ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports

Yu Zhao*, Tingting Yu*, Ting Su†, Yang Liu†, Wei Zheng‡, Jingzhi Zhang‡, William G.J. Halfond§,
* University of Kentucky, USA - † Nanyang Technological University, Singapore
‡ Northwestern Polytechnical University, China - § University of Southern California, USA

Abstract—**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 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) and dynamic GUI exploration to synthesize event sequences with the goal of reproducing the reported crash. We have evaluated ReCDroid on 51 original bug reports from 33 Android apps. The results show that ReCDroid successfully reproduced 33 crashes (63.5% success rate) directly from the textual description of bug reports. A user study involving 12 participants demonstrates that ReCDroid can improve the productivity of developers when resolving crash bug reports.**

I. Introduction

Mobile applications (apps) have become extremely popular – in 2017 there were over 2.2 million apps in Google Play's app store [5]. 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 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 [1]. 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 [17], Google Code Issue Tracker [3], and Github Issue Tracker [2]). 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 [40]. 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 [13]. 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. ReCDroid leverages several natural language processing (NLP) techniques to analyze the text of the reports and automatically identify GUI components and related information (e.g., input values) that are necessary to reproduce the crashes. ReCDroid then 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 [8]).

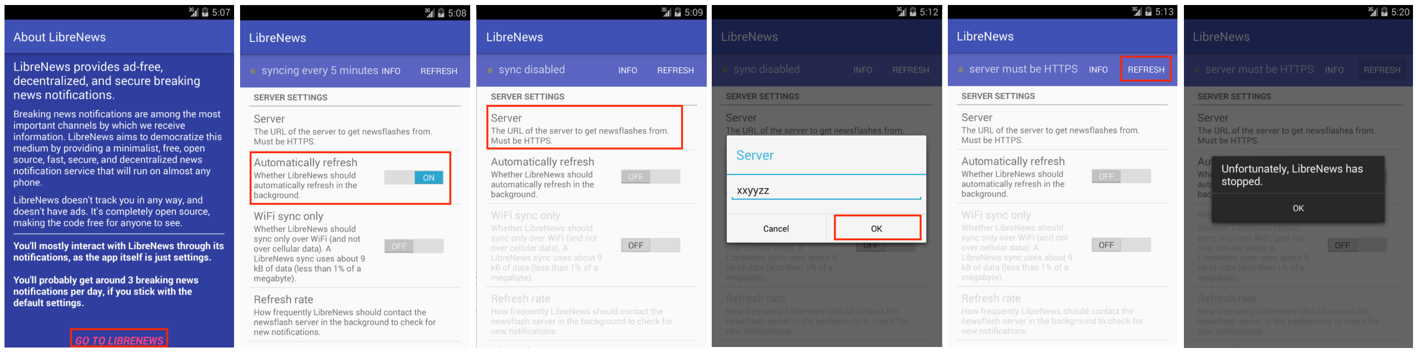
ReCDroid differs from prior work for analyzing the reproducibility of bug reports [18], [39] because most existing techniques focus on improving the quality of bug reports. None of them have considered using information from bug reports to automatically guide bug reproduction. In contrast, ReCDroid takes 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 [18], [23], [26], [42] because they focus extracting useful information (e.g., test cases [23]) without directly targeting at reproducing crashes.

ReCDroid has been implemented as a software tool on top of two execution engines — Robotium [55] and UI Automator [8]. To determine the effectiveness of our approach, we ran ReCDroid on 51 bug reports from 33 popular Android apps. ReCDroid was able to successfully reproduce 33 (63.5%) of the crashes. Furthermore, 12 out of the 18 crashes could have been reproduced by ReCDroid if limitations in the implementation of the execution engines were to be removed.

To determine the usefulness of our tool, we conducted a light-weighted user study that showed that ReCDroid can reproduce 18 crashes not reproduced by at least one developer and was highly preferred by developers in comparison to a manual process. We also found that ReCDroid was highly robust in handling situations where reduced amounts of information were provided in the reports. 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.

In summary, our paper makes the following contributions:

- The design and development of a novel approach to automatically reproduce crash failures for Android apps directly from the textual description of bug reports.
- An empirical study showing that ReCDroid is effective at reproducing Android crashes and likely to improve the


(a)

5:07

About LibreNews

LibreNews provides ad-free, decentralized, and secure breaking news notifications.

Breaking news notifications are among the most important channels by which we receive information. LibreNews aims to democratize this medium by providing a minimalist, free, open source, fast, secure, and decentralized news notification service that will run on almost any phone.

LibreNews doesn't track you in any way, and doesn't have ads. It's completely open source, making the code free for anyone to see.

