2020

Automated Testing and Bug Reproduction of Android Apps

Yu Zhao
University of Kentucky, lunarlightgg@gmail.com
Author ORCID Identifier:
https://orcid.org/0000-0002-4417-7655
Digital Object Identifier: https://doi.org/10.13023/etd.2020.454

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Yu Zhao, Student

Dr. Tingting Yu, Major Professor

Dr. Zongming Fei, Director of Graduate Studies

Automated Testing and Bug Reproduction of Android Apps

DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for
the degree of Doctor of Philosophy
in the College of Engineering at the
University of Kentucky

By
Yu Zhao
Lexington, Kentucky

Director: Dr. Tingting Yu, Professor of Computer Science
Lexington, Kentucky 2020

https://orcid.org/0000-0002-4417-7655

ABSTRACT OF DISSERTATION

Automated Testing and Bug Reproduction of Android Apps

The large demand of mobile devices creates significant concerns about the quality of mobile applications (apps). The corresponding increase in app complexity has made app testing and maintenance activities more challenging. During app development phase, developers need to test the app in order to guarantee its quality before releasing it to the market. During the deployment phase, developers heavily rely on bug reports to reproduce failures reported by users. Because of the rapid releasing cycle of apps and limited human resources, it is difficult for developers to manually construct test cases for testing the apps or diagnose failures from a large number of bug reports. However, existing automated test case generation techniques are ineffective in exploring most effective events that can quickly improve code coverage and fault detection capability. In addition, none of existing techniques can reproduce failures directly from bug reports.

This dissertation provides a framework that employs artifact intelligence (AI) techniques to improve testing and debugging of mobile apps. Specifically, the testing approach employs a Q-network that learns a behavior model from a set of existing apps and the learned model can be used to explore and generate tests for new apps. The framework is able to capture the fine-grained details of GUI events (e.g., visiting times of events, text on the widgets) and use them as features that are fed into a deep neural network, which acts as the agent to guide the app exploration. The debugging approach focuses on automatically reproducing crashes from bug reports for mobile apps. The approach 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.

KEYWORDS: mobile app testing, bug reports, debugging, machine learning, natural language processing

Author's signature: Yu Zhao

Date: December 2, 2020

Automated Testing and Bug Reproduction of Android Apps

By
Yu Zhao

Director of Dissertation: Tingting Yu

Director of Graduate Studies: Zongming Fei

Date: December 2, 2020

I dedicate my dissertation work to my parents, my loving spouse, and my child.

ACKNOWLEDGMENTS

I would like to say thanks to the people who accompanied me in my Ph.D. study. Thanks to their assistance, help, guidance, support, and friendship. First and foremost, I would like to thank my advisors Dr. Tingting Yu and Dr. Qian Chen. Dr. Tingting Yu has been advising me since the second semester of my Ph.D. study. I have appreciated she giving me a chance to join her research group when I urgent need to join a new research team because my first advisor left the University of Kentucky. I can not imagine how I would have survived through my Ph.D. study without this opportunity and financial support that she provided to me. Dr. Tingting Yu guides me to work on the software testing research area start from scratch. She is a patient, hard-working, and inspiring mentor. She also would like to do coding by herself and share the experience with those who are in her research group. I get a big influence on her hard work and passion for the research. She teaches me how to do a good research from thinking about an idea to presenting a paper. She always encourages me to pursue top-notch research. I would like to thank her for everything she did for me. Specifically, I feel so grateful that she provides valuable advice all the time. I can not achieve my current accomplishment without her lead. Dr. Qian Chen had been advising me in the first semester of my Ph.D. study. I appreciated that he encouraged me to pursue a Ph.D. degree when I had worked in the industry for about 4 years. At that time, he gave me some encouragements and suggestions to pursue my Ph.D. study on my research presentation. It is no exaggeration to say Dr. Qian Chen changed my life. Without him, I might not have enough courage to continue my research career. He had taught me many things in the computer network area. I always inspire myself with his words "the most important thing for a Ph.D. student is hard work". His wealth of knowledge and insight in research also encourages me

to keep learning new things. He always helped and encouraged me to achieve my full potential. I am extremely grateful to have Dr. Tingting Yu and Dr. Qian Chen as my advisors in my Ph.D. study. I want to repay them by achieving higher research accomplishments in the future.

In my research career, I also want to thank Dr. Yunhuai Liu who was the advisor and director of me when I was doing my internship and working in the industry for about 5 years. It was he to find me from crowds and guide me to work on the computer science research career. He kept reminding me not to give up my research career. He is also an open-minded and ambitious researcher. I am grateful for his valuable advice and never forget every working late at night with him before the paper submission deadline. I want to say thanks for his countless hours to guide me in my research. I consider myself to be fortunate to have his guidance and mentor in my life.

I am grateful to Dr. Jane Hayes, Dr. Zongming Fei, and Yuan Liao as my committee members. They have provided valuable feedback and suggestions through my Ph.D. journey. I thank the Director of Graduate Studies in the computer science department Dr. Zongming Fei and Dr. Mirosław Truszczyński who helped me through my whole Ph.D. study time. I am very honorable and very lucky to be a teaching assistant for both of them in my Ph.D. study. I have learned a lot of teaching skills from their wealth of teaching experience.

Among all members of Dr. Tingting Yu's research group and Dr. Qian Chen's research group, they are all very nice people. Specifically, I want to thank Dr. Xin Li and Dr. Huazhe Wang. They helped me to pass through a tough time when I was new to the US. Thanks for Justin. He was glad to share his 10 years of industry java coding experience. Without his help, I might need to spend much more time to learn java coding. Thanks to Dr. Zhouyang Jia, his valuable knowledge in software engineer research helped a lot in my study.

I am grateful to my language partner in Lexington, Ky, Alex, and Victor. I cannot write enough about their love, kindness, and support during my years at the University of Kentucky. They not only helped me to practice my English speaking but also to be my family members. They helped me a lot in my daily life. They gave me many valuable suggestions when I met problems in my study and life. They kept encouraging me to pass through my tough time and celebrating every achievement.

Lastly, I would like to thank my family, Mom, Dad, my loving spouse - Yuanyuan Wu, and my little son Jack for their continuous love and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Mobile applications (apps) have become extremely popular – in 2017 there were over 2.2 million apps in Google Play's app store [3]. 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 80% of app users would abandon an app if they were to repeatedly encounter a functionality issue [4]. This motivates developers to rapidly identify and resolve issues, or risk losing users.

To guarantee the quality of mobile apps, developers often generate test cases to detect and resolve bugs before the app is released to the market. Because of the rapid releasing cycle of apps and limited human resources, it is difficult for developers to manually construct test cases. Therefore, different automated mobile app testing techniques have been developed and applied [5, 6, 7, 8, 9].

The test cases for mobile apps are often represented by sequences of Graphical User Interface (GUI) events (e.g., click, scroll, edit, swipe, etc) to mimic the interactions between users and apps. The goal of an automated test generator is generating such event sequences with the goal of achieving higher code coverage and/or detecting bugs. A successful test generator is able to exercise the correct GUI widget in the current app page, so that when exercising that widget, it can bring the app to a new page, leading to the exploration of new events. However, existing mobile app testing tools [5, 6, 7, 8, 9] often explore a limited set of events because they have limited capability of understanding which GUI events would expand the exploration like a human does. This can lead to automated test generators performing unnecessary actions that are unlikely to lead to new coverage. Therefore, we need an testing approach that can effectively exercise the important events to lead high coverage and fault detect capability.

While the goal of testing is to detect bugs, after an app is released, it may still contain bugs that were failed to be detected during testing phase and these bugs may be revealed at user's end. To track and expedite the process of resolving app issues, many modern software projects use bug-tracking systems (e.g., Bugzilla [10], Google Code Issue Tracker [11], 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 [13]. 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 [14]. 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.

Given the foregoing discussion, the overall goal of our research is to provide an automated framework, targeted at Android apps, that can effectively and efficiently detect bugs during testing phase and reproduce bugs from bug reports. Our essential idea is leveraging a variety of artificial intelligence (AI) methods that enable machine to understand the semantic meaning of GUI widgets and the textual description of bug reports, so that it can explore important GUI widgets during testing and reproduce bugs directly from bug reports written by natural language.

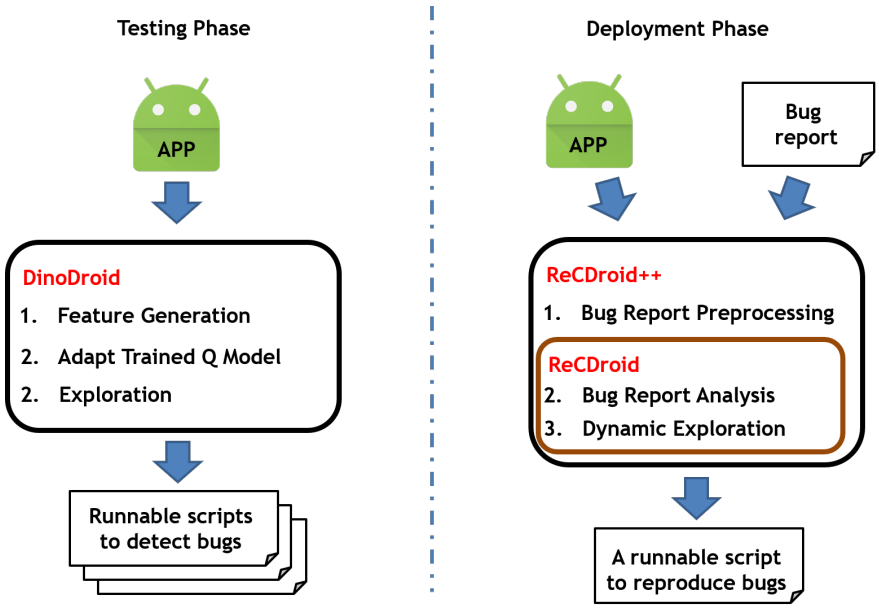


Figure 1.1: A Summary of My Research

1.1 Research Summary

This research focuses on handling Android app bugs in both testing and deployment phases. Fig. 1.1 shows an overview of this research.

We developed **DinoDroid**, an approach based on deep Q network to automate testing of Android apps. DinoDroid learns a behavior model from a set of existing apps and the learned model can be used to explore and generate tests for new apps. DinoDroid is able to capture the fine-grained details of GUI events (e.g., visiting times of events, text on the widgets) and use them as features that are fed into a deep neural network, which acts as the agent to guide the app exploration. DinoDroid

automatically adapts the learned model during the exploration without the need of any modeling strategies or heuristics. We conduct experiments on 64 open-source Android apps. The results showed DinoDroid outperforms existing Android testing tools in terms of code coverage and bug detection.

We developed **ReCDroid**, a novel approach that can automatically reproduce crashes from bug reports for Android apps. ReCDroid uses natural language processing (NLP) to analyze the steps text of the bug report and automatically identify GUI components and related information (e.g., click events) that are necessary to reproduce the crashes. Then a dynamic GUI exploration is designed to synthesize event sequences with the goal of reproducing the reported crash. 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.

ReCDroid+ is an extended version of ReCDroid. Besides all of the components in ReCDroid, ReCDroid+ includes a new bug report preprocessing component, which can automatically extract information that be directly processed by ReCDroid. Therefore, ReCDroid+ is an end-to-end bug reproduction tool, meaning that no human effort is needed during the process of reproducing bug reports. Specifically ReCDroid+ leverages a combination of HTML parser [15], convolutional neural networks (CNN) [16], and long-short term memory (LSTM) [17] to extract the step-to-reproduce (S2R) sentences from the bug report text. These S2R sentences are sent to ReCDroid as input. 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 bug reports.

1.2 Contributions

In summary, our research makes the following contributions:

- We present a learning-based approach, called DinoDroid, to automatically test android app. DinoDroid involves a novel deep Q-learning model that can process complex features at a fine-grained level. An empirical study showing that the approach is effective in achieving higher code coverage and better bug detection than the state-of-the-art tools. The implementation of the approach as a publicly available tool, DinoDroid, along with all experiment data [18]
- We design and development of a novel approach ReCDroid 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 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) [19].
- We introduce an extension version of ReCDroid as ReCDroid+ which can automatically reproduce the raw HTML bug report downloaded from bug tracking systems. A novel text analysis technique that uses natural language processing (NLP) and deep learning to derive a set of heuristics that can automatically

capture steps sentence relevant to the bug report. An empirical study showing that ReCDroid+ is effective at reproducing Android crashes and likely to improve the productivity of raw HTML bug report.

Chapter 2

Background

In this chapter, we first describe background related to Android framework, mobile app testing and bug report reproduction, natural language processing, and Q-Learning. We then discuss related work.

2.1 Natural Language Processing

Natural Language Processing (NLP) is a branch of Artificial Intelligence (AI). It studies the interaction between human and machine using natural language. In this dissertation, we utilize NLP to deal with how machine understands human language on app content and app-related document. We introduce several NLP techniques that are employed in our work.

Word Embedding is a feature learning technique. Many machine learning algorithms and all deep learning methods can not process the texts as their raw format. They need real numbers as the input to handle NLP tasks as text classification. So word embedding is used to transfer a word from a vocabulary into a vector with real numbers. By leveraging corpora of unlabeled text, the word embedding can be computed for capturing both syntactic and semantic information about words [20]. It is used as the underlying input representation and has been shown to be a common part of NLP tasks. In this dissertation, two different Word Embedding methods are used to map a word to a digit vector.

The count-based word embedding method maps all words in a sentence to vector with counting. Consider a corpus C of all sentences to be handled. A dictionary is built by extracting each unique word from the C. To transfer a sentence into a vector, count-based word embedding method counts the frequency of each word in the target sentence and fills it to the matched position of the dictionary. After that, an embedding vector is generated in which each item represents the frequency of the word in the target sentence. A simple example is used to understand the word embedding method. Considering, there are 3 sentences "I click A", "I click B", and "I click A and click B". Count based word embedding method puts every word from sentences into a dictionary. The dictionary is ["I", "click", "A" ,"and", "B"]. Every word in a sentence will be counted frequency and match the word vector to generate

an embedding vector. "I click A" is embedded to [1 1 1 0 0]. "I click B" is embedded to [1 1 0 0 1]. "I click A and click B" is embedded to [1 2 1 1 1].

Prediction based approaches for word embedding seems to have better performance than the count based word embedding across a variety of NLP domains as text classification and name entity recognition [21]. These methods generate lower dimensions of the embedding vector to better dense word representation. The popular Word2vec [22] is one of these prediction based approaches. Word2Vec exploits a neural network model to learn word associations from a large corpus of text. The word vectors are randomly initialized and trained to predict the current word from a window of surrounding context words. After the train, the vectors indicate the semantic similarity between the words represented by those vectors.

N-gram technique [23] uses n-word rather than 1-word to represent the words' features. It is used to keep order information or keep word dependence information as the words' features. It is very common to use N-gram with count-based word embedding method. For example, The dictionary ["I", "click", "A" ,"and", "B"] as the 1-gram can be transferred to 2-gram ["I click", "click A", "click B", ...] or 3-gram ["I click A", "click A and", ...] in before example.

Text Classification is one task of NLP technologies. Text classifiers automatically analyze text then assign a set of categories based on the content of text [24]. It can be used with broad applications as spam detection, sentiment analysis, and topic labeling. In this dissertation, we utilize text classification to identify S2R sentences and crash sentences in a bug report.

The text classification method inputs the feature of text and uses a classifier to assign the categories. There are multiple types of classifier design. It is important to find a good classifier for the text classification pipeline [24]. Logistic regression is one of the simplest classification algorithm [25]. It has been widely addressed in data mining domains. In the information retrieval tasks, the Naive Bayes Classifier (NBC) [26] is very popular which is a computationally inexpensive and memory friendly method. Support Vector Machine (SVM) [27] is a popular technique that employs a non-probabilistic binary linear classifier. This technique can also be used in all domains of data mining such as text classification, bioinformatics, image, and video, etc. Comparing to previous machine learning, deep learning approaches have achieved surpassing results [24] on tasks as image, NLP, face recognition, etc. CNN and LSTM as the deep neural network model won several international pattern recognition competitions and set numerous benchmark records on large and complex data sets [28]. Deep learning algorithms have the capacity to model complex and non-linear relationships within data [29]. It is the reason why it is successful on broadly large data set classifications.

Dependency parsing is the NLP task to extract a dependency parse of a sentence [30]. It represents the sentence's grammatical structure and uses head and child to represent the dependency syntactic relation. The syntactic relations form a tree structure. Every one word has one head in the dependency parse. The syntactic relation includes subject, predicate, object, adjectival modifier and so on. As shown in Fig. 2.1, the word "puppy"'s adjectival modifier is "black" in the tree. Dependency parsing is one of the mainstream research areas in NLP [31]. Depen-

dency representations are useful for a broad applications of NLP tasks, for example, machine translation [32], information extraction [33], and parser stacking [34].

