard deviation) of ReCDroid+ for bug reports re-written by the four participants. In total, ReCDroid+ successfully reproduced 157 out of 168 (93.4%) bug reports, with an average time of 410 seconds—49.4% more time than reproducing the S2R sentences from original bug reports. In eleven cases, ReCDroid+ failed to reproduce the crash due to the following reasons: (1) incorrect input values were provided; (2) important steps were missing; and (3) important words were misspelled. These results suggest that *ReCDroid+ is robust in handling bug reports written by different users.*

6 DISCUSSION

Limitations. The current implementation in ReCDroid+ does not support item-list, swipe, or scroll actions. In our experiment, three fail-to-be-reproduced bug reports (FastAdaptor-113,

Table 5. RQ4: Different Quality Levels

#BR.	Pure S2R sentences		QL-10% (5)		QL-20% (5)		QL-50% (5)		Re-write (4)	
	# Events	Time (sec)	Success	Time (sec)	Success	Time (sec)	Success	Time (sec)	Success	Time (sec)
newsblur-1053	7	158	5	196(102)	5	94(50)	5	136(88)	4	41(1)
markor-194	4	1181	5	1601(24)	4	1564(85)	4	1608(30)	3	1603(16)
birthdroid-13	5	107	5	159(128)	5	383(205)	5	659(185)	4	136(30)
car-report-43	16	310	5	280(3)	5	288(6)	5	286(1)	4	593(195)
opensudoku-173	9	576	5	770(458)	3	2267(1153)	3	2325(1636)	4	516(6)
acv-11	8	501	5	1077(1299)	5	1844(1448)	5	1911(1321)	3	1543(242)
acv-12	4	104	4	112(4)	1	121(-)	2	475(0)	2	78(1)
anymemo-18	3	67	5	90(49)	5	62(9)	5	1527(1009)	4	126(43)
anymemo-422	2	249	5	293(10)	5	296(6)	5	270(15)	4	255(15)
anymemo-440	8	934	3	1570(85)	3	1488(85)	0	>(-)	4	453(97)
notepad-23	6	216	5	333(167)	5	683(544)	5	920(671)	4	403(271)
olam-2	2	57	5	52(2)	4	50(1)	3	50(1)	4	56(8)
olam-1	2	35	5	27(1)	5	27(1)	3	27(1)	4	31(2)
FastAdapter-394	1	48	5	48(1)	5	455(374)	5	740(8)	3	243(308)
LibreNews-22	6	113	5	123(33)	5	176(77)	5	287(239)	4	253(282)
LibreNews-23	3	48	2	56(12)	2	62(4)	3	108(54)	4	63(10)
LibreNews-27	5	70	5	93(3)	5	88(1)	5	426(460)	4	74(4)
SMSSync-464	4	751	4	984(88)	4	1137(82)	3	1181(81)	4	2427(215)
transistor-63	3	41	5	52(21)	5	44(15)	5	52(20)	4	38(2)
zom-271	5	126	5	277(283)	5	202(74)	5	245(201)	4	185(64)
PixART-125	5	577	5	924(86)	5	1167(7)	5	1719(253)	3	1055(69)
PixART-127	5	138	5	435(337)	5	338(97)	5	803(536)	4	199(12)
ScreenCam-25	6	722	5	1545(943)	5	1261(42)	5	1265(37)	4	1158(30)
ventriloid-1	9	67	4	150(103)	4	108(83)	0	>(-)	4	56(1)
Nextcloud-487	2	63	5	310(461)	5	509(556)	5	1092(2)	4	2116(2467)
obdreader-22	8	892	5	1884(1717)	5	1862(1714)	3	1216(142)	3	976(153)
dagger-46	1	31	5	25(3)	5	24(1)	5	23(1)	4	29(4)
ODK-1042	2	72	4	74(1)	5	97(55)	2	77(4)	4	103(61)
ODK-2075	3	90	5	152(95)	5	164(60)	5	1015(1048)	3	135(73)
ODK-2086	3	90	4	644(757)	5	534(672)	5	812(989)	4	489(711)
ODK-2191	3	230	5	255(11)	5	266(14)	5	270(14)	3	135(73)
ODK-2525	2	81	5	687(136)	5	645(163)	5	448(257)	4	51(1)
ODK-2601	4	185	5	1186(1180)	4	1442(1109)	5	1871(2428)	4	271(132)
k9-3255	4	178	4	255(30)	3	487(463)	1	1022(-)	3	52(3)
k9-2612	2	103	5	152(20)	5	102(17)	5	1221(2550)	4	783(1466)
k9-2019	3	60	5	56(1)	5	55(0)	5	950(1214)	4	70(52)
Anki-4586	7	97	5	205(277)	5	275(324)	1	987(-)	4	116(1)
TagMo-12	2	15	5	14(0)	5	17(5)	5	14(0)	4	16(0)
FlashCards-13	3	64	5	140(11)	5	135(9)	5	137(10)	4	43(0)
Gnu-596	1	18	5	14(1)	4	14(0)	4	14(0)	4	15(2)
Gnu-633	3	84	5	81(2)	3	75(4)	1	142(-)	3	88(14)
timeTracker-35	4	1974	5	1276(797)	5	1032(845)	5	1484(698)	4	141(45)