You'll mostly interact with LibreNews through its notifications, as the app itself is just settings.

You'll probably get around 3 breaking news notifications per day, if you stick with the default settings.

GO TO LIBRENEWS

(b)

5:08

LibreNews

syncing every 5 minutes INFO REFRESH

SERVER SETTINGS

Server
The URL of the server to get newsflashes from. Must be HTTPS.

Automatically refresh
Whether LibreNews should automatically refresh in the background. ON

WiFi sync only
Whether LibreNews should sync only over WiFi (and not over cellular data). A LibreNews sync uses about 9 kB of data (less than 1% of a megabyte). OFF

Refresh rate
How frequently LibreNews should contact the newsflash server in the background to check for new notifications.

(c)

5:09

LibreNews

sync disabled INFO REFRESH

SERVER SETTINGS

Server
The URL of the server to get newsflashes from. Must be HTTPS.

Automatically refresh
Whether LibreNews should automatically refresh in the background. OFF

WiFi sync only
Whether LibreNews should sync only over WiFi (and not over cellular data). A LibreNews sync uses about 9 kB of data (less than 1% of a megabyte). OFF

Refresh rate
How frequently LibreNews should contact the newsflash server in the background to check for new notifications.

(d)

5:12

LibreNews

sync disabled INFO REFRESH

SERVER SETTINGS

Server

xxyyzz

Cancel OK

(e)

5:13

LibreNews

server must be HTTPS INFO REFRESH

SERVER SETTINGS

Server
The URL of the server to get newsflashes from. Must be HTTPS.

Automatically refresh
Whether LibreNews should automatically refresh in the background. OFF

WiFi sync only
Whether LibreNews should sync only over WiFi (and not over cellular data). A LibreNews sync uses about 9 kB of data (less than 1% of a megabyte). OFF

Refresh rate
How frequently LibreNews should contact the newsflash server in the background to check for new notifications.

(f)

5:20

LibreNews

server must be HTTPS INFO REFRESH

SERVER SETTINGS

Server
The URL of the server to get newsflashes from. Must be HTTPS.

Unfortunately, LibreNews has stopped.

OK

Fig. 1: The steps of reproducing the crash described in Fig. 2.

productivity of bug resolution.

- The implementation of our approach as a publicly available tool, ReCDroid, along with all experiment data (e.g., datasets, user study) [7].

II. Overview

A. Observations

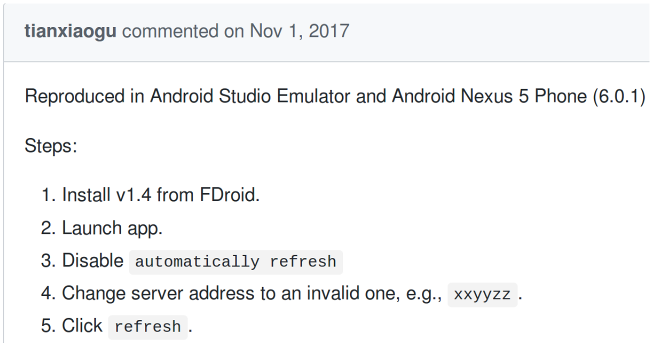
As the first step, we spent a month studying a large number of Android bug reports to understand their characteristics for guiding the design and implementation of ReCDroid.

We collected Android apps from both Google Code Archive [4] and GitHub [2]. We crawled the bug reports from the first 50 pages in Google Code, resulting in 7666 bug reports. We then searched Android apps from GitHub by using the keyword "Android", resulting in 3233 bug reports. Among all 10899 bug reports, we used keywords, such as "crash" and "exception" to search for reports involving app crashes. This yielded a total number of 1038 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 1038 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).

B. Design Challenges

An example bug report is shown in Fig. 2. 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 Fig. 1 for triggering the crash. To achieve this goal, our approach must address four main challenges. First, what types of information need to be extracted from a bug report? Second, how can such information be extracted from reports written in natural language? Third, how can this information, which may vary


tianxiaogu commented on Nov 1, 2017

Reproduced in Android Studio Emulator and Android Nexus 5 Phone (6.0.1)

Steps:

1. Install v1.4 from FDroid.
2. Launch app.
3. Disable `automatically refresh`
4. Change server address to an invalid one, e.g., `xxyyzz` .
5. Click `refresh` .

Fig. 2: Bug Report for LibreNews issue#22

in specificity and completeness, be used to reproduce the crash? Fourth, how can this process be done efficiently in terms of a minimal reproduction sequence and the time to find this sequence? In the remainder of this section, we provide an overview of how our approach's design addresses these challenges. Details and algorithms of our approach are presented in Section III.