Part-of-speech tagging is the process of labeling each word in a sentence which indicates the status of that word according to their morphological and/or syntactic properties [35]. A part-of-speech is a grammatical category, including verbs, nouns, adjectives, adverbs, determiner, and so on. As shown in Fig. 2.1, the word "black" it labeled as "ADJ" and "puppy" is labeled as "NOUN". Part-of-speech tagging has a crucial role in fields of natural language processing (NLP) including machine translation.

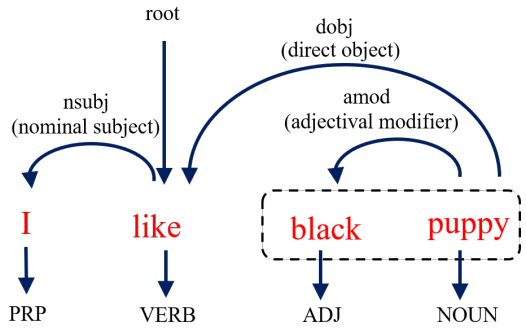


Figure 2.1: Dependency parsing and Pos Tagging

2.2 Reinforcement Learning

We introduce Q-learning and deep Q-learning.

2.2.1 Q-learning

Q learning [36] is a model-free reinforcement learning method that seeks to learn policies that can maximize a numerical reward for agents interacting with an unknown environment. Q-learning is based on trial-and-error learning in which an agent interacts with its environment and assigns utility estimates known as Q-values to each state.

The Q learning can be formalized as a Markov Decision Processes (MDPs) which can be described by a 4-tuple (S, A,P, R). S represents the set of states and A represents the set of actions. As shown in Fig. 2.2 the agent iteratively interacts with the outside environment. At each time step t, an agent interacting with the MDP observes a state $s_t \in S$ and selects an action $a_t \in A$. and executes it on the outside environment. After exercising the action, there is a new state $s_{t+1} \in S$, which can be observed by the agent. In the meantime, an immediate reward $r_t \in R$ is received.

Q learning exploits Q function to estimate how good to select an action in a current state. An expected cumulative reward of a sequence of actions can be returned by executing a sequence of actions that starts with an action a_t from a state st and then follows the policy π. The optimal policy Q^{π} is the maximum expected cumulative

reward which is achievable for a given (state, action) pair.

$$Q^*(s_t, a_t) = max \sum_{t>0} (\gamma^t r_t | s = s_t, a = a_t, \pi) \tag{2.1}$$

At each step s_t, with the Bellman equation in dynamic programming [37], the optimal strategy for the equation 2.1 is to select the action which maximizes the sum: $r + Q^*(s_{t+1}; a_{t+1})$ where r is the immediate reward of the current step s_t.

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{2.2}$$

The Q learning algorithm uses equation 2.2 to value of Q^* on each state-action pair iteratively. If the states and actions are discrete and finite, this pair can be represented in a tabular. In the tabular, the row and column represents the state and action where all pairs are initialized with default value. As shown in Fig. 2.2, every time the agent executes an action a_t to reach state s_{t+1} and gets a reward r_{t+1}, the relating state-action value is updated as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.3}$$

In this equation, α is a learning rate between 0 and 1, γ is a discount factor between 0 and 1, s_t is the state at time t, and a_t is the action taken at time t. In this equation, the value of subsequent state-action pair will influence the value of preceding pairs. This estimator can converge to the true value if the environment is sufficiently explored [38]. Whenever Q-learning is learned to estimate the values precisely, these Q-values can be used to determine optimal behavior in each state by selecting action $a_t = \arg\max_a Q(s_t, a)$.

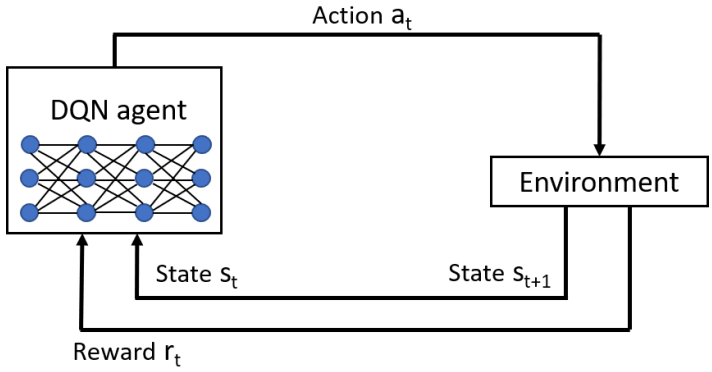


Figure 2.2: Deep Q Network

2.2.2 Deep Q learning

Deep Q network (DQN) is used to scale the classic Q-learning to more complex state and action spaces [39, 40]. For the classical Q-learning, $Q(s_t, a_t)$ are stored and

visited in a Q table. It can only handle the fully observed, low-dimensional state and action space. As shown in Fig. 2.2, A DQN is a multi-layered neural network that for a given state s_t outputs a vector of action values $Q(s_t, a)$. Because neural network can input and outputs high-dimensional state and action space, DQN can scale more complex state and action spaces. A neural network can also generalize Q-values to unseen states, which is not possible when using a Q-table. It utilizes the follow loss function [40] to alter the network to minimize the loss function:

$$L(w) = (r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2. \tag{2.4}$$

Following is Q-Learning gradient:

$$(r_t + \gamma max Q(s_{t+1}, a, w) - Q(s_t, a_t, w)) \frac{\mathrm{d}Q(s_t, a_t, w)}{\mathrm{d}w} \tag{2.5}$$

To minimize the loss function L(w), an optimal w as the multi-layered neural network's weight is computed by the stochastic gradient descent method. The Q neural network converges toward the optimal Q-function Q^* with minimum loss function L(w). As we can see in the equation 2.4, with the input of (s_t, a_t), the neural network is trained to predict the Q value as:

$$Q(s_t, a_t) = r_t + \gamma * \max_a Q(s_{t+1}, a) \tag{2.6}$$

So in a training sample on a current state s_t, the input is s_t and a_t and output of the multi-layered neural network is the corresponding Q value which can be computed by $r_t + \gamma * \max_a Q(s_{t+1}, a)$.

2.3 Reproducing Bug Reports

When bug or unexpected behavior of software is observed by a user, a bug report might be written and submitted by user to report the problems using issue trackers [41]. After the developer receives the bug report in the issue tracker, the first step of developer to fix bug is reproducing the bug.

The bug tracking systems keeps track of reported software bugs in software development projects. There are some public accessing bug tracking systems e.g. Github, Google Code Archive, and BitBucket. As shown in Fig. 2.3, there are different bugs are reported to the LibreNews-Android app in Github.

Bug reports are documents that describe the essential information about the found software bugs. In one bug report there are mainly 3 parts of necessary information: Observed Behavior (OB), Steps to Reproduce (S2R), and Expected Behavior (EB) [42]. Developer can use this information to reproduce the bug. For example, in a bug tracking system, each app may have multiple issues as bug reports. For example, as we click the last issue in Fig. 2.3, a bug report will come out as shown in Fig. 2.4. In this bug report, these three parts of information are clear and structured by tags. In the bug report, OB is a crash. EB is no crash. S2R is "1. Click on the refresh rate option", "2. Set 12hours", "3. Exit the app", and "4. Change the time

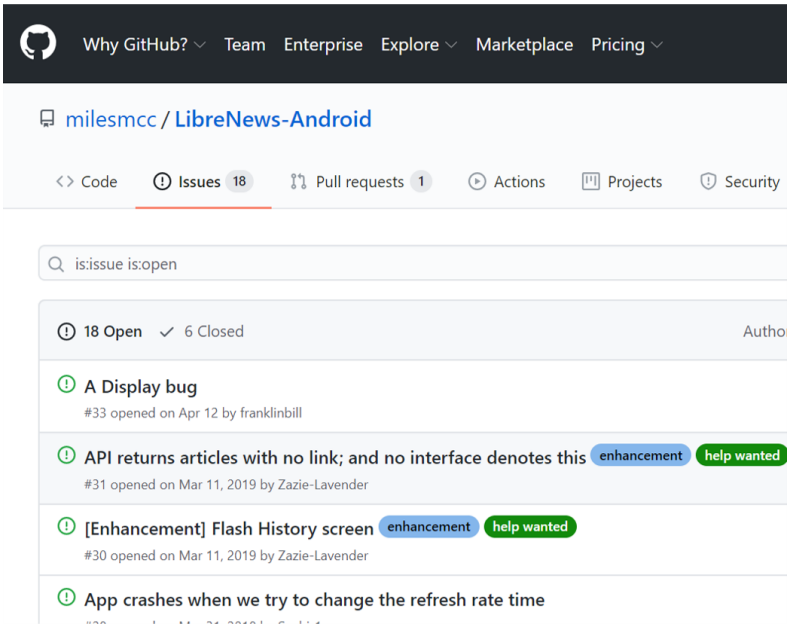


Figure 2.3: LibreNews-Android Issues on Github [1]

option". However, some bug reports may not be structured. For example, in Fig. 2.5, these 3 information require developer to extract manually with understanding of the meaning of sentences.

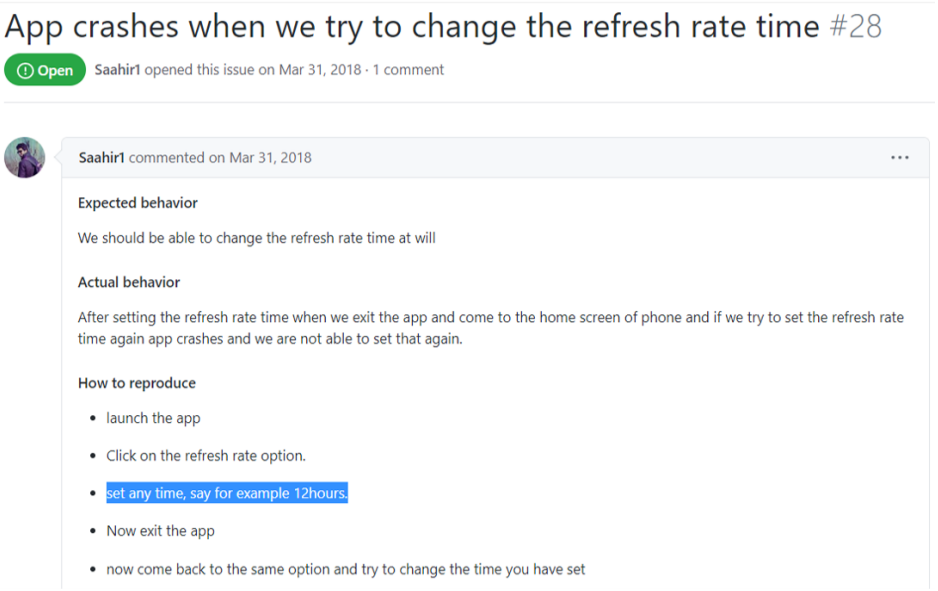


Figure 2.4: Structured LibreNews-Android's bug report

Some bug reports may miss one or more necessary parts. DeMIBuD [42] manually analysis 3000 bug reports. They find 93.5%, 35.2%, and 51.4% bug reports explicitly describe OB, EB, and S2R, respectively.

Figure 2.5: Unstructured LibreNews-Android's bug report

Software developers attempt to reproduce software bug report and fix the bug report. However, some of bug reports are hard to reproduce. An existing survey work [43] analysis the reason of non-Reproducible bugs. They identify 11 key factors to explain the non-reproducibility of software bugs. In the survey result, 92% of the invited participants consider missing information it the major cause of non-reproducibility bug. About 85% of participants consider it is hard to reproduce duplicate bugs, performance bugs, memory misuse related bugs, and third-party defects. Positive bug reports, bug intermittency, ambiguous software specifications, and restricted security access are also considered to be the reason for non-reproducibility by 75% of participants. Finally, touch and gesture related bugs also are agreed by 67% of participants to be difficult to reproduce.

Erfani et al. [44] conducts a study on analyzing the characteristics of Non-reproducible Bug Reports. In this work, the authors mine an industrial and five open-source bug repositories. They extract 32124 non-reproducible bugs from 188319 bug reports in total. They have some interesting results. Only 17% of bug reports are Non-reproducible Bug Reports. The Non-reproducible Bug reports can be classified into 6 categories as "Interbug Dependencies" 45%, "Environmental Differences" 24%, "Insufficient Information" 14%, "Conflicting Expectations" 12% and "Non-deterministic Behaviour" 3%.

2.4 Android Framework

This subsection provides background on the Android platform which helps to understand the android testing chapters.

Android is an open-source operating system based on a modified version of Linux Kernel. It is designed for touchscreen mobile devices which include smartphones and tablets. Android app is developed in Java. The android code is compiled to standard Java bytecode, then is converted to customized Dalvik bytecode format and packed to APK file. Users can download APK from app market and install and use it on the android device.

Android apps require an AndroidManifest.xml file to be built [45]. This file involves the essential information about the app to the Android build tools, the Android

operating system, and Google Play. In this file, many vital contents for managing the lifecycle of an application are declared as app's package name, components of the app, permissions of app, and android versions of the app.

The Android framework defines four types of components [46]: Activities, Services, Broadcast Receivers, and Content Providers. These components are supported by the Android Software Development Kit (SDK). Developers can extend the super classes (components) provided by SDK and implement their own function based on app development requirements.

An **Activity** provides a screen, Graphical User Interface (GUI) that users can interact with to perform actions. A set of layouts can be contained in an activity e.g. LinearLayout, TextView, and EditText. The layouts contain GUI controls which are known as view widgets, for example, TextView for viewing text and EditText for text inputs. The layouts and their controls are typically described in a configuration XML file with each layout and control having a unique identifier. As shown in Fig. 2.6, the developer can configure and inject widgets on the layout of activity based on the requirement. Every widget can be implemented along with an action such as moving to another page and jumping out a dialog on current activity. Every layouts and widgets are configured in an XML file with a unique identifier and perform an action.

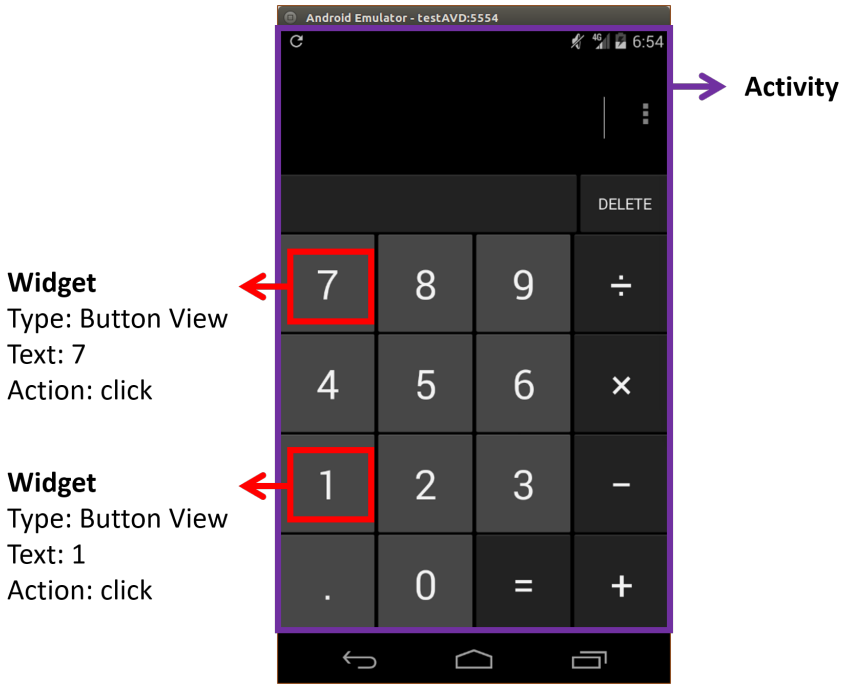


Figure 2.6: Activity, widget, and action example

Service does not have any user interact on the screen. It is usually used to perform long-term running tasks, such as playing music or triggering alarm clocks [47]. Including Activities and Broadcast Receivers, some other components can start a service. Once started, a service can run for some time, even after user mores to activity or another app.

Broadcast receiver subscribes to Intents broadcasts from other applications and the system. Android apps can send or receive broadcast messages similar to the publish-subscribe design pattern [48]. Android system and other Android apps can send and receive these messages.

Content provider manages and shares application data to other apps. It enables users to access data of other applications These data are stored in the files or database, including contract information, message, book, photo, video, and music.

An intent is an abstract description of an operation of app to be performed. Activities, Services, and Broadcast Receivers are activated via Intent messages. For example, to visit a new activity from the current activity, developer can build an intent with "Intent intent = new Intent(this, newActivity.class)" and send it with "startActivity(intent)".

2.4.1 Lifecycle of Android Activity

The activity instances in app transition through different states in its lifecycle [2]. As we can see in Fig. 2.7, the operation of an activity in a lifecycle can be onCreate(), onStart(), onResume(), onPause(), onStop(), onRestart(), and onDestroy(). The widgets focused by our research are depended by the related activity which are effected by these different states of activity.