materialistic-1067, AIMSICD-816) were due to the lack of support on these actions. We believe that ReCDroid+ can be extended to incorporate these actions with additional engineering effort. Second, ReCDroid+ cannot handle concurrency bugs or nondeterministic bugs [30, 31]. In our experiment, two fail-to-be-reproduced bug reports (Memento-169, Shortz-135) were due to non-determinism and one (ODK-1796) was due to a concurrency bug. For example, to trigger the crash in ODK-1796, it requires waiting on one screen for seconds and then clicking the next screen at a very fast speed. For concurrency bugs, we may leverage existing work on handling concurrency bugs in Android apps [39, 69, 71], such as allowing specific actions to wait for a certain time period before exploration.

Third, ReCDroid+ focuses on reproducing crashes. It does not generate automated test oracles from bug reports, so it is not able to reproduce non-crash bugs. Nevertheless, ReCDroid+ can still be useful in this case with certain human interventions. For example, during the automated dynamic

exploration, a developer can observe if a non-crashed bug (e.g., an error message) is reproduced. There has been some existing work [80] on generating test oracles for Android apps in certain circumstances, but no research has considered test oracles in reproducing bug reports. We will consider this research topic to be part of our future work. Finally, ReCDroid+ does not support highly specialized text inputs if the input is not specified in the bug report. Recent approaches in symbolic executions may prove useful in overcoming this limitation [40].

Threats to Validity. The primary threat to external validity for this study involves the representativeness of our apps and bug reports. However, we do reduce this threat to some extent by crawling bug reports from open source apps to avoid introducing biases. We cannot claim that our results can be generalized to all bug reports of all domains though. The primary threat to internal validity involves the confounding effects of participants. We assumed that the students participating in the study (for RQ3) were substitutes for developers. We believe the assumption is reasonable given that all 12 participants indicated that they had experience in Android programming. Recent work [62] has also shown that students can represent professionals in software engineering experiments. In addition, we used four case-insensitive keywords to search for crash bug reports, which may miss reports not containing these keywords. We can add more keywords to broaden the search, but it may involve additional human effort to filter out non-crash bug reports.

7 RELATED WORK

Related work has focused on augmenting bug reports for Android apps [56, 57]. Specifically, FUSION [56] leverages dynamic analysis to obtain GUI events of Android apps, and uses these events to help users auto-complete reproduction steps in bug reports. This approach helps end users to produce more comprehensive reports that will ease bug reproduction. However, this technique does not reproduce crashes from the original bug reports. We see our approach and FUSION as complementary, if users were to utilize FUSION, this would improve the overall quality of the bug reports and increase the success rate of our technique even further.

A tool called Yakusu [32] on translating executable test cases from bug reports presented in a recent paper is probably most related to our approach. However, the goal of Yakusu is translating test cases from bug reports instead of reproducing bugs (e.g., crashes) described in the bug report. Their dynamic search algorithm stops when all GUI components extracted from bug reports are explored regardless of whether the crash is found. Therefore, event sequences generated by Yakusu may not reproduce all relevant crashes. Regarding efficiency, for the same eight bug reports that ReCDroid+ can reproduce and that Yakusu can generate test cases, ReCDroid+ spent less time in six bug reports. In total, it took ReCDroid+ 1,370 seconds to reproduce the eight bug reports, whereas Yakusu spent 2,498 seconds translating the bug reports into test cases – 82% slower than ReCDroid+.