What type of information to extract? 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.

To illustrate, consider the fourth step in Fig. 2. Here, "change" is the user action, "Server" is the target GUI component, and "xxyyzz" is the input value.

How to map 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

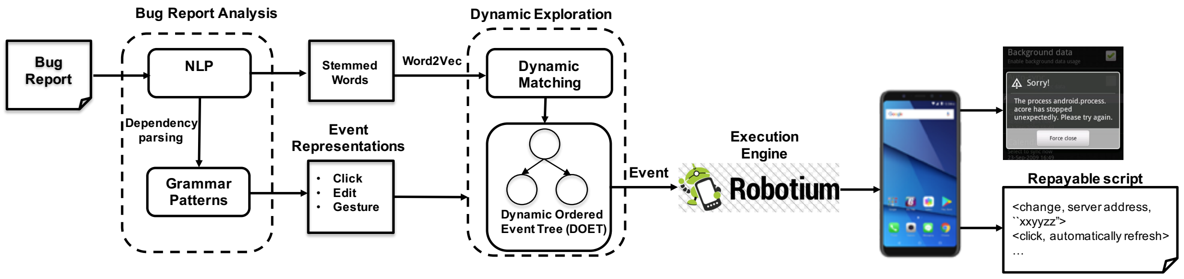


Fig. 3: Overview of the ReCDroid Framework.

"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 [33], [38] 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 [28] 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 22 typical patterns, which we summarize and describe in Section III. 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 [34] to determine whether two words are semantically related. For example, the words "movie" and "film" have a fairly high similarity.

How to create 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 [12], [15], [27], [37], [52] are not a good fit for this particular need. For example, many existing tools (e.g., A3E [15]) 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.

How to make 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., [12], [15], [52]) that add backtracking steps to their crawling sequence, which results in an overall much longer sequence of reproducing actions.

III. ReCDroid Approach

The architecture of ReCDroid is shown in Fig. 3. ReCDroid consists of two major phases — bug report analysis and dynamic exploration. To carry out the bug report analysis, ReCDroid employs NLP techniques to extract GUI event representations from bug reports. 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.

TABLE I: Summary of Grammar Patterns

Category	*ID*	*Pct.*	*Grammar Pattern*	*Example*	Event Tuple
Click	CR1	12.5%	action → dobj (→NP)	Click$_{[action]}$ {easy level$_{[dobj]}$}$_{[NP]}$	<click, (easy) level>
	CR2	0.7%	action → nsubjapss (→ NP)	{Easy level$_{[dobj]}$}$_{[NP]}$ is clicked$_{[action]}$	<click, (easy) level>
	CR3	8.6%	action → pobj (→ NP)	I made a click$_{[action]}$ on {easy level$_{[pobj]}$}$_{[NP]}$	<click, (easy) level>
Edit	TR1	7.3%	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	1.8%	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	0.7%	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	0.6%	TR1|TR2 + *EG*(NOUN → NUM → UNIT | STR)	Input$_{[action]}$ a number$_{[dobj]}$ to kilometer$_{[pobj]}$, e.g., {10$_{[NUM]}$ km$_{[UNIT]}$}$_{EG}$	<input, kilometer|km, 10>
Gesture	NR1	0.4%	action	Rotate$_{[action]}$ the screen	<rotate>

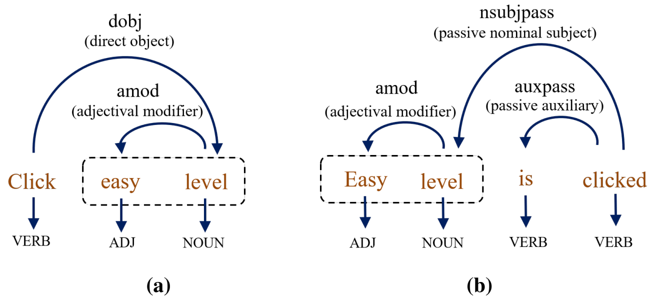


Fig. 4: Examples of Dependency Trees

A. Phase 1: Analyzing Bug Reports

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

1) Grammar Patterns: The 22 grammar patterns were derived from the corpus of 813 Android bug reports described in Section II-A. 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. Fig. 4 shows an example of the results of the SpaCy dependency analysis on two sentences with different structures.

Broadly, the grammar patterns could be grouped intro 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 I lists the eight typical grammar patterns (The full list can be found in our artifacts [7]). Column 3 shows the percentage of the 813 bug reports in which each grammar pattern applies. We next describe these patterns.