2.4.2 Android Virtual Devices

Android Virtual Devices (AVD) (a.k.a emulators) is provided to developer to simulate and imitate real devices [49]. Without needing to have each physical device, Developer can test app on a variety of devices and Android API levels. Emulator provides most of the capabilities of a real device. User can simulate the network loading, incoming call, short message, and camera on the AVD. It also supports the hardware sensor e.g. rotation. As shown in Fig. 2.8, AVD can install a set of application like browser, Camera, Email. It also supports system level event as "power", "back" and "home".

In this research, android emulators are used extensively to install and test apps. The android emulator can be configured on an x86 architecture host machine. The Android Debug Bridge (ADB) [50] tool is a versatile command-line tool that can be used to communicate an android emulator or a real android device with a host machine.

2.5 Mobile App Testing

Software testing [51] is the process of executing test cases on the program to detect differences between the actual output and the expected output. There has been a great deal of work on software testing, including white-box testing, black-box testing, and grey-box testing [52]. Test case generation can generate test cases automatically. The measurement for the performance of the generation involves code coverage, branch coverage, bug trigger possibility, and so on.

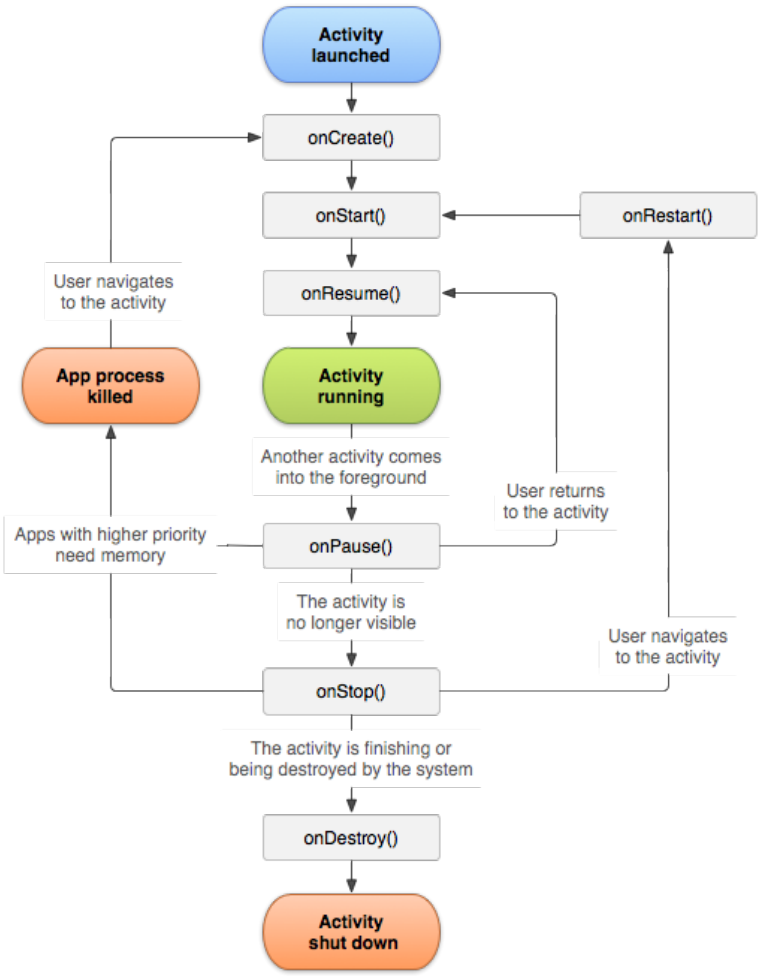


Figure 2.7: Android Activity Lifecycle [2]

Compared to traditional software, mobile devices have distinguished characteristics [53], such as mobile connectivity, limited resources, limited energy, new user interfaces, and context awareness. Android apps are event-driven. The app is running in the idle state until a new user GUI interaction comes in. In Android, GUI events include user interaction as clicks, inputs, scrolls, swipes, or system events such as new coming message [54]. The test cases for mobile apps are represented by sequences of GUI events to mimic the interactions between users and apps. As shown in Fig. 2.9, the android test case is [click: "settings", click: "minimum length", input: "4", click: "ok"]. The target of an automated test generator is generating such event sequences to achieve higher code coverage and detecting bugs.

2.6 Android Testing Tools

There are some GUI modeling frameworks that can provide APIs for performing GUI testing. These frameworks are low-level frameworks. The provided APIs do not support automated GUI testing directly. For example, the provided APIs only

Figure 2.8: Android Emulator

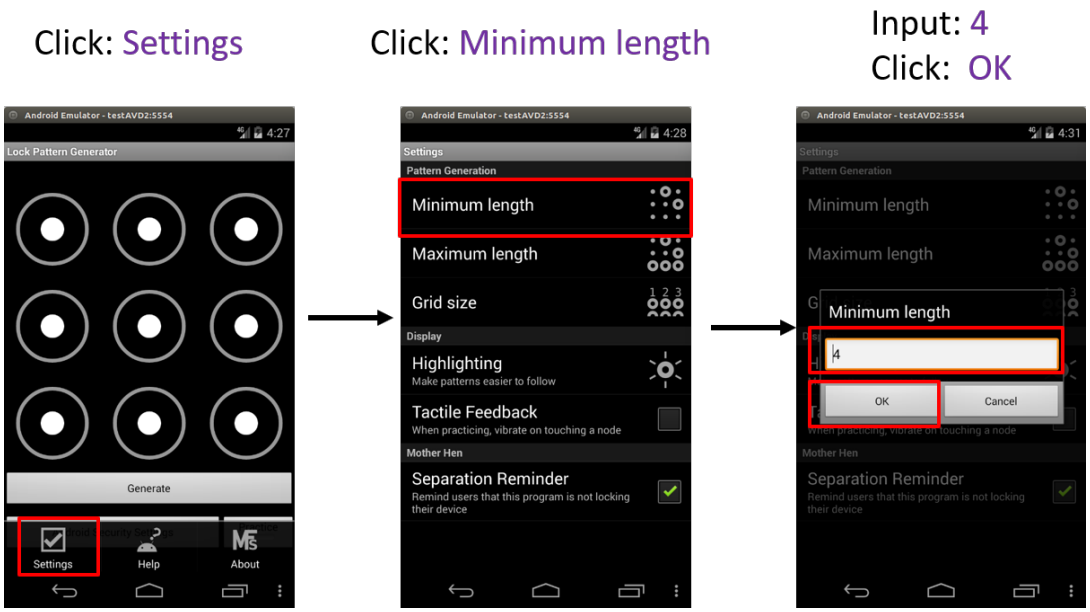


Figure 2.9: Mobile Test Case

support simulating the basic user actions, such as click, type, and rotate. As for which view (component) to click and to type has to be manually determined by developers [9].

User needs to write a script with detailed descriptions of the actions based on the provided APIs. Then the GUI testing tool can test the applications using the script. GUI testing tools for mobile applications are very useful as most of the automated

mobile testing tools are implemented on top of them, such as GUI ripper [7] and Robotium [55].

2.6.1 UI Automator

The UI Automator testing framework [56] is a free Google official tool to test the user interface (UI) of Android applications. It can support UI black-box testing. The UI Automator supports "Change the device rotation", "Press a key or D-pad button", "Press the Back, Home, or Menu buttons", "Open the notification shade", "Take a screenshot of the current window" actions in Android UI testing [56]. The provided APIs are very simple and clear. For example, the UiDevice API supports actions on device buttons. The UiDevice.pressHome() can click the Home button in the Android application. The UiObject API can be instantiated as a view (component) of the current page. The UiObject.click() can click on that view.

The attached view tool UIautomatorViewer [57] is an important tool for helping developers find the view (component) information on the current page. The information includes the view dependency tree, view id, view type, view text, view click type, and so on. A developer can open UIautomatorViewer to find interesting views with details. Then the developer can write a script to use UI Automator to test the application. For example, a script can be written by UiObject(button1).click(), sleep(), UiObject(button2).click(). The actions of button1 and button2 can be extracted from the UIautomatorViewer. UIautomatorViewer and UIa Automator can work separately. People can also use UIautomatorViewer to analyze an Android application written by other people.

UI Automator has several advantages comparing to other Android GUI testing tools. The first advantage is that UI Automator is developed by Google, which developed Android. So the reliability and robustness of UI Automator are convincingly good. The second advantage is that UI Automator supports multi-application GUI testing. For example, if a user wants to switch from application A to an application B and switch back, UI Automator supports these actions. In the meantime, UI Automator has certain drawbacks. First, UI Automator cannot obtain the class name of activities (page) [58]. In the official API documentation, the API for implementing this function "getcurrentactivityname" is unreliable. However, activity name is very important in GUI testing because it indicates the node name in a GUI event flow graph. Also, UI Automator does not support web application.

2.6.2 Robotium

Robotium [59] is similar to UI Automator [56]. Robotium is a free black-box Android testing tool. It is not developed by Google development team but is an open-source tool [59]. Robotium requires developers to explicitly write actions in java programs. The basic platform is the JUnit framework. When testing using Robotium, Junit will output results either pass or fail. Robotium supports a variety of actions, including click, rotate, and type actions, similar to UIautomator. In addition, Robotium can find views (components) by the view IDs and support actions on hybird (web) Android

applications. The provided APIs in Robitium is clear and simple. For example, the view v=solo.getview(ID) can choose a view on an arbitrary page. The v.click() can click on this view.

Robotium has several advantages. For example, Robotium can provide the names of activities on the current page of the application. The activity names are very important for GUI testing.In addition, Robotium is an open source tool enabling free extensions. One disadvantage of Robotium is that it does not support testing multi-applications. It can not perform actions across multiple applications.

2.6.3 Other Tools

Monkeyrunner [60] is a Google official Android testing tool. It is a python-based testing framework, which provides python APIs. It is convenient for developers who are familiar with python. Monkeyrunner provides actions such as press button and type. It can also take the screenshot of the current page. Monkeyrunner does not rely on any testing platforms. The python APIs provided by Monkeyrunner can control any applications running on the device using the commands from terminal. Whereas the UI Automator depends on the Android Studio testing framework and Robotium depends on the JUnit testing framework.

Appium [61] is an open-source, cross-platform GUI testing framework. It provides a standard interface for multi-platform mobile device applications. For example, if using Appium to test an iOS application, Appium will call the UI Automator driver for iOS. If using Appium to test an Android device application, Appium will call the UiAutomator driver for Android. Appium also supports multiple languages, such as Java, Objective-C, JavaScript, PHP, Python, Ruby, C#, Clojure, and perl.

Espresso [62] can be used to write concise, reliable Android UI test cases. It is also a Google official GUI testing tool. The core APIs of Espresso are small, so it is easy to learn. In general, Espresso is similar to UI Automator [56].

Chapter 3

Related work

In this section, we discuss related work on mobile app testing. We also discuss existing work on analyzing bug reports to support software engineering activities.

3.1 Mobile App Testing

There have been a number of techniques on automated Android GUI testing.

Random testing [5] uses random strategies to generate events to test android applications. Because of its simplicity and availability, it can send thousands of events per second to the apps and can thus get high code coverage. However, the generated events may be largely redundant and noneffective. DynoDroid [6] improved random testing by exploring the app in a manner that can avoid testing redundant widgets. But it may still be ineffective in reaching functionalities involving deep levels of the app due to randomness.

Sapienz [63] uses multi-objective search-based testing to maximize coverage and fault revelation at the same time to optimize test sequences and minimize length. It extracts statically-defined string constants by reverse-engineering the APK. These extracted strings can be used as specific inputs for text fields. Sapienz makes use of a random crossover and mutation algorithm to optimize test sequences. It may also generate invalid sequences [38] and cost time on the iterative evaluation of new generated test.

Model-based methods [7, 8, 9, 64, 65, 66, 67, 68] build and use a GUI model of the app to generate test input. The models are usually represented as finite state machines or that store the transitions between app window states. For example, Stoat [69] utilizes a stochastic Finite State Machine model to describe the behavior of app under test. Unlike model-based testing, DinoDroid does not need to manually model the app behaviors. Instead, it can automatically learn app behaviors by deep Q network.

Systematic testing tools, such as symbolic execution [70, 71], aims to generate test cases to cover some code that is hard to reach. While symbolic execution may be able to exercise functionalities that are hard to exercise by other strategies, it is less scalable and not effective in code coverage and bug detection.

Machine learning techniques have also been used in testing Android apps. These techniques can be classified into two categories [38]. The techniques in the first category typically have an explicit training process to learn knowledge from existing apps then apply the learned experience on new apps [72, 73, 74, 75, 76]. For example, QBE [72] learns how to test android apps in a training set by a Q-learning. As discussed in Section 4.2.3, QBE is not able to capture fine-grained app behaviors due to the limitations of Q-learning.

In addition to QBE, Degott et al. [73] use learning to identify valid interactions for a GUI widget (e.g., whether a widget accepts interactions). It then uses this information to guide the exploration. Their following work [76] uses MBA reinforcement learning to guide the exploration based on the abstracted features: valid interactions and invalid interactions. However, like QBE, these techniques can not handle complex app features because of the high-level abstraction of state information.

Humanoid [74] employs deep learning to train a model from labeled human-generated interaction traces and use the model together with a set of heuristic rules to guide the exploration of new apps. For example, since the result of deep learning is only used to selected one event from the unexplored events in the current page. If all events in the current page are explored, the rule asks to compute the shortest path in a graph to find a page with unexplored events.

The techniques in the second category do not have explicit training process [77, 78, 79, 80, 38]. Instead, they use Q-learning to guide the exploration of individual apps, where each app generates a unique behavior model. As discussed in Section 4.2.3, the model learned from one app usually cannot be applied to another app. Also, they share the same limitations with Q-learning, where complex features cannot be maintained in the Q-table. QDROID [78] designs a deep neural network agent to guide the exploration of Android apps. However, like QBE, the widgets are abstracted into several categories. The neural network is used to predict one of the four categories. QDROID then randomly selects a widget under the predicted category. However, the abstraction of the states and actions may cause the loss of information of individual widgets.

Wuji [81] combines reinforcement learning with evolutionary algorithms to test android games. It designs a unique state vector for each game used as the position of the player character or the health points. Therefore, Wuji can not transfer the learned knowledge to new apps.

3.2 Reproducing Bug Report

There has been some work on improving the quality of bug reports for Android apps [82, 13]. Specifically, FUSION [82] 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 [83] 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.

There has been considerable work on using NLP to summarize and classify bug reports [84, 85]. For example, Rastkar et al. [84] summarize bug reports automatically so that developers can perform their tasks by consulting shorter summaries instead of entire bug reports. Gegick et al. [85] use text mining to classify bug reports as either security- or non-security-related. Chaparro et al. [42] use several techniques to detect missing information from bug reports. PerfLearner [86] 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. [87] 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 [88, 89]. For example, PrefFinder [89] uses NLP and information retrieval (IR) to automatically find user preferences for correcting the configuration of a running system. DASE [88] 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 [90] is the only existing work that can automatically identify S2R sentences in bug reports. 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 [91]. 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.

There are tools for automatically reproducing in-field failures from various sources, including core dumps [92, 93], function call sequences [94], call stack [95], and runtime logs [96, 97]. 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 [7, 9, 98, 70, 6, 69, 99]. 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.

Chapter 4

DinoDroid: Testing Android Apps Using Deep Q Network

In this chapter, we propose an approach DinoDroid which is an automate testing tool which is based on deep Q network to test of Android apps. DinoDroid is able to capture the fine-grained details of GUI events (e.g., visiting times of events, text on the widgets) and use them as features that are fed into a deep neural network, which acts as the agent to guide the app exploration. DinoDroid automatically adapts the learned model during the exploration without the need of any modeling strategies or heuristics.

4.1 Overview

Many automated GUI testing for mobile apps have been proposed, such as random testing [5, 6] and model-based testing [7, 8, 9]. Random test generation (e.g., Monkey [5]) is popular in testing mobile apps because of its simplicity and availability. It generates tests by sending thousands of GUI events per second to the app. While random testing can sometimes be effective, it is difficult to explore hard-to-reach events for driving the app to new pages. Model-based testing can improve code coverage by employing specific strategies or heuristics to guide the exploration. For example, A3E [9] employs depth-first search (DFS) to explore the model of app based on event-flow across app pages. Stoat [69] utilizes a stochastic Finite State Machine model to describe the behavior of AUT. It then utilizes MCMC sampling to direct the mutation of the model.

However, model-based testing often relies on human-designed models and it is almost impossible to precisely model an app's behavior. Also, many techniques add heuristics to the model for improving testing. For example, Stoat designed rules to assign each event an execution weight in order to speed up exploration. However, heuristics are often derived from limited observations and may not generalize to a wide categories of apps.

The inherent limitation of the above techniques is that they do not *automatically* understand GUI layout and the content of the GUI elements, so it is difficult for them to exercise the most effective events that can bring the app into new states.