In addition, Yakusu does not extract input values for editable events. Instead, it will randomly send an input. In contrast, ReCDroid+ defines a family of grammar rules that can systematically extract the relevant inputs from bug reports. As our study (Finding 3) shows, a non-trivial portion of crashes involve specific user inputs. Moreover, Yakusu's ontology-based approach employs static analysis on the app source code. It matches the textual content extracted from the source code with the bug report. In contrast, ReCDroid+ does not rely on source code but obtains the textual content of GUI widgets at runtime. Furthermore, we conducted a more thorough empirical study to show how NLP uncovered bugs that would not be discovered otherwise. Moreover, we conducted a user study, although light-weighted, to show usefulness of ReCDroid+. Finally in terms of generality, the family of grammar rules derived by ReCDroid+ is from a large number of bug reports. We also provided empirical evidence to explain the assumption and the heuristics employed in ReCDroid+.

There has been considerable work on using NLP to summarize and classify bug reports [34, 61]. For example, Rastkar et al. [61] summarize bug reports automatically so that developers can perform their tasks by consulting shorter summaries instead of entire bug reports. Gegick et al. [34] use text mining to classify bug reports as either security- or non-security-related. Chaparro et al. [26] use several techniques to detect missing information from bug reports. PerfLearner [36] extracts execution commands and input parameters from descriptions of performance bug reports and use them to generate test frames for guiding actual performance test case generation. Zhang et al. [83] employ NLP to process bug reports and use search-based algorithm to infer models, which can be used to generate new test cases. While these techniques apply NLP techniques to analyze bug reports, they cannot synthesize GUI events from bug reports to help bug reproduction.

There are several techniques on using NLP to facilitate dynamic analysis [41, 75]. For example, PrefFinder [41] uses NLP and **information retrieval (IR)** to automatically find user preferences for correcting the configuration of a running system. DASE [75] aims to to extract input constraints from user manuals and uses the constraints to guide symbolic execution to avoid generating too many invalid inputs. However, these techniques make assumptions on the format of the textual description and none of them automatically reproduces bugs from bug reports.

To the best of our knowledge, EULER [25] and S2RMiner [84] are the only existing works that can automatically identify S2R sentences in bug reports. Our previous work, S2RMiner [84], employs **support vector machine (SVM)** to extract S2R sentences from a bug report. It combines n-grams and CountVectorizer [60] to transform text features into numerical features. However, the size of the training dataset is less than 500 bug reports. In contrast, ReCDroid+ employs a deep learning model designed by CNN and LSTM, which can potentially achieve better performance than the traditional SVM.

EULER leverages neural sequence labeling in combination with discourse patterns and dependency parsing to identify S2R sentences. Compared with EULER in identifying S2R and crash sentences, ReCDroid+ has several advantages. First, ReCDroid+ employs binary classification, whereas EULER employs multi-class classification to model the dependence among sentences. In general, it is computationally more expensive to solve a multi-class problem than a binary problem with the same size of data [21]. Second, EULER does not consider other characteristics of S2R sentences, such as listing symbols. Third, Name leverages a set of heuristic rules to refine the results output by the deep learning model, which can improve the accuracy of prediction.

Both EULER and S2RMiner cannot identify crash sentences, which is critical for automatically reproduce crashes.

There are tools for automatically reproducing in-field failures from various sources, including core dumps [73, 81], function call sequences [42], call stack [74], and runtime logs [77, 78]. However, none of these techniques can reproduce bugs from bug descriptions written in natural language. On the other hand, these techniques are orthogonal to ReCDroid+ and developers may decide which technique to use based on the information available in the bug report.

There has been a great deal of work on detecting bugs or achieving high coverage for Android applications using GUI testing [16, 20, 27, 29, 48–51, 66]. Some techniques [48, 49] utilize historical user event logs to improve code coverage. These techniques systematically explore the GUI events of the target app, guided by various advanced algorithms. However, none of these techniques reproduce issues directly from bug reports.