Click Events. ReCDroid uses seven grammar patterns to extract the click event tuple. The "input" element in the tuple is not applicable to click events. In Table I, 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 14 grammar patterns for extracting edit events. In Table I, 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 if the NOUN is a number (NUM), it is identified as an input value. ReCDroid then searches for the word right after the number and if the word is a unit (UNIT∈{kg, cm, litter}), it is added as a target GUI component. Otherwise, if no numbers are found in EG, the whole phrase EG is identified as a regular string input (STR).

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.

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 [28]. It then applies *stemming* [32][1] 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 [7].

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

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 Fig. 2 contains a word "change", so the grammar pattern TR3 is applied.

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 [47], 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 III-B2 illustrates how ReCDroid obtains additional information from unstructured texts to address the mismatch between bug reports and target apps.

B. Phase 2: 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.

During the exploration, ReCDroid iteratively selects, for each screen, the most relevant component to execute. 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 always select the leftmost relevant tree leaf to explore without iterating through all components on the screen.

1) ReCDroid' 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.

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*/
```

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 III-B4), 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 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 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. 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

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 $false$
```

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.

2) Dynamic Matching: To determine whether a GUI component matches a bug report (Line 11), ReCDroid utilizes `Word2Vec` [34], a word embedding technique, to check if the name (i.e., the displayed text) of a GUI component is semantically similar with any of the GUI components from the extracted event representations or the words from sentences in which grammar patterns cannot be used. The `Word2Vec` model is trained from a public dataset *text8* containing 16 million words and is provided along with the source code of `Word2Vec` [10]. 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[2], from both 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 [14], [41], [46].

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", "quote" because

[2]An n-gram is a contiguous sequence of n items from a given sequence of text, which has been widely used in information retrieval [48] and natural language processing [16].

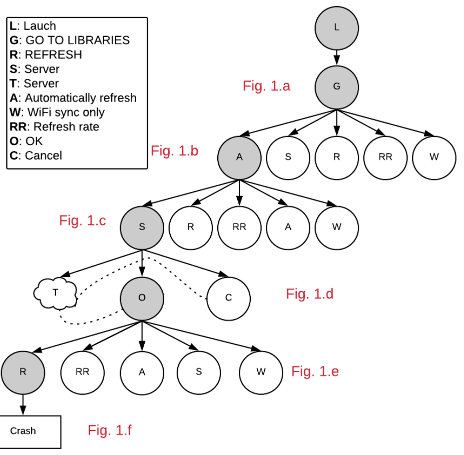


Fig. 5: Dynamic Ordered Event Tree (DOET) for Figure 1

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) A Running Example: Fig. 5 shows a partial DOET for the example in Fig. 1. 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 Fig. 1a. 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 G to $ready$ and continues the search (Lines 17–18). In the 2nd iteration, ReCDroid clicks component G and brings the app to Fig. 1b. 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 (Fig. 1b) and selects the leftmost leaf (A) to execute, which brings the app to the screen of Fig. 1c. At this tree level, A is placed on the right because it has been explored before. In the 4th iteration, exercising the leftmost leaf node S brings the app to Fig. 1d, 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 (Fig. 1d), 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 Fig. 1c, 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 Fig. 1b would be selected. So in the next iteration, ReCDroid restarts the search and executes the sequence $L \rightarrow G \rightarrow S$.

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. We omit the details of the loop detection algorithm due to space limitations.

IV. Empirical Study

To evaluate ReCDroid, we consider four research questions:

RQ1: How effective and efficient is ReCDroid at reproducing crashes in bug reports?

RQ2: To what extent do the NLP techniques in ReCDroid affect its effectiveness and efficiency?

RQ3: Does ReCDroid benefit developers compared to manual reproduction?

RQ4: Can ReCDroid reproduce crashes from different levels of low-quality bug reports?

A. 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 330 bug reports containing the keywords "crash" and "exception" from GitHub. We next included all 15 bug reports from the FUSION paper [39] and 25 bug reports from a recent paper on translating Android bug reports into test cases [23]. 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.

We then manually filtered the 370 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 298 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 51 bug reports from 33 apks. The cost of the manual process is quite high: the preparation of the dataset required around 400 hours of researcher time.

B. Implementation

We conducted our experiment on a physical x86 machine running with Ubuntu 14.04. The NLP techniques of ReCDroid was implemented based on the spaCy dependency parser [28]. The dynamic exploration component was implemented on top of two execution engines, Robotium [55] and UI Automator [8], 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.

C. Experiment Design

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., two hours) and efficiency in terms of the time it took to reproduce each crash.