Recently, machine learning techniques have been proposed to perform GUI testing in mobile apps [72, 38, 74, 76]. For example, Humanoid [74] uses deep learning to learn from human-generated interaction traces and uses the learned model to guide test generation as a human tester. However, they rely on human-generated datasets (i.e., interaction traces) to train the model and need to combine with heuristics to guide the testing.

Reinforcement learning (RL) can teach machine to decide which events to explore rather than relying on pre-defined models or human-made strategies [100]. Specifically, a Q-table is used to record the reward of each event and the information of previous testing knowledge. The reward function can be computed based on the differences between pages [38] or the unique activities [72]. Reinforcement learners will learn utility values that the agent uses to maximize cumulative reward with the goal of achieving higher code coverage or detecting more bugs.

While existing RL techniques have improved app testing, they focus on abstracting the information of app pages and use the abstracted features to train behavior models for testing [73, 78]. For example, QBE [73], a Q-learning-based Android testing tool, abstracts each app page into five states according to the number of widgets (e.g., too-few, few, moderate, many, too-many). However, these techniques do not understand the fine-grained information of the page like human testers normally do during testing, such as the visiting frequency of GUI widgets, the page layout, and the content of widgets. Therefore, the learned model may not capture the accurate behaviors of the app. Also, many RL-based techniques focus on training each app independently [77, 78, 79, 80, 38] and thus cannot transfer the model learned from one app to another.

To address the aforementioned challenges, we propose a novel approach, DinoDroid, based on deep Q network (DQN). Specifically, DinoDroid learns a behavior model from a set of existing apps and the learned model can be used to explore and generate tests for new apps. During the training process, DinoDroid is able to understand and learn the details of app events by leveraging a deep neural network model. More precisely, DinoDroid takes in a set of features, such as widget visiting frequency and widget content. The insight of these features represents what a human tester would do during the exploration. For example, a human tester may decide which widget to execute based on its textual content or how many times it has been visited. DinoDroid does not use any heuristics to tune the parameters of these features, but let the DQN agent learn a behavior model based on the feature values (represented by vectors) automatically obtained during training and testing phases.

A key novel component of DinoDroid is a deep neural network (DNN) model that can process multiple complex features to predict Q-value for each GUI event to guide Q-learning. With the DNN, DinoDroid can be easily extended to take other types features. Specifically, to test an app, DinoDroid first trains a set of existing apps to learn a behavior model. The DNN serves as an agent to compute the Q-values used to decide which event to trigger at each iteration. In the meantime, DinoDroid maintains a special event flow graph (EFG) to record and update the feature vectors, which are used for DNN to compute Q-values. Given a new app under test, the learned model is used as an initial model and the agent continuously adapts the model to the new

app by generating new actions to cover the code missed by the existing model. By applying the initial model, it allows limiting testing time and increasing code coverage for the new app in a short time.

The remainder of this research is organized as follows. We present motivation and background in Section 4.2. Section 6.3 describes the approach of DinoDroid, followed by the evaluation 6.7. We then discuss the limitations in Section 6.8, and finally conclude our work in Section 6.9.

4.2 Motivation and Background

In this section, we first describe a motivating example of DinoDroid, followed by the background of the problem formulation and the discussion of existing work.

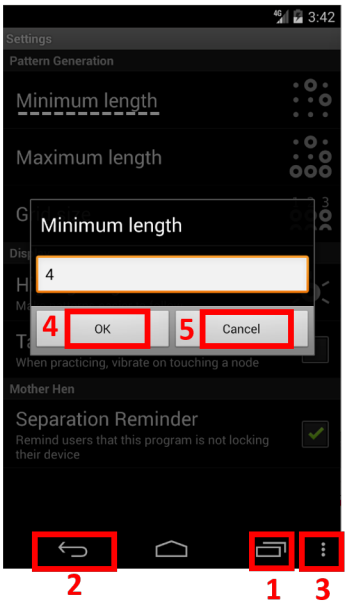
4.2.1 A Motivating Example.

Fig. 5.5 shows an example of the app *lockpatterngenerator* [101]. After clicking "Minimum length", a message box popped up with a textfield and two clickable buttons. Therefore, the current page has a total five clickable widgets (i.e., "restart", "back", "menu", "OK", and "Cancel") specific to the app. The home buttons is not considered because it is irrelevant to the app. When a human tester encounters this page, he/she needs to decide which widget to exercise based on his/her prior experience. For example, the tester is likely to exercise the widgets that have never been visited before.

In this example, suppose none of the five widgets in the current page have been visited before, intuitively, the tester tends to to select "OK" because when clicking on "OK", it is more likely to bring the app to a new page. "Cancel" is very possible be the next widget to consider because "restart", "back", and "menu" are system events, the results of clicking them are likely to be expected by the tester based on his/her past experience of testing a number of other apps. To decide if a widget has a higher priority to be exercised, the tester may need to consider its "features", such as how many times it has been visited, what the next pages are after exercising the widget, and the content of the widget. DinoDroid is able to automatically learn a behavior model from a set of existing apps based on these features and the learned model can be used to test new apps.

Tab.(a)-Tab.(c) in Fig. 5.5 show the learning process of DinoDroid. In this example, DinoDroid dynamically records the *feature values*, including the visiting times of each event, the number of unvisited events in the next page (i.e., child page), and the text on the widget. An event is defined as the action taken to exercise a GUI widget (e.g., click a button). The three types of features represent the knowledge base of app testing and are used to learning agent to explore the app. DinoDroid uses a deep neural network to predict the accumulative reward (i.e., Q-value) of each event in the current page based on the aforementioned features and selects the event that has the largest Q-value to exercise.

Tab.(a) shows the feature values and the Q-values when the first time the page appears. Since "OK" has the largest Q-value, it is clicked. DinoDroid will then



-> (a) First time to visit this page
-> Click 4 and Visited Times+=1
-> Transfer to other page
-> Explore pages
-> Parent page to click "Minimum Length"
-> (b) or (c) Second time to visit this page

Event	Text	Visited Times	# Unvisited children	Q-Value
Click 1	restart	0	Nan	-0.31
Click 2	back	0	Nan	0.87
Click 3	menu	0	Nan	1.02
Click 4	OK	0	Nan	2.20
Click 5	Cancel	0	Nan	2.01

(a)First time to visit this page

Event	Text	Visited Times	# Unvisited children	Q-Value
Click 1	restart	0	Nan	-0.31
Click 2	back	0	Nan	0.87
Click 3	menu	0	Nan	1.02
Click 4	OK	1	10	0.93
Click 5	Cancel	0	Nan	2.01

(b) Second time to visit. "Click 4" has 10 unvisited children.

Click 4	OK	1	0	-1.76

(c) Second time to visit. "Click 4" has 0 unvisited children.

Figure 4.1: A Motivating Example

continue exploring the events in the new pages and updating the Q-value. When the second time the page appears, the Q-value associated with "OK" decreases because it is already visited. As such,"Cancel" has the largest Q-value and is exercised. In this case (Tab.(b)), the child page of "OK" contains 10 unvisited events. However, suppose the child page contains zero unvisited events (Tab.(c)), the Q-value becomes much smaller. This is because DinoDroid tends to select the event whose child page contains more unvisited events.

4.2.2 Problem Formulation

In the background section 2.2, we have discussed the Q learning agent receives a state s from environment and reward r from environment and select an action a to execute on the environment.

In Android GUI testing, a *state* s is encoded as an app page. We use s_t to represent the current state and s_{t+1} to represent the next page. An *action* a is an event issued by DinoDroid to exercise a GUI widget. An *event* e is the exercise of a GUI widget with a particular action type (e.g., click "search" button). A *reward* r is calculated based on the improvement of coverage. If code coverage increases, r is assigned a positive number (r=5 by default); otherwise, r is assigned a negative number (r=-2

by default). An *Agent* decides what action to take based on the accumulative rewards (i.e., Q-values) on the current observed state. A *Policy* is $\pi(a, s) = Pr(a_t = a|s_t = s)$, which maximizes the expected cumulative reward. Q-learning learns a policy to tell the agent what action to take.

4.2.3 Limitation of Existing Q-Learning Techniques

The techniques that are mostly related to DinoDroid are Q-learning-based Android app testing [77, 79, 80, 38]. These techniques all use traditional Q-learning based on a Q table. They have several limitations. First, Q-table cannot handle high-dimensional features, such as text and image. Each new feature space for a state will cause exponential growth in a Q-table. A feature, such as word embedding, can be represented as an N-dimension vector with M possible values, which would need $O(m^n)$ columns in a Q table. Second, most existing techniques use the resource ID of an event as the state of a Q table. However, different apps have different ID assignments. Therefore, they focus on training and testing individual apps and cannot train a model from multiple apps. Also the model trained from one app cannot be used to test another app. The only work that uses Q-table to transfer knowledge among apps is QBE [72]. However, the cost is expensive. Instead of using resource id to represent each state, QBE abstracts the page information into five states: too-few, few, moderate, many, and too-many, based on the number of widgets. It also abstracts actions into seven categories (i.e., menu, back, click, longclick, text, swipe, contextualg). As such, QBE is able to limit the size of Q-table. However, abstracting the states and actions could cause the learning to lose a lot of important information when making decisions of selecting events to explore.

Unlike traditional Q-learning, DinoDroid designs a novel deep neural network (DNN) model to process complex features with infinite feature space. Specifically, multiple complex features can be input into the DNN to obtain the Q value. For example, the DNN can handle word embedding with n-length vector and high-dimensional image matrix. By doing this, DinoDroid is able to understand the features at the widget-level and learn a more accurate behavior model. Since the features are general across apps, DinoDroid is able to train a model from multiple apps and transfer the learned knowledge into a new app.

4.3 DinoDroid Approach

Fig. 6.4 shows the overview of DinoDroid. An iteration t begins with an app page. In the current state s_t, DinoDroid selects the event e_t with the highest accumulative reward (i.e., Q-value), performs the corresponding action (a_t), and brings the app to a new state (s_{t+1}). After exercising a_t, DinoDroid employs a special event flow graph (EFG) to generate feature vectors for each event in s_{t+1}. The reward r_t is generated based on the observed code coverage and crash of App. DQN agent uses a neural network model to compute and update the Q-value for the event e_t responding with a_t. The learning process continues iteratively from the apps provided as the training set. When testing a new app, DinoDroid uses the learned model as the initial model

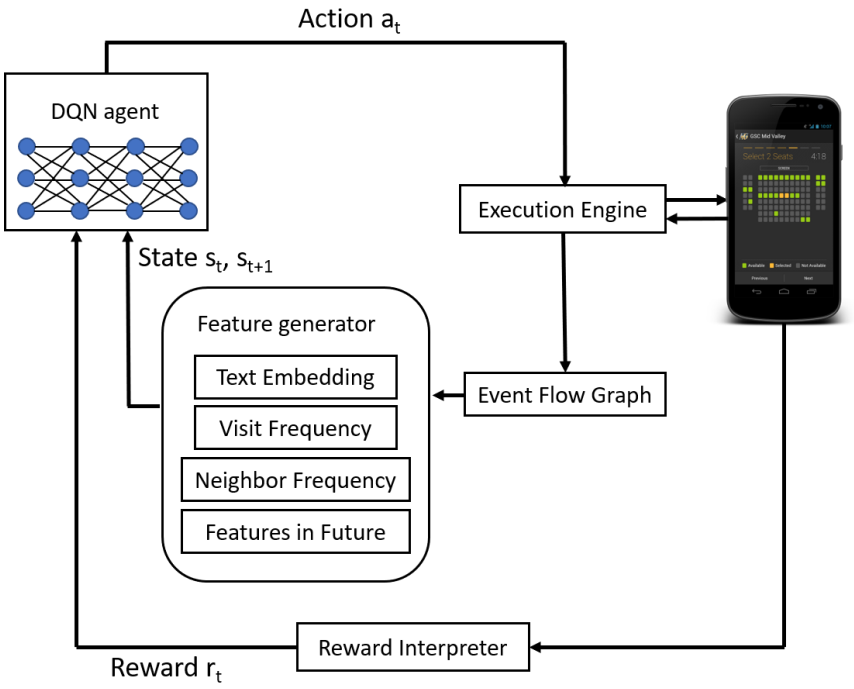


Figure 4.2: Approach Overview

to guide the testing of the new app. The model is updated following the same process until it is adapted to the new app (e.g., a time limit is reached or the coverage reaches a plateau).

4.3.1 DinoDroid's Algorithm

Alg. 1 shows the details of DinoDroid. When testing or training an app (AUT), DinoDroid checks If the behavior model exists. If so, the model will be updated to adapt the AUT; otherwise, DinoDroid starts building a new model (Line 2). A memory is used to record the samples generated from earlier iterations (Line 3). The event flow graph (EFG) is initialized for each app at the beginning of training or testing (Line 4). The details of the EFG will be described in Section 4.3.2.2. DinoDroid launches the app and reaches the first page. The initial state of DQN is obtained and the EFG is updated accordingly (Lines 5–6). The algorithm then takes the current page and the EFG to generate features for each GUI event in the current page (Line 7). DinoDroid considers three types of features as described in Section 4.3.2.

Now the iteration begins until a time limit is reached (Lines 8–22). To amplify the chance of bug detection, DinoDroid issues a random system event every at every 10 iterations (Line 9–10). At each iteration, DinoDroid uses the deep neural network model to select the event with the highest accumulative reward (i.e., Q-value) and perform the corresponding action a_t (Line 11). The details of the getActionEvent will be described in Section 4.3.3.2. After the execution, the algorithm obtains three kinds of information: the new page, the current code coverage, and the crash log

Algorithm 1 DinoDroid's testing

Require: App under test AUT, DQN's Model M with before knowledge, execution time LIMIT
Ensure: updated new M
1: **if** M not exist **then**
2: $M \leftarrow$ `buildNewModel`() /*First time to run*/
3: Memory $\leftarrow \emptyset$ /*Memory stores previous samples */
4: G $\leftarrow \emptyset$ /*Initialize event flow graph */
5: $p_0 \leftarrow$ `execute`(AUT) /*First lanuch to get the first page p_0*/
6: G $\leftarrow$ `updateGraph`(p_0, G) /*update G with new page/
7: $s_0 \leftarrow$ `featureGenerator`(p_0,G) /Every event in s_0 contains 3 features/
8: **while** t < LIMIT **do**
9: **if** t mod 10 equals 0 **then**
10: sendSystemEvent() /*send random system event*/
11: $a_t \leftarrow$ `getActionEvent`(s_t, M) /*Event selection */
12: p_{t+1}, codeCoverage, crash $\leftarrow$ `execute`(AUT, a_t)
13: $r_t \leftarrow$ `rewardInterpreter`(codeCoverage, crash)
14: G $\leftarrow$ `updateGraph`(p_{t+1}, G) /*update G with new page/
15: $s_{t+1} \leftarrow$ `featureGenerator`(p_{t+1},G)
16: $Q(s_t, a_t) = r_t + \gamma * \max_a Q(s_{t+1}, a)$ where $\gamma = 0.6$
17: $Q \leftarrow Q(s_t, a_t)$
18: batch $\leftarrow$ extactTrainBatch(Memory) $\cup$ (a_t,Q)
19: M $\leftarrow$ updateModel(batch, M) /*Learning for DQN */
20: Memory $\leftarrow$ Memory $\cup$ (a_t,Q)
21: $s_{t+1} \leftarrow s_t$
22: return M

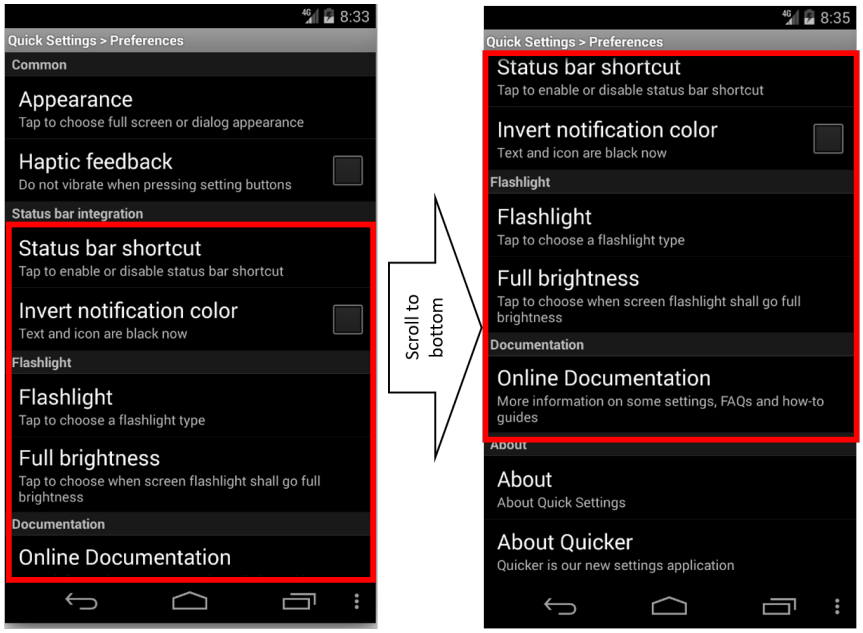
(may be empty) (Line 12). The information is used to compute the reward r_t (Line 13). Then, the event flow graph G is updated based on the new page p_{t+1} (Line 14). With the updated G and the new page, the algorithm can generate features for each event in the state s_{t+1} (Line 15). Based on s_t, s_{t+1}, a_t, and r_t, DinoDroid uses the equation 2.6 to compute the Q-value of each event in s_t (Line 16). To train the neural network, it uses a set of training samples, including both the current sample and history samples. Each sample uses a_t as input and Q-value as output (i.e., label). The history samples (obtained from earlier iterations) are recorded in a memory (Lines 17–21).