8 CONCLUSIONS AND FUTURE WORK

Wehave presented ReCDroid+, an automated approach to reproducing crashes from bug reports for Android applications. ReCDroid+ leverages natural language processing techniques and

heuristics to analyze bug reports and identify GUI events that are necessary for crash reproduction. It then directs the exploration of the corresponding app toward the extracted events to reproduce the crash. We have evaluated ReCDroid+ on 66 bug reports from 37 Android apps and showed that it successfully reproduced 42 crashes; 12 fail-to-be-reproduced bug reports were due to the limitations of the execution engines rather than ReCDroid+. A user study suggests that ReCDroid+ reproduced 18 crashes not reproduced by at least one developer and was preferred by developers over manual reproduction. Additional evaluation also indicates that ReCDroid+ is robust in handling low-quality bug reports.

As future work we intend to leverage the user reviews from App store to extract additional information for helping bug reproduction. We also intend to develop techniques to automatically extract grammar patterns from bug reports.

REFERENCES

[1] 2013. acv-11. https://github.com/robotmedia/droid-comic-viewer/issues/11.
[2] 2014. lxml.etree. https://lxml.de/tutorial.html.
[3] 2015. Mark message as unread make app crash. https://github.com/moezbhatti/qksms/issues/241.
[4] 2016. Google Code. https://code.google.com.
[5] 2016. Google Code Archive. https://code.google.com/archive/.
[6] 2018. UI Automator. https://github.com/xiaocong/uiautomator.
[7] 2018. Word2vec. https://github.com/dav/word2vec.
[8] 2019. ReCDroid. https://github.com/AndroidTestBugReport/ReCDroid.
[9] 2019. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.
[10] 2020. 22 patterns. https://github.com/AndroidTestBugReport/ReCDroid/blob/master/nlp%20pattern/grammar%20patterns.xlsx.
[11] 2020. APPLAUSE. https://www.applause.com/blog/app-abandonment-bug-testing.
[12] 2020. GitHub. https://github.com.
[13] 2020. Google Play Data. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store.
[14] 2020. maxPool. https://keras.io/api/layers/pooling_layers/.
[15] Charu C. Aggarwal and ChengXiang Zhai. 2012. A survey of text classification algorithms. In *Mining Text Data*. Springer, 163–222.
[16] 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 International Conference on Automated Software Engineering*. 258–261.
[17] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2015), 53–59.
[18] Vincenzo Ambriola and Vincenzo Gervasi. 1997. Processing natural language requirements. In *Proceedings of the International Conference Automated Software Engineering*. 36–46.
[19] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. 2009. DebugAdvisor: A recommender system for debugging. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering*. 373–382.
[20] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN Notices*, Vol. 48. 641–660.
[21] Younes Bennani and Khalid Benabdeslem. 2006. Dendogram-based SVM for multi-class classification. *Journal of Computing and Information Technology* 14, 4 (2006), 283–289.
[22] Robert L. Brennan and Dale J. Prediger. 1981. Coefficient kappa: Some uses, misuses, and alternatives. *Educational and Psychological Measurement* 41, 3 (1981), 687–699.
[23] Peter F. Brown, Peter V. Desouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. 1992. Class-based n-gram models of natural language. *Computational Linguistics* 18, 4 (1992), 467–479.
[24] Bugzilla 2016. Bugzilla keyword descriptions. https://bugzilla.mozilla.org/describekeywords.cgi.
[25] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. 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*. 86–96.

[26] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 396–407.

[27] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.

[28] Jason P. C. Chiu and Eric Nichols. 2016. Named entity recognition with bidirectional LSTM-CNNs. *Transactions of the Association for Computational Linguistics* 4 (2016), 357–370.

[29] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 623–640.

[30] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the International Conference on Automated Software Engineering*. 486–497.

[31] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the International Conference on Software Engineering*. 408–419.

[32] 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 SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.

[33] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *Thought-Works*. http://www.thoughtworks.com/ContinuousIntegration.pdf 122 (2006), 14.

[34] M. Gegick, P. Rotella, and T. Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *Proceedings of the International Working Conference on Mining Software Repositories*. 11–20.

[35] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM.

[36] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from bug reports to understand and generate performance test frames. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 17–28.

[37] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. 204–217.

[38] Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. *To Appear*.

[39] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race detection for event-driven mobile applications. *ACM SIGPLAN Notices* 49, 6 (2014), 326–336.

[40] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. 2012. *SymDroid: Symbolic Execution for Dalvik Bytecode*. Technical Report.