2) RQ2: 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 VI). The time limits for running ReCDroid$_N$ and ReCDroid$_D$ were also set to two hours.

3) RQ3: Usefulness of ReCDroid: The goal of RQ3 is to evaluate the experience developer had using ReCDroid to reproduce bugs compared to using manual reproduction. We recruited 12 graduate students as the participants. All had at least 6-month Android development experience and three were real Android developers working in companies for 3 years before entering graduate school. Each participant read the 39 bug reports and tried to manually reproduce the crashes. All apps were preinstalled. For each bug report, the 12 participants 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 reproduced. After the participants attempted to reproduce all bugs, they were asked to use ReCDroid on the 39 bug reports. This was followed by a survey question: would you prefer to use ReCDroid to reproduce bugs from bug reports over manual reproduction? Note that to avoid bias, the participants were not aware of the purpose of this user study.

4) RQ4: Handling Low-Quality Bug Reports: The goal of RQ4 is to assess the ability of ReCDroid to handle different levels of low-quality bug reports. 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. We focused on removing words from texts containing reproducing steps. Specifically, we considered three variations for each of the 33 bug report reports reproduced by ReCDroid in order to mimic different levels of quality: 1) removing 10% of the words in the report, 2) removing 20% of the words in the report, and 3) removing 50% of the words in the report. Due to the randomization of removing words from

TABLE II: RQ1 — RQ3: Different Techniques and User Study

#BR.	*# steps*	*# ptn*	*Reproduce Success*			*# Events in Sequence*			*Time (Seconds)*			User
			RD	*RD_N*	*RD_D*	*RD*	*RD_N*	*RD_D*	*RD*	*RD_N*	*RD_D*	(12)
newsblur-1053	5	7 (3, 2)	✓	✓	✓	7	7	7	157.6	133.5	132.3	12
markor-194	3	1 (0, 1)	✓	N	N	4	-	-	1180.8	>	>	12
birthdroid-13	1	5 (3, 0)	✓	✓	✓	5	8	8	106.5	483.7	1088.5	9
car-report-43$\star$	4	4 (0, 0)	✓	✓	✓	16	16	16	309.5	299.4	101	8
opensudoku-173	8	10 (0, 2)	✓	N	N	9	-	-	576.4	>	>	10
acv-11$\star$	5	8 (1, 2)	✓	✓	✓	8	8	5	500.5	489.3	2060.1	7
anymemo-18	1	2 (0, 0)	✓	✓	✓	3	3	3	67.1	60.9	797.7	11
anymemo-440	4	6 (0, 1)	✓	✓	N	8	8	-	933.9	889.4	-	12
notepad-23$\star$	3	3 (0, 1)	✓	✓	✓	6	6	6	216.2	292	1731.1	11
olam-2$\star$	1	2 (0, 0)	✓	N	N	2	-	-	56.7	>	>	7
olam-1	1	1 (0, 1)	✓	N	N	2	-	-	35.2	>	>	11
FastAdapter-394	1	0 (0, 1)	✓	✓	✓	1	1	1	47.6	27.6	445.1	9
LibreNews-22	4	6 (2, 1)	✓	✓	✓	6	6	5	113.2	138.2	728.5	12
LibreNews-23	6	4 (2, 1)	✓	N	N	3	-	-	47.7	>	>	12
LibreNews-27	4	4 (2, 1)	✓	✓	✓	5	5	5	70.2	67.6	1074.7	11
SMSSync-464	2	0 (0, 2)	✓	✓	✓	4	4	4	751	703.3	5193.8	10
transistor-63	5	8 (6, 2)	✓	✓	✓	3	3	3	41.1	36.6	65.1	12
zom-271	5	1 (0, 1)	✓	✓	✓	5	5	5	125.5	115.8	507.7	11
PixART-125	3	5 (0, 1)	✓	✓	✓	5	5	5	576.9	649.6	1031.6	12
PixART-127$*$	3	5 (1, 0)	✓	✓	✓	5	5	5	137.6	147.1	991.9	12
ScreenCam-25$*$	3	2 (1, 1)	✓	✓	N	6	6	-	721.7	833.3	>	11
ventriloid-1	3	3 (1, 2)	✓	N	N	9	-	-	66.8	>	>	11
Nextcloud-487	1	2 (3, 1)	✓	✓	✓	2	2	2	63.3	72	943.8	11
obdreader-22	4	0 (0, 4)	✓	✓	N	8	8	-	891.7	939.7	>	12
dagger-46$*$	1	3 (2, 0)	✓	✓	✓	1	1	1	31.1	26.2	20.6	12
ODK-2086	2	3 (0, 0)	✓	✓	✓	3	3	3	89.6	91.2	2982	12
k9-3255	2	5 (1, 1)	✓	N	N	4	-	-	177.6	>	>	12
k9-2612$*$	4	3 (0, 0)	✓	✓	✓	2	3	2	103	134.6	5730.6	10
k9-2019$*$	1	0 (0, 1)	✓	✓	✓	3	3	3	59.5	58.1	1352.4	11
Anki-4586$*$	5	3 (1, 0)	✓	✓	N	7	7	-	96.6	100	>	12
TagMo-12$*$	1	2 (0, 0)	✓	✓	✓	2	2	2	14.8	15.7	29.6	12
openMF-734$*$	2	2 (0, 1)	✓	✓	✓	2	2	2	81.9	83.6	440.5	11
FlashCards-13$*$	4	3 (2, 0)	✓	✓	✓	3	3	3	63.5	93.9	93.7	12
FastAdaptor-113	2	3 (1, 1)	N	N	N	-	-	-	>	>	>	7
Memento-169	3	7 (3, 0)	N	N	N	-	-	-	>	>	>	2
ScreenCam-32	1	0 (0, 1)	N	N	N	-	-	-	>	>	>	10
ODK-1796	2	1 (0, 1)	N	N	N	-	-	-	>	>	>	4
AIMSICD-816	3	2 (0, 1)	N	N	N	-	-	-	>	>	>	1
materialistic-76	6	7 (2, 1)	N	N	N	-	-	-	>	>	>	5
Total	-	-	33	26	22	-	-	-	-	-	-	-