4.3.2 Feature Generation

DinoDroid's deep neural network model provides an interface that can handle any features provided by users (discussed in Section 4.3.3.1). DinoDroid currently supports three features: *Visiting times of current events*, *Visiting times of children events*, and *textual content of events*. DinoDroid employs an event flow graph (EFG) at runtime to obtain the features and transforms them into numerical vectors provided as inputs to the neural network.

4.3.2.1 Types of Features

Visiting times of current events (VTCR). The insight behind this feature is that a human often avoids repeated executing the events that bring the app to the same page. Therefore, all GUI events should be given chances to execute. Instead of using heuristics to weigh different events to decide which ones should be given higher



Preference Page, id=3 | Preference Scroll Page id=4

Preference Page, id=3

id	text	VTCR	Next	Similar
0	"restart"	0	-	4-0
1	"back"	0	-	4-1
2	"menu"	0	-	
3	"scroll"	1	4	
4	"appre.."	0	-	
5	"haptic.."	0	-	
6	"status.."	2	8	4-3
7	"inver.."	0	-	4-4
8	"flash.."	0	-	4-5
9	"full b.."	0	-	4-6
10	"onlin.."	0	-	4-7

Preference Page, id=4

id	text	VTCR	Next	Similar
0	"restart"	0	-	3-0
1	"back"	0	-	3-1
2	"menu"	0	-	3-2
3	"status.."	2	8	3-6
4	"inver.."	0	-	3-7
5	"flash.."	0	-	3-8
6	"full b.."	0	-	3-9
7	"onlin.."	0	-	3-10
8	"about"	0	-	
9	"about.."	0	-	

Figure 4.3: Event Flow Graph Example

priorities to execute [69], DinoDroid can automatically guide the selection of events based on the recorded feature value.

DinoDroid uses a vector with a length N as a feature vector to record the visiting time for each event, i.e., each element in the vector is filled in with the updated visiting time. The length is the same for all three features to ensure that DQN treats them equally during training. By default, N=10.

Visiting times of children events (VTCD). A human tester often makes a decision on which event to execute not only based on events of the current page, but

also considering the content of succeeding pages (children pages). For example, if executing an event e can trigger a page with unvisited events, e is likely to have a higher priority than the events only triggering pages with visited events. Therefore, DinoDroid considers the event visiting times of K generation of children pages as a feature. The m^{th} generation of children pages are defined as the succeeding pages with distance m from the current event in the event flow graph. By default $K = 3$.

Since each generation of children pages may contain a number of events, adding them all together into a feature vector (i.e., each element represents the visiting times of a single event in the children pages) may lead to an unbearable size of vectors for a neural network to train. Instead, DinoDroid designs a fixed length of feature vector for each generation children pages. By default, the length is equal to 10, which is the same size as the VTCR's feature vector. Each element in the vector represents the number of events in the target generation of children page that are visited N times, where N is equal to the index of the vector. For example, if an event is visited zero time, it is placed as the first element of the vector. If an event is visited 9 or more times, it is placed as the last element of the vector. For each generation of children pages, DinoDroid will generate one vector to represent the feature of the generation. In total, K features are generated to represent VTCD on each generation. One unique event in the App is only counted for one time in all generations.

Textual content of events (TXCT). The textual content of events may help DinoDroid make a decision on selecting events to execute. In the example of Fig. 5.5, the meaning of "OK" event indicates it has a higher chance to bring the app to a new page than the other events. To obtain the TXCT feature for each event DinoDroid first employs Word2Vec [102] to convert each word in the event to a vector with length L. By default, L=400. The Word2Vec model is trained from a public dataset text8 containing 16 million words and is provided along with the source code of Word2Vec [22]. As such, each word is associated with a vector. Therefore, the TXCT feature is encoded as a matrix $L * W$, where W is the number of the words in the event.

4.3.2.2 Compacted Event Flow Graph

DinoDroid uses an event flow graph (EFG) to obtain features. The graph is represented by $\mathcal{G} = (V, E)$. The set of vertices, V, represents events by exercising the app's clickable and long-clickable events, and the set of edges, E, represents event transitions (i.e., from one page to another by exercising the event). Each vertex records the three types of features described in Section 4.3.2 Fig. 4.3 shows an example, where each page is associated with a list of vertices (i.e., events with unique IDs) and their features. The last label ("Similar") is discussed later in the section.

DinoDroid's event flow graph is is compacted in order to accommodate DQN. This is also the main difference from traditional EFGs [7] . In traditional EFG, whenever a new page is encountered, it will be added as a vertex to the EFG. However, if we use the vertices as states in DQN, it could cause the states to be huge or even unbounded [64, 98, 103, 104, 69]. Also, when encountering a similar page with a minor difference from an earlier page, if DinoDroid treats it as a new state, it could generate

unbounded fake new events and thus waste exploration time. For example, in Fig. 4.3, there are many same events on the pages triggered by the "scroll" event. If we consider all of the events for each scroll triggered page, DinoDroid could waste time to visit the same events again and again without code coverage increase. Therefore, such similar pages should be combined to avoid this problem.

There have been techniques using heuristics to compact EFG for efficient exploration. For example, Stoat [69] ignores the details of ListView events by categorizing it into "empty" and "non-empty", so it can merge two similar pages with only the ListView events different into the same state. However, it may lose important information since some of the events triggered by the items under the ListView may be critical. In contrast, DinoDroid compacts the EFG by merging vertices instead of pages. Specifically, whenever a new page P' is encountered, DinoDroid retrieves the pages with the same Android Activity ID [105] from the existing EFG. It then compares the text of each vertex in P' with the vertices of each of the retrieved pages. If the texts are the same, the two events are merged as a single vertex. The "Similar" tag in Fig. 4.3 records the information of the merged vertex (i,e,, page id—event id). As such, when an event e is executed, the feature vectors/matrices of both e's vertex and its merged vertex are updated. Therefore, the same events will get equal chances to be executed.

4.3.3 DinoDroid's Deep Q-Network

In deep Q network, DinoDroid employs the learning model (e_t, r_t, s_{t+1}), meaning that if triggering e_t (i.e., exercising an event), the app will transit to a new state s_{t+1} and the event e_t is updated with a reward. One of the key components in DinoDroid is a Deep Neural Network Model, which uses equation 2.6 to compute the Q value. It takes the event's features as input and outputs the Q-value, which is equal to $r_t + \gamma * \max_a Q(s_{t+1}, a)$. In the training process, besides the last action's Q-value, DinoDroid adds four other randomly selected history samples in a same batch to train the neural network together. By doing this, it is easy for DQN to remember history samples when training new samples. Comparing to regular RL with Q-table, DQN is able to calculate Q-values from any dimensions of features for each event.

4.3.3.1 DNN's Feature Handler

DinoDroid's DNN model uses a *feature handler* to process each feature. A feature handler transforms the feature value into a vector, For example, in Fig. 4.4, embedding vectors of text "status bar shortcut" are input into the CNN and the maxpool will output a vector as the pre-process result [16]. The output is a vector representing the text feature. DinoDroid can be easily extended by writing new handlers to process other features. For example, users can take image as another type of feature by leveraging existing image classification models. The DNN model accepts the formats of single value, vector, and matrix. By default, single value and vector are handled by a fully connected layer and matrix is handled by 1-D CNN [106].

DinoDroid combines the vectors generated from all feature vectors into an one-dimensional vector. DinoDroid utilizes three fully connected layers [107] to process it. The last layer with a linear activation to output just one value to represent the Q value. We use a stochastic gradient descent algorithm, Adam [108], to optimize the model with learning rate 0.0001. The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

4.3.3.2 Event Selection

DinoDroid maintains a Q-value (reward) for each event. At the first K iterations, DinoDroid triggers random events to construct the DQN model. Randomness is necessary for an agent navigating through a stochastic maze to learn the optimal policy [109]. By default, K=20. After that, DinoDroid starts to use DNN to select event.

To determine which event to tirgger in the current page, DinoDroid uses the ε-greedy policy [110], a widely adopted policy in reinforcement learning, to select the next event. DinoDroid selects the event with the highest Q value with probability $1-\varepsilon$ and a random event with probability ε. The Q value is computed by the neural network model (Section 2.2). The value of ε can be adjusted by user. By default $\varepsilon = 0.2$. In order to amplify the chance of bug detection, DinoDroid also generates system-level events at every 10 iterations, such as screen rotation, volume control, and phone calls [69].

4.3.3.3 Reward Function

In reinforcement learning, the reward is used to guide training and testing. DinoDroid's reward function is based on code coverage. Specifically, DinoDroid sets the reward to a positive number (r=5) when the code coverage increases or revealing a unique bug, and a negative number (r=-2) when the coverage does not change. The absolute value of positive reward is larger than that of the negative reward is because we want the machine to favor higher coverage or detecting bugs. The reward values are configurable, however, these default values work well on different types of applications, as shown in the experiments.

Other reward functions, such as measuring the app state changes may also be used [38, 72, 76, 77, 79] For example, Q-testing [38] calculates the difference between the current state and the recorded states, e.g., the number of unvisited events in a state. While executing unvisited events may have a positive relationship with higher code coverage, it is not a direct measurement. It is very likely that new unvisited states do not lead to code coverage increase. In Fig. 4.3, the scroll event can reach a new state but it does not increase code coverage. On the other hand, an already visited event, when being executed again, could lead to coverage increase due to a different event flow path. For example, in a music play app, every time the 'play" is clicked to play a different song, the code coverage may increase. Therefore, DinoDroid

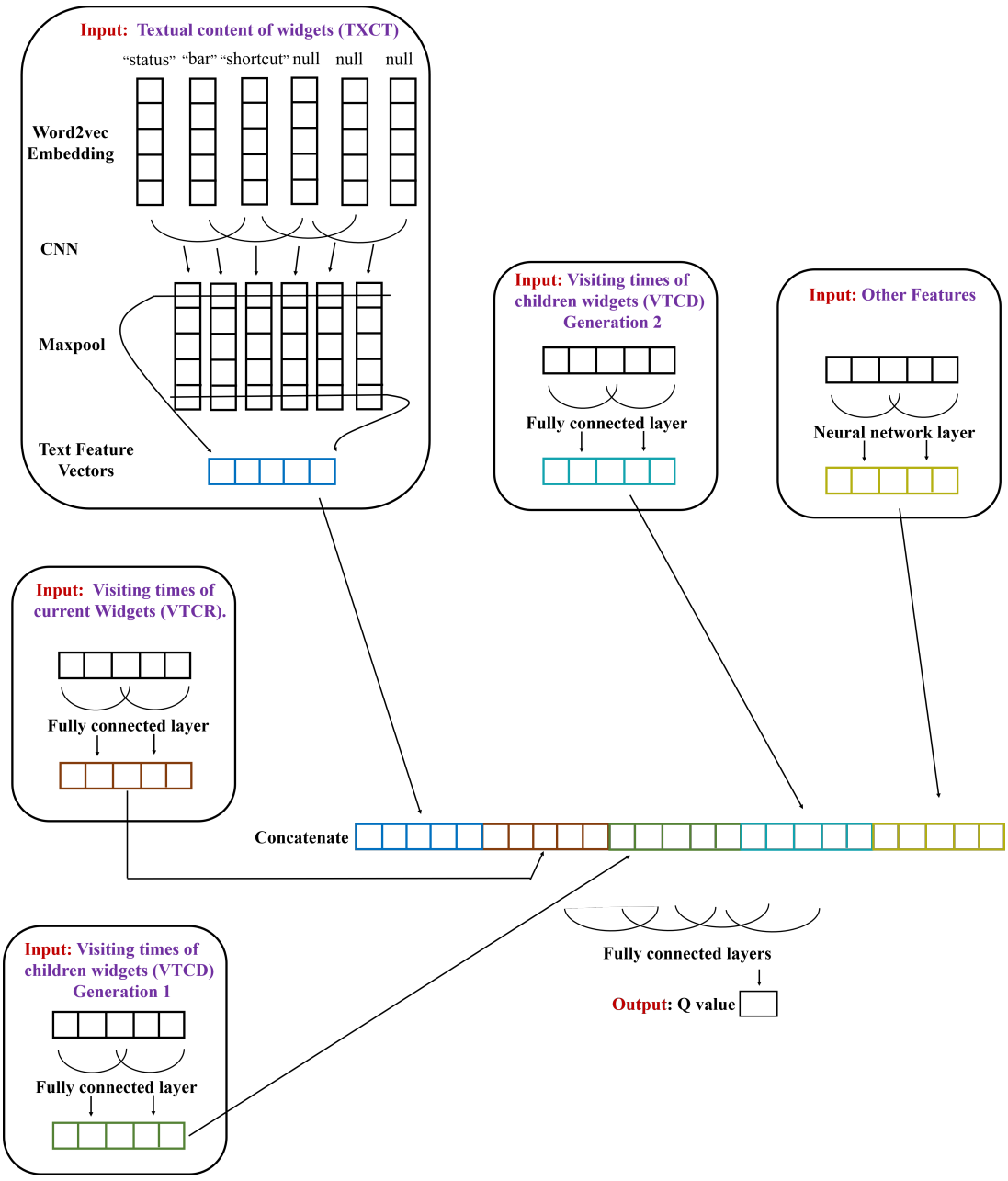


Figure 4.4: Deep Q-Network Model

uses code coverage directly to calculate reward while encoding the indirect measures (e.g., visiting times) as a type of feature.

4.4 Evaluation

To evaluate DinoDroid, we consider three research questions:

RQ1: How does DinoDroid compare with the state-of-the-art Android testing tools in terms of code coverage?

RQ2: How does DinoDroid compare with the state-of-the-art Android testing tools

in terms of bug detection?

RQ3: Can DinoDroid understand the features and learn a correct model?

4.4.1 Datasets

We need to prepare datasets for evaluating our approach. Since Sapienz [63] and Stoat [69] are two of the baseline tools compared with, we used the dataset [111] in their papers. The dataset contains a total of 68 apps. We removed four apps because they crash right after launch on our Intel Atom(x86) emulator. The executable lines of code in the apps range from 109 to 22,208, indicating that they represent apps with different levels of complexity.

4.4.2 Implementation

We conducted our experiment on a four-core 3.60ghz CPU physical x86 machine running with Ubuntu 16.04. The dynamic exploration component is implemented on UI Automator [112]. The system events issued during testing are obtained from Androguard [113]. DinoDroid uses Emma [114] to obtain statement coverage. Keras [115] is used to build and run the deep neural network. The DQN agent is implemented by ourselves using Python.

4.4.3 Study Operation

We performed a two-fold cross validation by random dividing the whole 64 apps data set into two sets with each set containing 32 apps. The training process is very costly. With an only 2-fold cross validation, it took 128 hours (nearly 6 days) to finish the experiment. With an added fold, it could take about three days extra.

The testing time of DinoDroid is set to one hour, which is further divided into three phases with each phase 20 minutes. For each phase, DinoDroid used the previously learned model as the initial model to test the app and update the model. By dividing the testing into multiple phases, it could avoid DinoDroid from repeatedly exploring the same events.

4.4.4 Comparison with Existing Tools

We compared DinoDroid with four existing Android testing tools: Monkey [5] , Stoat [69], Sapienz [63], and QBE [72]. Monkey is a random testing tool, Stoat uses model-based testing, Sapienz employs evolutionary testing, and QBE uses a traditional Q-learning. The details of the tools are discussed in Section 3. For Q-learning, we chose QBE because it is mostly similar with DinoDroid since it is able to transfer the knowledge learned from the training set to new apps without manual labeling work.

The testing time for all tools are set to one hour. Specifically, We followed previous work [111, 38] to set 200 milliseconds delay between events for Monkey to avoid abnormal behaviors. Stoat's default time settings have two phases: FSM and MCMC.

Following the suggestion of [38], each phase is set to 30 minutes. Since QBE is also RL method, the testing time was also divided into three phases as DinoDroid did.