[41] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. 2014. PrefFinder: Getting the right preference in configurable software systems. In *Proceedings of the International Conference on Automated Software Engineering*. 151–162.

[42] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the International Conference on Software Engineering*. 474–484.

[43] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[44] Anne Kao and Steve R. Poteet. 2007. *Natural Language Processing and Text Mining*. Springer Science & Business Media.

[45] KBP 2012. Knowledge Base Population. https://nlp.stanford.edu/projects/kbp/.

[46] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[47] Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics* 3 (2015), 211–225.

[48] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. A deep learning based approach to automated Android app testing. *arXiv E-prints* (2019), arXiv–1901.

[49] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining Android app usages for generating actionable GUI-based execution scenarios. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 111–122.

[50] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 224–234.

[51] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of Android apps. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.

[52] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the International Symposium on Software Testing and Analysis*. 94–105.
[53] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering* 43, 9 (2016), 817–847.
[54] Andrew McCallum, Kamal Nigam, et al. 1998. A comparison of event models for Naive Bayes text classification. In *AAAI-98 Workshop on Learning for Text Categorization*, Vol. 752. Citeseer, 41–48.
[55] Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, et al. 2015. Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23, 3 (2015), 530–539.
[56] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing bug reports for Android applications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 673–686.
[57] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing Android application crashes. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 33–44.
[58] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology* 4, 1 (2008), 13–20.
[59] Frank Padberg, Philip Pfaffe, and Martin Blersch. 2013. On mining concurrency defect-related reports from bug repositories. 10.
[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[61] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2014. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering* 40, 4 (2014), 366–380.
[62] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *Proceedings of the International Conference on Software Engineering-Volume 1*. 666–676.
[63] Shai Shalev-Shwartz and Tong Zhang. 2013. Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research* 14, Feb (2013), 567–599.
[64] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.
[65] Ting Su. 2016. FSMdroid: Guided GUI testing of Android apps. In *Proceedings of the International Conference on Software Engineering Companion*. 689–691.
[66] 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 Joint Meeting on Foundations of Software Engineering*. 245–256.
[67] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the International Conference on Automated Software Engineering*. 253–262.
[68] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. iComment: Bugs or bad comments? In *ACM SIGOPS Operating Systems Review*, Vol. 41. 145–158.
[69] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating test cases to expose concurrency bugs in Android applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 648–653.
[70] Jos Van Der Westhuizen and Joan Lasenby. 2018. The unreasonable effectiveness of the forget gate. *arXiv preprint arXiv:1804.04849*.
[71] Jue Wang, Yanyan Jiang, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu. 2018. AATT+: Effectively manifesting concurrency bugs in Android apps. *Science of Computer Programming* 163 (2018), 1–18.
[72] Xuerui Wang, Andrew McCallum, and Xing Wei. 2007. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Proceedings of the International Conference on Data Mining*. 697–702.
[73] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ACM SIGPLAN Notices*, Vol. 45. 155–166.
[74] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating reproducible and replayable bug reports from Android application crashes. In *Proceedings of the International Conference on Program Comprehension*. 48–59.
[75] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. DASE: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the International Conference on Software Engineering*. 620–631.

[76] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. 250–265.
[77] Tingting Yu, Tarannum S. Zaman, and Chao Wang. 2017. DESCRY: Reproducing system-level concurrency failures. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 694–704.
[78] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the International Conference on Software Engineering*. 102–112.
[79] Hrushikesh Zadgaonkar. 2013. *Robotium Automated Testing for Android*. Packt Publishing Ltd.
[80] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated generation of oracles for testing user-interaction features of mobile apps. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. 183–192.
[81] Cristian Zamfir and George Candea. 2010. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems*. 321–334.
[82] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems*. 649–657.
[83] Yuanyuan Zhang, Mark Harman, Yue Jia, and Federica Sarro. 2015. Inferring test models from Kate's bug reports using multi-objective search. In *Proceedings of the International Symposium on Search Based Software Engineering*. 301–307.
[84] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically extracting bug reproducing steps from Android bug reports. In *International Conference on Software and Systems Reuse*. Springer, 100–111.
[85] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. 2015. A C-LSTM neural network for text classification. *arXiv preprint arXiv:1511.08630*.

Received September 2020; revised September 2021; accepted September 2021