RD.= ReCDroid. "✓"=Crash reproduced. "N"=Crash not reproduced. "-"=Not applicable. ">"=exceeded time limit (2 hours).

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 495 ($33 \times 3 \times 5$) bug reports. Again, the time limit was set to 2 hours.

V. Results and Analysis

Table II summarizes the results of applying ReCDroid, ReCDroid$_N$, and ReCDroid$_D$ in 39 out of the 51 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., [6]). Columns 2–3 show the number of reproducing steps in each bug report and the number of unique grammar patterns applicable to each bug report. The numbers in the parenthesis of Column 3 indicate the number of false positives (left) and false negatives (right) when applied the grammar patterns. A false positive means that a grammar pattern is applied but the identified text is irrelevant to bug reproduction. A false negative means that a relevant reproducing step is not identified by any grammar patterns. Columns 4–12 show whether the technique successfully reproduced the crash, the size of the event sequence, and the time each technique took.

1) RQ1: Effectiveness and Efficiency of ReCDroid: As Table II shows, ReCDroid reproduced 33 out of 39 crashes; a success rate of 84.6%. The time required to reproduce the crashes ranged from 14 to 1,180 seconds with an average time of 257.9 seconds. All four crash bug reports (marked with $\star$) from the FUSION paper [39] and nine bug reports (marked with $*$) from Yakusu [23] were successfully reproduced. The results indicate that *ReCDroid is effective in reproducing crashes from bug reports*. The six cases where ReCDroid failed will be discussed in Section VI.

2) RQ2: The Role of NLP in ReCDroid: When compared ReCDroid to ReCDroid$_N$ and ReCDroid$_D$, ReCDroid successfully reproduced 26.9% and 50% 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 4% smaller than ReCDroid$_N$ and 1% bigger than ReCDroid$_D$. Both ReCDroid$_N$ and ReCDroid$_D$ generated short event sequences because like ReCDroid, they do not backtrack. Instead, whenever a back-track was needed, they restarted the search from the home screen of the app (Algorithm 1). With regards to efficiency, ReCDroid required 62.6% less time than ReCDroid$_N$ and 86.4% 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 (Column 3), since false positives may misguide the search and false negatives may jeopardize the search efficiency (certain useful information is missing). In the 33 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.

3) RQ3: Usefulness of ReCDroid: The last column of Table II shows the number of participants (out of 12) that successfully reproduced the crashes. While all crashes were reproduced by the participants, among all 33 crashes reproduced by ReCDroid, 18 of them failed to be reproduced by at least one participant. For the seven 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 cannot be reproduced by the developers.* 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 3 and 8 in Table II indicate, in 25 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 39 bug reports. The results show that the time for successful manual reproduction ranged from 9 seconds to 1,640 seconds, with an average 248.1 seconds — 3.7% 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 reproduc-*

ing crashes. Among all 33 crashes successfully reproduced by ReCDroid, the reproduction time required by individual participants ranged from 9 to 1,640 seconds. In fact, two out of the 12 participants spent a little more time (2% on average) than ReCDroid.