4.5 Results and Analysis

4.5.1 RQ1: Code Coverage

Table 4.1 shows the results of code coverage obtained from DinoDroid and the other four tools on 64 apps. As shown in Fig. 4.5 on average DinoDroid achieves 49% line coverage, which is 21.6%, 15.8%, 14.5%, and 15% more effective than Monkey, Stoat, Sapienz, and QBE, respectively. Specifically, DinoDroid achieved the highest coverage in 32 apps, compared to 20 apps in Sapienz, 12 in Stoat, 3 in QBE, and 3 in Monkey. The results indicate *DinoDroid is effective in achieving high coverage.*

We analyzed the app code covered by different tools. Taking the app "Nectroid" [116] as an example, many events reside in deep levels of EFG, which require a number of steps to explore. For instance, two widgets are associated with important functionalities: "add a new site" and "delete a site". Exercising the first widget requires 7 steps (launched page → menu → settings → select a site→new site→fill blankets→OK) and exercising the second widget requires 6 steps. Monkey and Sapienz failed to cover the two particular sequences because they were not able to navigate the app to the deep levels. Stoat was very close to reach "add a site", but at the last step, it selected "Cancel" instead of "OK". On the other hand, DinoDroid exercised the "OK" widget in 90% of pages that contain "OK", "Cancel", and a few other functional widgets during the learning process. We conjecture that this is because DinoDroid is able to learn that clicking "OK" widget is more likely to increase coverage.

4.5.2 RQ2: Bug Detection

Table 4.1 shows the number of unique bugs detected by the five tools (on the left of "/"). Like Stoat, we consider a bug to be a crash or an exception. The results showed that DinoDroid detected the largest number of bugs (87) compared to Monkey(25), Stoat(62), Sapienz(21) and QBE (18). Specifically, DinoDroid detected most faults in 42 apps, which is more effect than Sapnize (5), Stoat (27), QBE (4), and Monkey (6).

DinoDroid and Stoat detected a significantly larger number of bugs because they both use androguard [113] to issue system-level events to amplify the chance of bug detection (Section 4.3.3.2) Therefore, for a fair comparison with Monkey, Sapienz, and QBE, we removed the bugs relevant to these system level events from the five tools.

DinoDroid and Stoat detected a significantly larger number of bugs because they both use androguard [113] to issue system-level events to amplify the chance of bug detection (Section 4.3.3.2) Therefore, for a fair comparison with Monkey, Sapienz, and QBE, we removed the bugs relevant to these system level events from the five

Table 4.1: Testing Result for Comparison

#APP.	# LOC	Code Coverage					# Fault Triggered				
		Mon.	QBE.	St.	Sap.	DD.	Mon.	QBE.	St.	Sap.	DD.
soundboard	109	31	42	42	63	42	0/0	0/0	0/0	0/0	0/0
gestures	121	26	32	32	52	32	0/0	0/0	0/0	0/0	0/0
fileexplorer	126	31	40	40	31	40	0/0	0/0	0/0	0/0	0/0
adsdroid	236	8	29	31	33	29	1/1	0/0	1/0	1/1	1/0
MunchLife	254	66	66	66	71	69	0/0	0/0	0/0	0/0	0/0
Amazed	340	66	69	58	74	73	2/2	0/0	0/0	1/1	1/1
battery	342	72	70	69	46	74	0/0	0/0	0/0	0/0	1/1
manpages	385	58	63	56	69	63	0/0	0/0	1/0	0/0	1/0
RandomMusicPlayer	400	53	54	74	57	60	0/0	1/0	0/0	0/0	1/0
AnyCut	436	61	62	61	65	62	0/0	0/0	0/0	0/0	0/0
autoanswer	479	11	13	26	18	22	0/0	0/0	1/0	0/0	1/0
LNM	492	54	55	61	52	57	0/0	0/0	2/0	0/0	1/0
baterydog	556	62	61	54	67	62	0/0	0/0	1/1	0/0	0/0
yahtzee	597	8	46	42	39	58	0/0	1/1	1/0	0/0	2/1
LolcatBuilder	646	27	20	25	31	53	0/0	0/0	0/0	0/0	0/0
CounterdownTimer	650	58	76	77	50	77	0/0	0/0	0/0	0/0	0/0
lockpatterngenerator	669	78	75	69	79	79	0/0	0/0	0/0	0/0	0/0
whohasmystuff	729	65	72	68	32	74	0/0	0/0	1/0	0/0	1/0
Translate	799	46	44	36	49	47	0/0	0/0	0/0	0/0	0/0
wikipedia	809	24	27	27	28	27	0/0	0/0	0/0	0/0	2/1
DivideAndConquer	814	83	81	52	80	58	0/0	0/0	0/0	1/1	0/0
zooborns	817	34	34	36	37	35	0/0	0/0	1/0	0/0	1/0
multismssender	828	46	46	51	60	68	0/0	0/0	1/0	0/0	1/0
Mirrored	862	44	45	45	46	45	0/0	0/0	1/0	0/0	1/0
myLock	885	26	27	44	30	41	0/0	0/0	2/1	0/0	2/1
aLogCat	901	66	68	71	43	79	0/0	0/0	0/0	0/0	0/0
aGrep	928	36	46	45	55	56	1/0	3/2	1/1	1/1	3/2
dialer2	978	37	39	32	40	46	0/0	0/0	1/0	0/0	1/0
hndroid	1038	9	9	9	10	9	1/1	0/0	1/1	1/1	1/1
Bites	1060	33	25	46	41	51	1/1	0/0	5/4	1/1	5/4
tippy	1083	82	58	72	82	84	0/0	0/0	0/0	0/0	0/0
weight−chart	1116	55	62	46	58	78	0/0	0/0	1/1	0/0	1/1
importcontacts	1139	40	41	31	42	41	0/0	0/0	0/0	0/0	0/0
worldclock	1242	89	92	92	90	91	0/0	0/0	1/0	0/0	1/0
blokish	1245	41	51	36	44	50	1/1	1/1	0/0	0/0	1/1
aka	1307	56	80	79	81	64	1/1	1/1	0/0	2/2	3/3
Photostream	1375	24	14	24	28	21	1/1	1/1	3/2	1/1	2/1
dalvik−explorer	1375	43	70	70	69	71	1/1	2/1	1/0	2/1	2/1
tomdroid	1519	51	50	53	51	53	0/0	0/0	0/0	0/0	1/1
PasswordMaker	1535	62	55	55	37	56	0/0	3/3	4/3	1/1	3/2
frozenbubble	1706	86	59	55	81	69	0/0	0/0	0/0	0/0	0/0
aarddict	2197	13	13	31	14	18	0/0	0/0	2/2	0/0	0/0
swiftp	2214	13	12	13	14	13	0/0	0/0	0/0	0/0	1/0
netcounter	2454	44	69	68	44	77	0/0	0/0	1/0	0/0	1/0
alarmclock	2491	66	67	68	57	71	1/1	0/0	3/1	1/1	2/0
Nectroid	2536	34	32	56	62	71	0/0	0/0	1/0	0/0	1/0
QuickSettings	2934	53	44	38	47	48	0/0	0/0	0/0	0/0	1/1
MyExpenses	2935	46	45	34	38	59	0/0	0/0	2/1	0/0	2/1
a2dp	3523	43	36	40	32	47	0/0	0/0	0/0	0/0	3/1
mnv	3673	18	28	48	12	43	2/2	1/1	1/0	0/0	2/1
hotdeath	3902	43	64	50	69	75	1/1	0/0	0/0	0/0	1/1
SyncMyPix	4104	21	20	25	21	26	0/0	0/0	1/0	0/0	1/0
jamendo	4430	21	23	16	24	28	0/0	0/0	2/1	0/0	3/1
mileage	4628	−	28	26	35	62	1/1	1/1	3/1	1/1	4/2
sanity	4840	15	15	22	15	35	1/1	0/0	1/0	2/1	2/1
fantastichmemo	8419	40	29	29	25	41	0/0	1/1	1/0	0/0	3/1
anymemo	8428	25	30	32	20	43	1/1	0/0	2/1	1/1	4/3
Book−Catalogue	9857	34	9	14	22	35	2/2	1/1	1/0	1/1	3/2
Wordpress	10100	5	4	4	4	6	0/0	0/0	3/2	0/0	2/0
passwordmanager	10833	11	7	12	5	8	0/0	0/0	0/0	0/0	1/0
aagtl	11724	16	17	15	18	18	2/2	1/1	2/2	1/1	2/2
morphoss	17148	10	18	17	15	25	2/2	0/0	0/0	0/0	3/1
addi	19945	14	18	17	19	18	2/2	1/1	2/1	1/1	2/1
k9mail	22208	5	7	8	5	7	0/0	0/0	2/0	1/1	2/0
Overall		40.3	42.3	42.8	42.6	49	25/24	18/16	62/26	21/19	87/41

DD.= DinoDroid. St.= Stoat Mon.= Monkey Sap.= Sapienz "-"=Not applicable(crash emulator) "a/b" indicates a=#crashes for all and b=#crashes without system level events

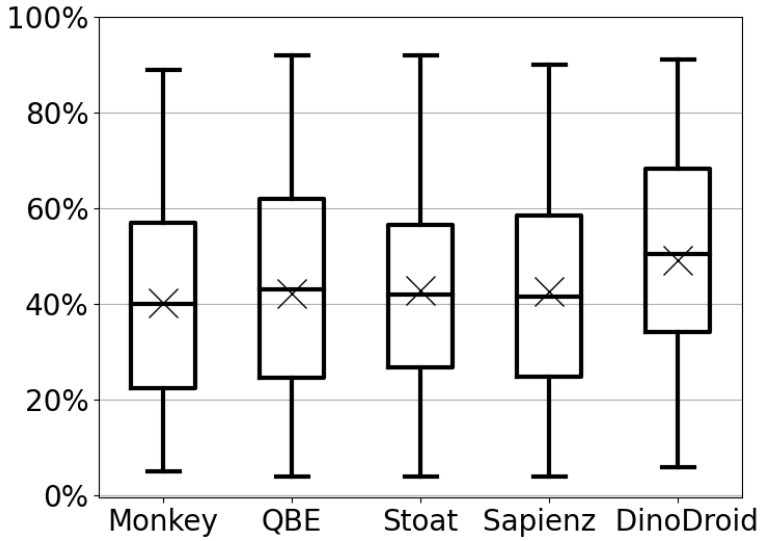


Figure 4.5: Line Coverage Comparison among Different Testing Tools

tools. To just consider the event exploration triggered crash, we do not count these crashes from system-level events. These crashes have particular crash log as below.

(1) android.app.ActivityThread.handleReceiver

(2) at android.app.ActivityThread.performLaunchActivity

(3) at android.app.ActivityThread.handleServiceArgs

(4) android.app.ActivityThread.performResumeActivity

The results (on the right of "/")still showed that DinoDroid detected the largest number of bugs (41), compared to Monkey (24), Stoat(26), Sapienz(19), QBE (16). Specifically, DinoDroid detected most number of bugs in 23 apps, which is more effective than Sapnize (7), Stoat (13), QBE (6), and Monkey (13).

The above results suggest that *DinoDroid is effective in detecting bugs*

Figure 4.6 shows the pairwise comparison of bug detection results between tools (with system-level events). For example, Stoat and DinoDroid detected 50 bugs in common. However, DinoDroid detected 27 bugs not detected by Stoat and Stoat detected 12 bugs not detected by DinoDroid. Figure 4.7 shows, with filtered out system level intent, DinoDroid still triggers more bugs than other methods. For example, DinoDroid triggers 26 bugs not detected by Stoat and Stoat detected 11 bugs not detected by DinoDroid

4.5.3 RQ3: Understanding the Learned Model

RQ3 is used to understand whether the DQN agent can correctly learn the app behaviors based on the provided features. To do this, we analyzed the traces generated by the DQN agent for the 64 apps.

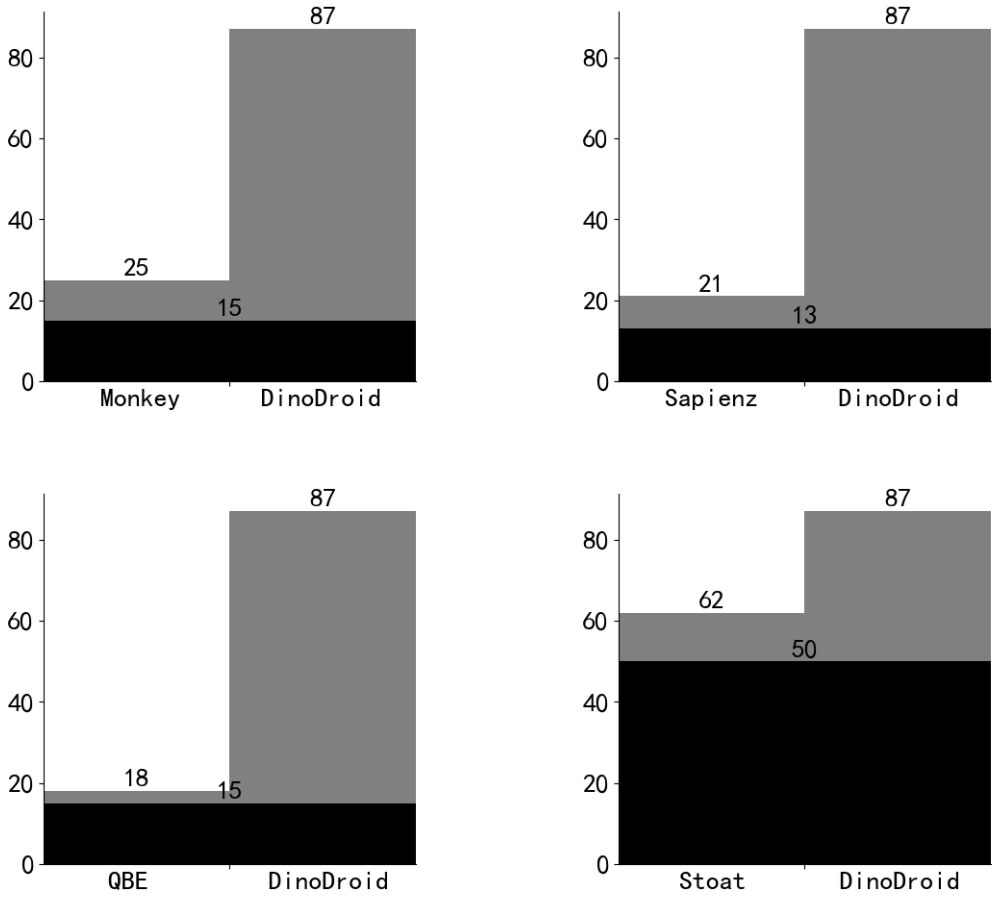


Figure 4.6: Comparison of tools in detecting crashes

4.5.3.1 Understanding the Features

VTCR feature. We first would like to understand the behavior of the VTCR feature. Specifically, we would like to know whether DinoDroid will give higher priority to unvisited events over visited events in the current page. We hereby computed the percentage of pages performing expected actions (i.e., triggering unvisited events) among all pages. We consider only unvisited and visited events instead of the number of visiting times because we need to control the factors of children pages since unvisited events do not invoke children pages. In total, there are 16,169 pages containing both visited and unvisited events, with a total of 197,279 events; 85.2% of the pages performed expected actions.

Note that it is possible that the visited and unvisited events are imbalance, e.g., there are significantly more unvisited events in every page, so the results could be biased because the possibility of selecting unvisited events is higher. We randomly selected an event from each page and found that 51.6% of the pages performed expected actions. This shows that our data is balanced and DinoDroid indeed behaves better than a random selection approach.

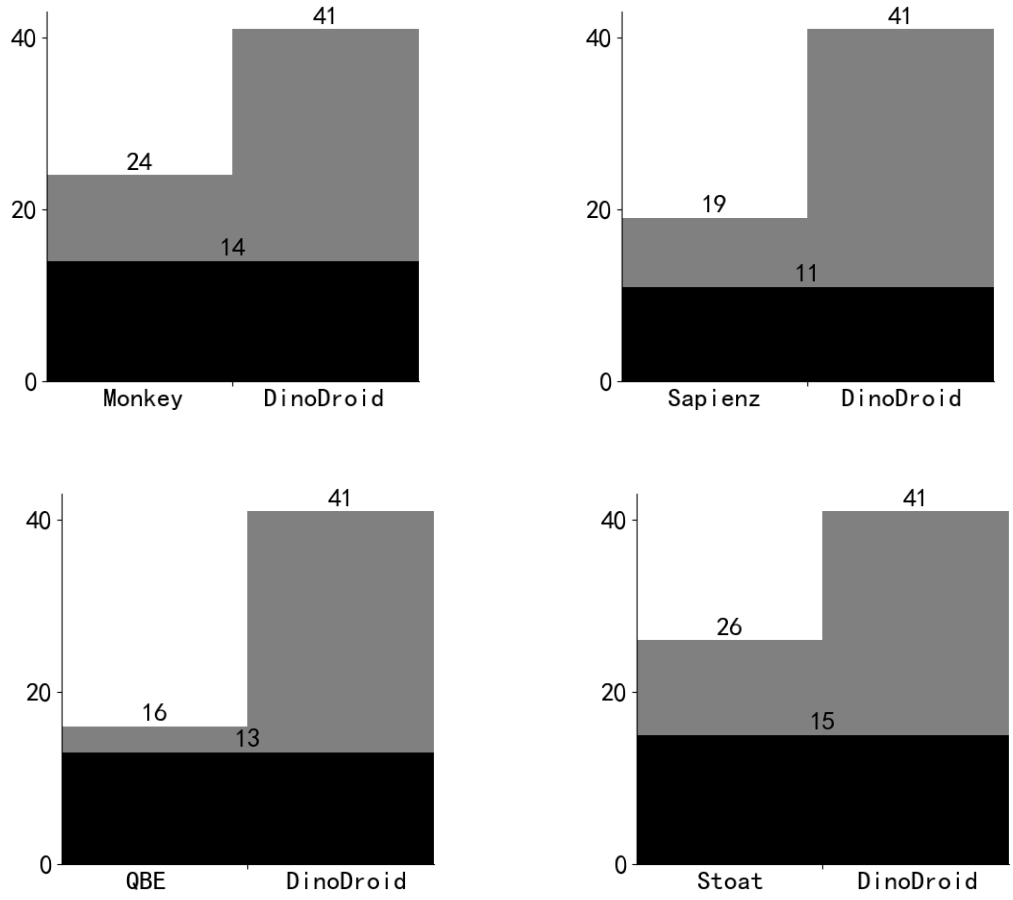


Figure 4.7: Comparison of tools in detecting crashes filter out activity intent

VTCD feature. We next would like to understand the behavior of the VTCD feature. Specifically, we would like to know whether DinoDroid tends to select events with unvisited children events over those with visited children events. To control the variation of event visiting times, we selected only pages that contain events with the same visiting times from the traces. As a result, 495 pages with a total of 3182 events were selected; 81.2 % of the pages performed expected actions (i.e., selecting events with unvisited children events). When using a random selection, only 33.6% pages performed expected actions.

TXCT feature. We also examined if DinoDroid is able to understand the content of events. To do this, we control the variations of VTCR and VTCD features by selecting pages that do not contain any visited events from the traces. There is a total of 3,060 pages. We hypothesize that if Q-values of these events are different, DinoDroid is able to recognize the content of the widgets (i.e., the expected action). The results show that 98% of pages performed expected actions with different Q-values.

The above results suggest that *the behaviors of the app features learned by DinoDroid are mostly expected.*

4.5.3.2 The Whole DQN Model Behaviors

The above experiment suggests how individual features affect the learning process of DinoDroid. We next conducted a deeper analysis to understand the behaviors of DinoDroid under the combination of the three features. To do this, we randomly selected a model from the two models learned from the 64 apps (by two cross-fold validation). We used this model to process the first page of the 32 apps, which were not used to train the model. We then manually set the values of VTCR and TXCT features while keeping the TXCT feature as default and see how the model behaves.

As shown in Table 4.2, "VTCR" indicates the number of times the current event is visited and "VTCD" indicates the three generations of children pages with the number of unvisited events and the number of events being visited once (i.e., the first two elements in the feature vector). Each number in the table is the Q-value averaged across all events with the same feature setting. For example, <(6#1); (1#1); (1#1)> + "1" (the second row + the third column) indicates that when an event is visited once, the first generation of its children page contains six unvisited events and one event is visited once, and the second and third generation of its children pages have one unvisited events and one event is visited once.

DinoDroid learned the following behaviors as shown in Table 4.2. First, the Q-value of (6#1);(1#1);(1#1) is larger than that of (1#6);(1#1);(1#1). This indicates that *an event with more unvisited children events in the first generation is more likely to be selected*. Second, the Q-value of (1#1);(6#1);(1#1) is larger than that of (1#1);(1#6);(1#1) and the Q-value of (1#1); (1#1); (6#1) is larger than that of (1#1); (1#1); (1#6). This indicates *the second and third generation (or perhaps the subsequence generations) of children events can also guide the exploration like the first generation*. Third, *the unvisited event in the current page has the highest priority to be selected* since its Q-value is significantly larger (i.e., VTCR=0). Fourth, *with the same VTCD feature values, the event with less visiting times in the current page has a higher priority to be selected* since the Q-value is decreasing as the value of VTCR increases.

All of the above behaviors *align with intuition about how human would test apps with the three types of features and demonstrate that DinoDroid's DQN can automatically learn these behaviors without the need to manually set heuristic rules.*

4.6 Limitations

DinoDroid currently handles only three features. As part of the future work, we will assess whether other features, such as images of widgets, can improve the performance of DinoDroid, Second, while DinoDroid can handle any feature on GUI, which can be represented as matrixes or vectors, some features may not have a straightforward representation by a simple matrix or a simple vector. For example, the complete visited event sequence feature with text information can only be represented by multiple complex matrices. It may be very time consuming for a machine to process this kind of features.

Table 4.2: DinoDroid's behavior on every specific feature combination

VTCD \ VTCR	0	1	2	3	4	5
<(6#1), (1#1), (1#1)>	-	3.45	2.32	1.2	-0.65	-0.47
<(1#6); (1#1); (1#1)>	-	2.57	0.23	-0.171	-3.35	-3.55
<(1#1); (6#1); (1#1)>	-	3.44	2.03	0.16	0.19	0.00
<(1#1); (1#6); (1#1)>	-	1.01	0.92	-0.31	-3.59	-3.96
<(1#1); (1#1); (6#1)>	-	1.92	0.97	1.07	-0.9	-3.2
<(1#1); (1#1); (1#6)>	-	1.09	0.81	0.21	-3.3	-4.53
<(1#1); (1#1); (1#1)>	-	1.01	0.78	-1.66	-3.86	-4.32
<(0#0); (0#0); (0#0)>	10.96	-4.42	-4.98	-5.09	-5.13	-5.14

<(A#B); (C#D); (E#F)>: the first generation has A unvisited events and B events are visited once; the second generation has C unvisited events and D events are visited once; the third generation has E unvisited events and F events are visited once.

4.7 Conclusions

We have presented DinoDroid, an automated approach to test Android application. It is a deep Q-learning-based approach, which can learn how to test android application rather than depending on heuristic rules. DinoDroid can take more complex features than existing learning-based methods as the input by using a Deep Q learning-based structure. With these features, DinoDroid can process complex features on the pages of an app. Based on these complex features, DinoDroid can learn a policy targeting at achieving high code coverage. We have evaluated DinoDroid on 64 apps from a widely used benchmark and showed that DinoDroid outperforms the state-of-the-art and state-of-practice Android GUI testing tools in both code coverage and bug detection. By analyzing the testing traces of the 64 apps, we are also able to tell that the machine really understands the features and provides a sensible strategy to generate tests.

Chapter 5

ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports

In this chapter, we propose an approach ReCDroid [1] which 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 events with the goal of reproducing the reported crash.

5.1 Overview

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 [112]).

ReCDroid differs from prior work for analyzing the reproducibility of bug reports [42, 82] 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 [84, 42, 86, 83] because they focus extracting useful information (e.g., test cases [83]) without directly targeting at reproducing crashes.

[1] The contents of this chapter have appeared in [117].

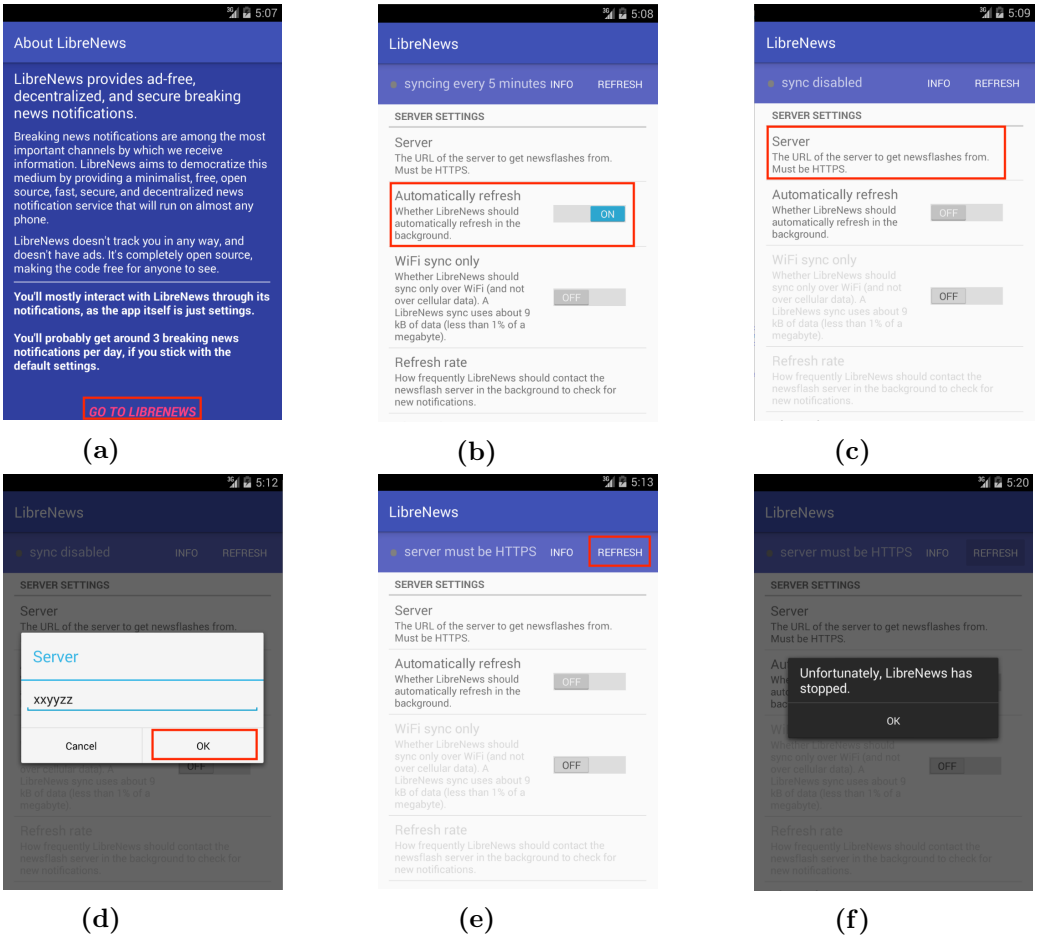


Figure 5.1: The steps of reproducing the crash described in Fig. 5.2.

ReCDroid has been implemented as a software tool on top of two execution engines — Robotium [55] and UI Automator [112]. 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.

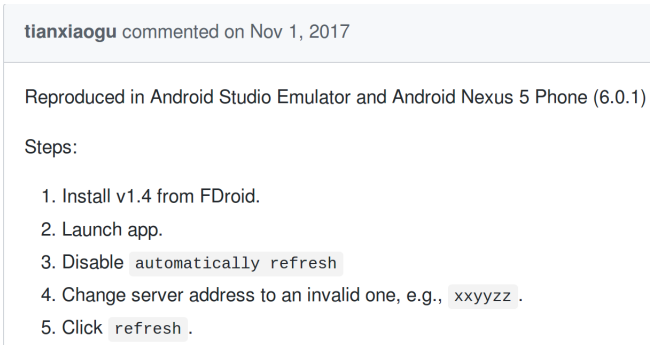


Figure 5.2: Bug Report for LibreNews issue#22

5.2 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 [118] and GitHub [12]. 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 keyworld "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).

5.3 Design Challenges

An example bug report is shown in Fig. 5.2. 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. 5.1 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 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

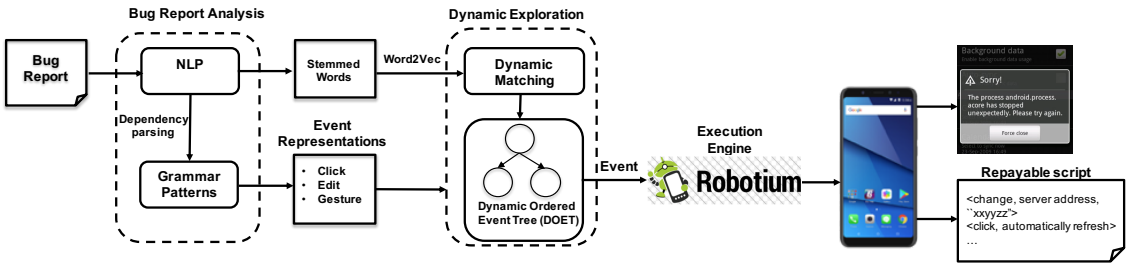


Figure 5.3: Overview of the ReCDroid Framework.

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 6.3.

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. 5.2. 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 "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 [119, 120] 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 [121] 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 6.3. 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 [102] 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 [9, 122, 63, 8, 68] are not a good fit for this particular need. For example, many existing tools (e.g., A3E [9]) 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., [9, 8, 68]) that add backtracking steps to their crawling sequence, which results in an overall much longer sequence of reproducing actions.

Table 5.1: Summary of Grammar Patterns

Category	*ID*	*Pct.*	*Grammar Pattern*	*Example*
Click	CR1	12.5%	`action` → `dobj` (→`NP`)	Click$_{[action]}$ {easy level$_{[dobj]}$}$_{[NP]}$
	CR2	0.7%	`action` → `nsubjapss` (→ `NP`)	{Easy level$_{[dobj]}$}$_{[NP]}$ is clicked$_{[action]}$
	CR3	8.6%	`action` → `pobj` (→ `NP`)	I made a click$_{[action]}$ on {easy level$_{[pobj]}$}$_{[NP]}$
Edit	TR1	7.3%	`action`→`dobj`|`obj`|`attr`→`prep`→`pobj` (→`NP`) `prep` ∈ {`on`,`in`,`to`}	Input$_{[action]}$ xxyyzz$_{[dobj]}$ to$_{[prep]}$ {server address$_{[pobj]}$}$_{[NP]}$
	TR2	1.8%	`action`→`dobj`|`obj`|`attr`(→`NP`)→`prep`→`pobj` `prep` ∈ {`with, by`}	Input$_{[action]}$ {server address$_{[dobj]}$}$_{NP}$ with$_{[prep]}$ xxyyzz$_{[pobj]}$
	TR3	0.7%	`action`→`dobj`|`obj`|`attr`(→`NP`)→`prep`→`pobj` `prep` ∈ {`to, with`}, `action` ∈ {`change`}	Change$_{[action]}$ {server address$_{[dobj]}$}$_{NP}$ to$_{[prep]}$ xxyyzz$_{[pobj]}$
	TR4	0.6%	`TR1`|`TR2` + EG(`NOUN` → `NUM` → `UNIT` | `STR`)	Input$_{[action]}$ a number$_{[dobj]}$ to kilometer$_{[pobj]}$ e.g., {10$_{[NUM]}$ km$_{[UNIT]}$}$_{EG}$
Gesture	NR1	0.4%	`action`	Rotate$_{[action]}$ the screen

5.4 ReCDroid Approach

The architecture of ReCDroid is shown in Fig. 6.4. 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.

5.4.1 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.

5.4.1.1 Grammar Patterns

The 22 grammar patterns were derived from the corpus of 813 Android bug reports described in Section 5.2. 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. 5.4 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 6.1 lists the eight typical grammar patterns (The full list can be found in our artifacts [19]). Column 3 shows the percentage of the 813 bug reports in which each grammar pattern applies. We next describe these patterns.

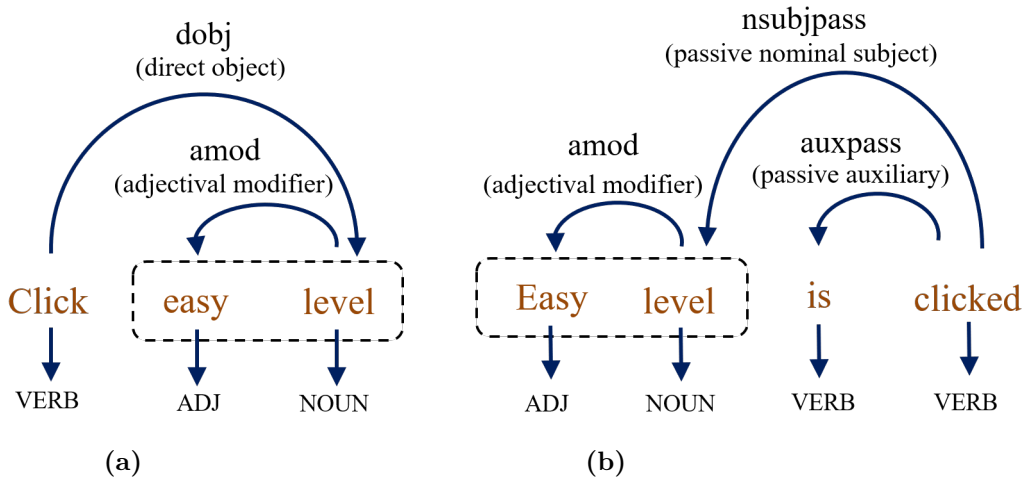


Figure 5.4: Examples of Dependency Trees

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 6.1, 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 6.1, 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`$\in${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.

5.4.1.2 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 [121]. It then applies *stemming* [123][2] 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 [19]. 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. 5.2 contains a word "change", so the grammar pattern TR3 is applied.