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 [24] 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.*

4) RQ4: Handling Low-Quality Bug Reports: Columns 2–7 of Table III 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 495 mutated bug reports for the three quality levels, ReCDroid was able to reproduce 94%, 92%, and 81% of the bug crashes, respectively. Even when 50% of the words were removed, ReCDroid could still successfully reproduce 25 crashes. The slowdowns caused by the missing information with respect to the original bug reports were only 1.7x, 2.2x, and 2.9x, respectively. These results suggest that *ReCDroid can be used to effectively handle low-quality bug reports with different levels of missing information.*

TABLE III: RQ4: Different Quality Levels

#BR.	*QL-10% (5)*		*QL-20% (5)*		*QL-50% (5)*	
	Success	*Time (sec)*	*Success*	*Time (sec)*	*Success*	*Time (sec)*
newsblur-1053	5	196(102)	5	94(50)	5	136(88)
markor-194	5	1601(24)	4	1564(85)	4	1608(30)
birthdroid-13	5	159(128)	5	383(205)	5	659(185)
car-report-43	5	280(3)	5	288(6)	5	286(1)
opensudoku-173	5	770(458)	3	2267(1153)	3	2325(1636)
acv-11	5	1077(1299)	5	1844(1448)	5	1911(1321)
anymemo-18	5	90(49)	5	62(9)	5	1527(1009)
anymemo-440	3	1570(85)	3	1488(85)	0	>(-)
notepad-23	5	333(167)	5	683(544)	5	920(671)
olam-2	5	52(2)	4	50(1)	3	50(1)
olam-1	5	27(1)	5	27(1)	3	27(1)
FastAdapter-394	5	48(1)	5	455(374)	5	740(8)
LibreNews-22	5	123(33)	5	176(77)	5	287(239)
LibreNews-23	2	56(12)	2	62(4)	3	108(54)
LibreNews-27	5	93(3)	5	88(1)	5	426(460)
SMSSync-464	4	984(88)	4	1137(82)	3	1181(81)
transistor-63	5	52(21)	5	44(15)	5	52(20)
zom-271	5	277(283)	5	202(74)	5	245(201)
PixART-125	5	924(86)	5	1167(7)	5	1719(253)
PixART-127	5	435(337)	5	338(97)	5	803(536)
ScreenCam-25	5	1545(943)	5	1261(42)	5	1265(37)
ventriloid-1	4	150(103)	4	108(83)	0	>(-)
Nextcloud-487	5	310(461)	5	509(556)	5	1092(2)
obdreader-22	5	1884(1717)	5	1862(1714)	3	1216(142)
dagger-46	5	25(3)	5	24(1)	5	23(1)
ODK-2086	4	644(757)	5	534(672)	5	812(989)
k9-3255	4	255(30)	3	487(463)	1	1022(-)
k9-2612	5	152(20)	5	102(17)	5	1221(2550)
k9-2019	5	56(1)	5	55(0)	5	950(1214)
Anki-4586	5	205(277)	5	275(324)	1	987(-)
TagMo-12	5	14(0)	5	17(5)	5	14(0)
openMF-734	5	82(1)	5	155(162)	5	82(1)
FlashCards-13	5	140(11)	5	135(9)	5	137(10)

VI. 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, 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 [21], [22]. In our experiment, three fail-to-be-reproduced bug reports (Memento-169, ScreenCam-32) 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. In some cases, heuristics can be added to handle timing issues, 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. Fourth, 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 [29]. Finally, ReCDroid is targeted at bug reports containing natural language description of reproducing steps. In the absence of reproducing steps, ReCDroid would act as a generic GUI exploration and testing tool (i.e., RD_D in the experiment).

Android Testing Tools. As a generic GUI exploration and testing tool, ReCDroid$_D$ is similar to existing Android testing tools [9], [11], [15], [37], [44], [45], which detect crashes in an unguided manner. ReCDroid$_D$ was shown to be competitive with Monkey [9], Sapienz [37], and the recent work Stoat [45] on our experiment subjects. Specifically, ReCDroid$_D$ reproduced 7 more crashes than Stoat, 7 more crashes than Sapienz, and 9 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 6.2% more time than Stoat, 27.6% less time than Sapienz, and 37.7% less time than Monkey. The details can be found in our released artifacts [7].

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 [43] has also shown that students can represent professionals in software engineering experiments.

VII. Related Work

Related work has focused on augmenting bug reports for Android apps [39], [40]. Specifically, FUSION [39] 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 [23] 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. Therefore, event sequences generated by Yakusu may not reproduce all relevant crashes. 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, 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. Furthermore, 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 [25], [42]. For example, Chaparro et al. [18] use several techniques to detect missing information from bug reports. PerfLearner [26] 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. [57] 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 [30], [51]. For example, DASE [51] 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.

There are tools for automatically reproducing in-field failures from various sources, including core dumps [49], [56], function call sequences [31], call stack [50], and runtime logs [53], [54]. 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 [11], [15], [19], [20], [35], [36], [45]. 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.

VIII. Conclusions and Future Work

We have 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 51 bug reports from 33 Android apps and showed that it successfully reproduced 33 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.

Acknowledgments

We would like to thank Xiaoxue Wu for helping to find benchmarks for the empirical study. We are also grateful to Ramakanth Kavuluru for helpful discussions on NLP techniques. This research is supported in part by the NSF grant CCF-1652149 and NTU research grant NGF-2017-03-033.

References

[1] APPLAUSE. https://www.applause.com/blog/app-abandonment-bug-testing.

[2] GitHub. https://github.com.

[3] Google Code. https://code.google.com.

[4] Google Code Archive. https://code.google.com/archive/.

[5] Google Play Data. https://en.wikipedia.org/wiki/Google_Play.

[6] Mark message as unread make app crash. https://github.com/moezbhatti/qksms/issues/241.

[7] ReCDroid. https://github.com/AndroidTestBugReport/ReCDroid.

[8] UI Automator. https://developer.android.com/training/testing/ui-automator.

[9] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[10] Word2vec. https://github.com/dav/word2vec.

[11] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the International Conference on Automated Software Engineering*, pages 258–261, 2012.

[12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[13] Vincenzo Ambriola and Vincenzo Gervasi. Processing natural language requirements. In *Proceedings of the International Conference Automated Software Engineering*, pages 36–46, 1997.

[14] B Ashok, Joseph Joy, Hongkang Liang, Sriram K Rajamani, Gopal Srinivasa, and Vipindeep Vangala. Debugadvisor: a recommender system for debugging. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering*, pages 373–382, 2009.

[15] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660, 2013.

[16] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.

[17] Bugzilla keyword descriptions, 2016. https://bugzilla.mozilla.org/describekeywords.cgi.

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

[19] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Weihe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for Android apps. In *Proceedings of the International Conference on Software Engineering, ICSE 2019*, page To appear, 2019.

[20] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

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

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

[23] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis*, pages 141–152, 2018.

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

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

[26] Xue Han, Tingting Yu, and David Lo. Learning from bug reports to understand and generate performance test frames. In *Proceedings of the International Conference on Automated Software Engineering*, 2018.

[27] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the International Conference on Mobile systems, Applications, and Services*, pages 204–217, 2014.

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

[29] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. Symdroid: Symbolic execution for dalvik bytecode. Technical report, 2012.

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

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

[32] Anne Kao and Steve R Poteet. *Natural language processing and text mining*. Springer Science & Business Media, 2007.

[33] Knowledge Base Population, 2012. https://nlp.stanford.edu/projects/kbp/.

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

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

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

[37] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[38] Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, et al. Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):530–539, 2015.

[39] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Auto-completing bug reports for android applications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 673–686, 2015.

[40] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 33–44, 2016.

[41] Frank Padberg, Philip Pfaffe, and Martin Blersch. On mining concurrency defect-related reports from bug repositories. 10, 2013.

[42] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014.

[43] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the International Conference on Software Engineering-Volume 1*, pages 666–676, 2015.

[44] Ting Su. Fsmdroid: Guided gui testing of android apps. In *Proceedings of the International Conference on Software Engineering Companion*, pages 689–691, 2016.

[45] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[46] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the International Conference on Automated Software Engineering*, pages 253–262, 2011.

[47] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158, 2007.

[48] Xuerui Wang, Andrew McCallum, and Xing Wei. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Proceedings of the International Conference on Data Mining*, pages 697–702, 2007.

[49] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ACM Sigplan Notices*, volume 45, pages 155–166, 2010.

[50] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In *Proceedings of the International Conference on Program Comprehension*, pages 48–59, 2015.

[51] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the International Conference on Software Engineering*, pages 620–631, 2015.

[52] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265, 2013.

[53] Tingting Yu, Tarannum S Zaman, and Chao Wang. Descry: reproducing system-level concurrency failures. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 694–704, 2017.

[54] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the International Conference on Software Engineering*, pages 102–112, 2012.

[55] Hrushikesh Zadgaonkar. *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.

[56] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems*, pages 321–334, 2010.

[57] Yuanyuan Zhang, Mark Harman, Yue Jia, and Federica Sarro. Inferring test models from kate's bug reports using multi-objective search. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 301–307, 2015.