5.4.1.3 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 [124], 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 5.4.2.2 illustrates how ReCDroid obtains additional information from unstructured texts to address the mismatch between bug reports and target apps.

5.4.2 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.

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

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.

5.4.2.1 ReCDroid' Guided Exploration Algorithm

Algorithm 2 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 5.4.2.4), 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 3). 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

Algorithm 2 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*/

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 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, $R = 2$.

5.4.2.2 Dynamic Matching

To determine whether a GUI component matches a bug report (Line 11), ReCDroid utilizes `Word2Vec` [102], 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` [22]. 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 3 `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** $false$

Algorithm 3 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[3], 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 [127, 128, 129].

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 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.

5.4.2.3 A Running Example

Fig. 5.5 shows a partial DOET for the example in Fig. 5.1. 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. 5.1a. 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

[3]An n-gram is a contiguous sequence of n items from a given sequence of text, which has been widely used in information retrieval [125] and natural language processing [126].

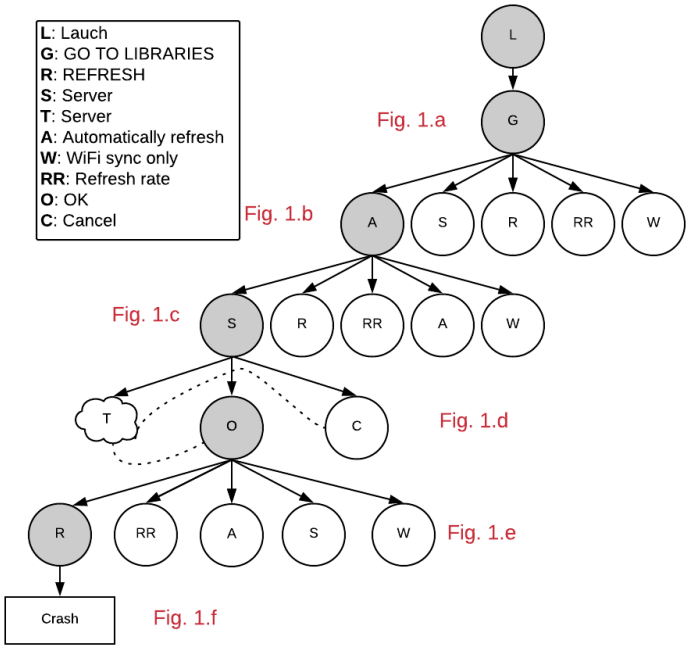


Figure 5.5: Dynamic Ordered Event Tree (DOET) for Figure 5.1

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. 5.1b. 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. 5.1b) and selects the leftmost leaf (A) to execute, which brings the app to the screen of Fig. 5.1c. 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. 5.1d, 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. 5.1d), 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. 5.1c, 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. 5.1b would be selected. So in the next iteration, ReCDroid restarts the search and executes the sequence $L \rightarrow G \rightarrow S$.

5.4.2.4 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.

5.5 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?

5.5.1 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 330 bug reports containing the keywords "crash" and "exception" from GitHub. We next included all 15 bug reports from the FUSION paper [82] and 25 bug reports from a recent paper on translating Android bug reports into test cases [83]. 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.

5.5.2 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 [121]. The dynamic exploration component was implemented on top of two execution engines, Robotium [55] and UI Automator [112], 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.

5.5.3 Experiment Design

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

5.5.3.2 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 6.8). The time limits for running ReCDroid$_N$ and ReCDroid$_D$ were also set to two hours.

5.5.3.3 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.

5.5.3.4 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 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.

5.6 Results and Analysis

Table 6.5 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., [130]). 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.

5.6.0.1 RQ1: Effectiveness and Efficiency of ReCDroid

As Table 6.5 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 [82] and nine bug reports (marked with $*$) from Yakusu [83] 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 6.8.

Table 5.2: 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⋆	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⋆	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⋆	3	3 (0, 1)	✓	✓	✓	6	6	6	216.2	292	1731.1	11
olam-2⋆	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).

5.6.0.2 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 backtrack was needed, they restarted the search from the home screen of the app (Algorithm 2). 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.

5.6.0.3 RQ3: Usefulness of ReCDroid

The last column of Table 6.5 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 6.5 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 reproducing 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 [131] 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

Table 5.3: 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)

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.*

5.6.0.4 RQ4: Handling Low-Quality Bug Reports

Columns 2–7 of Table 6.7 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.*

5.7 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 [132, 133]. 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 [134]. 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 [63, 5, 135, 69, 9, 7], which detect crashes in an unguided manner. ReCDroid$_D$ was shown to be competitive with Monkey [5], Sapienz [63], and the recent work Stoat [69] 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 [19].

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

One Related Work A tool called Yakusu [83] translate executable test cases from bug reports. It is probably most related to our approach. However, Yakusu translates 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. 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.

5.8 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.

Chapter 6

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

6.1 Introduction

In this chapter, we propose an approach ReCDroid+ extends and refines a previously presented section 5. Specifically, in the previous section of ReCDroid, it requires developers to manually extract S2R from bug report and label crash bug reports as the input of ReCDroid. Developers also need to manually verify the crash statements in the bug report because ReCDroid can only reproduce the crash bug report. In this chapter, ReCDroid+ is designed to automatically extract S2R sentence and identify the crash sentence. Then ReCDroid+ is a end-to-end bug reproduction tool meaning that no human-effort is needed in the whole bug reproduction process.

Firstly, we implement a preliminary work S2RMiner that employs support vector machine (SVM) to extract S2R sentences from a bug report. S2RMiner combines n-grams and CountVectorizer [137] to transform text features into numerical features. However the approach of S2RMiner is too straight forward to extract accurate S2R sentences which only achieves F score as 0.65. The data set of S2RMiner is also small as 1000 bug reports which are not enough to train a high performance model.

To achieve a higher accuracy of the S2R extraction and add a new functionality of identifying the crash sentence, we design ReCDroid+ which employs a deep learning model designed by CNN & LSTM. So ReCDroid+ can potentially achieve better performance than the traditional SVM used by S2RMiner. ReCDroid+ achieves a higher F1 score 0.70 in S2R extraction than S2RMiner 0.65. ReCDroid+ achieves a high crash sentence identification F1 score 0.79. The data set is also extended to 4000 bug reports. Besides the deep model and big data set, a set of rules are developed to overcome the false positives and false negatives of the extraction. It can also identify whether the bug report is in reproducing usage scope.

To determine the effectiveness of our approach, we add 15 new bug reports into the

51 bug reports data set in original ReCDroid. Different from the original ReCDroid, ReCDroid+] can input the raw HTML bug reports as the input rather than manual extracted S2R sentences. We ran ReCDroid+ on 66 raw bug reports from 37 popular Android apps to end-to-end automatically reproduce crash bug report. ReCDroid+ was able to successfully reproduce 42 (63.6%) of the crashes.

To determine the usefulness of our tool, we repeat the evaluation of ReCDroid on the extended data set, a light-weighted user study is conducted to show that ReCDroid+ can reproduce 21 crashes not reproduced by at least one developer. We invited another four participants to re-write the bug description for the 42 reproduced crashes. ReCDroid+ can detect all 93% of the 168 bug reports which means ReCDroid+ has a high effectiveness in handling bug reports written by different users. The robustness of ReCDroid+ in handling low-quality bug reports is evaluated on the data set which 10%, 20%, and 50% content from the 42 original reproduced bug reports are randomly removed. Among all 630 mutated bug reports, ReCDroid+ can reproduce 88% of them. We consider ReCDroid+ is a useful approach in automatically reproducing bug crashes.

6.1.1 Challenges of extracting Information from Bug Reports

An example bug report is shown in Fig. 5.2. 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. 5.1 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.

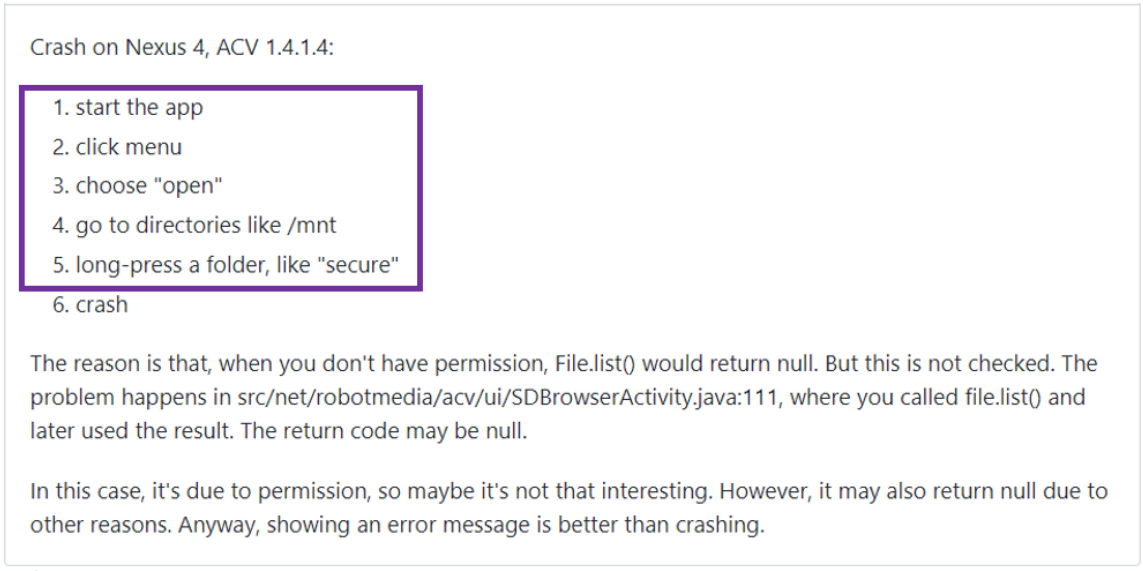
There are two challenges in identifying the needed information for crash reproducing. First, we need to determine whether a given bug report involves a crashed output because our goal is to reproduce crashes. Second, we need to extract S2R sentences.

A bug report often contains mixed types of information, such as comments, code, status of the issue, and information unrelated to the bug. Fig.6.1 shows an example bug report. Given the whole text description, it is unclear which sentence belongs to S2R. Therefore, existing bug reproduction tools (e.g., YAKUSU and ReCDroid) cannot directly work on the raw description of the bug report.

Fig.6.1 shows an example bug report. Given the whole text description, it is unclear which sentence belongs to S2R.

ReCDroid+ is designed and implemented to accurately extract all S2R sentences from the bug report. Specifically, ReCDroid+ takes the HTML format of the issue page (Fig.6.2) as input and outputs a sequence of S2R sentences (i.e., the text inside the rectangle indicates S2R). If a bug report does not have S2R (e.g., Fig. 3), ReCDroid+ will report S2R is missing.

To automate the process of identifying crash bug reports and extracting S2R, we designed a novel deep learning model that can automatically identify sentences involving the crash symptoms and S2R. Unlike traditional text classification methods [138, 139, 140], ReCDroid+'s deep learning model targets the bug report sentences



Crash on Nexus 4, ACV 1.4.1.4:

1. start the app
2. click menu
3. choose "open"
4. go to directories like /mnt
5. long-press a folder, like "secure"
6. crash

The reason is that, when you don't have permission, File.list() would return null. But this is not checked. The problem happens in src/net/robotmedia/acv/ui/SDBrowserActivity.java:111, where you called file.list() and later used the result. The return code may be null.

In this case, it's due to permission, so maybe it's not that interesting. However, it may also return null due to other reasons. Anyway, showing an error message is better than crashing.

Figure 6.1: A bug report

```
<task-lists disabled="" sortable="">
<table class="d-block">
  <tbody class="d-block">
    <tr class="d-block">
      <td class="d-block comment-body markdown-body  js-comment-body">

          <p>Crash on Nexus 4, ACV 1.4.1.4:</p>
<ol>
<li>start the app</li>
<li>click menu</li>
<li>choose "open"</li>
<li>go to directories like /mnt</li>
<li>long-press a folder, like "secure"</li>
<li>crash</li>
</ol>
<p>The reason is that, when you don't have permission, File.list() would
 return null. But this is not checked. The problem happens in
src/net/robotmedia/acv/ui/SDBrowserActivity.java:111, where you called
file.list() and later used the result. The return code may be null.</p>
<p>In this case, it's due to permission, so maybe it's not that
interesting. However, it may also return null due to other reasons.
Anyway, showing an error message is better than crashing.</p>
      </td>
    </tr>
  </tbody>
</table>
</task-lists>
```

Figure 6.2: HTML format of Fig.6.1

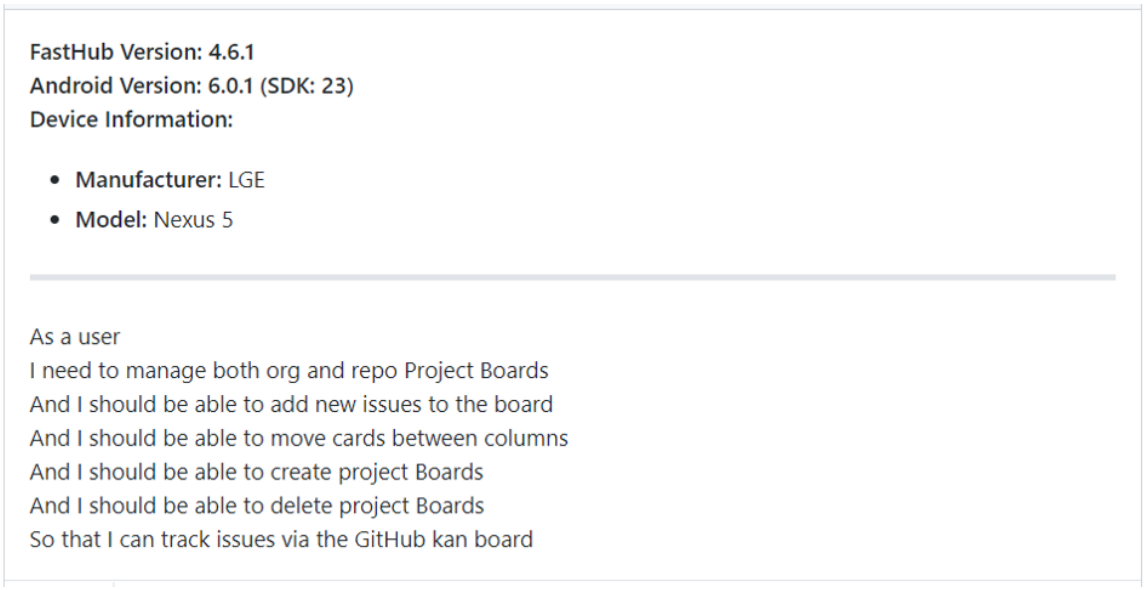


Figure 6.3: Missing S2R

and takes into acount 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.

6.2 S2Rminer Approach

S2Rminer [1] consists two major phases. In the first phase, S2Rminer uses a HTML parser to extract the key text containing S2R from the HTML format of an issue page. As shown in Fig.2, the HTML issue page contains HTML tags, such as < /li>,, and <tbody class="d-block">. S2Rminer filters out the HTML tags and obtains only text "start the app".

In the second phase, S2Rminer uses NLP techniques to extract text features from the sentences of the filtered text. It then uses machine learning to label whether a sentence belongs to S2R. Finally, the sentences labeled with S2R are saved into a output file.

[1]The contents of this chapter have appeared in [141].

6.2.1 Phase 1: HTML Parsing

Many bug tracking systems allow reporters to submit bug reports through web pages and developers can reply to the bug report by adding comments to the page. Therefore, bug report descriptions are often downloaded as HTML files. The original HTML file has a number of HTML tags. In addition, the raw HTML file contains many other types of information, such as bug symptoms, expected behaviors, developers' replies, CSS code, page information, and so on. These types of information are irrelevant to S2R. Even on a simple bug report shown in Fig.1, the associated HTML file contains 1371 lines and is as large as 104 KB. S2Rminer needs to eliminate all such information to obtain the minimum amount of text containing S2R.

Specifically, S2Rminer removes all HTML tags and parses the first block of text in the HTML page. The intuition is that only the first comment involves S2R described by the reporter.

6.2.2 Phase 2: S2R Extraction

The problem of detecting S2R sentences can be formulated into the problem of text classification [142]. Given a sentence, a text classification tool can predict whether it is a S2R sentence or not. S2Rminer performs the classification in three steps. First, it splits the text into individual sentences by employing several heuristics. Second, for each sentence, S2Rminer extracts text features used for building a classifier. Third, leveraging the text features, S2Rminer builds a classifier that can predict whether a sentence is S2R. All S2R sentences are saved into an output file.

Splitting text into sentences. S2Rminer first needs to detect individual sentences for being labeled as S2R sentence or non-S2R sentence. We cannot simply view each text line as a sentence because a line may contain more than one sentences. In the example of Fig. 1, the first line in the second to last paragraph ("The reason is that ...") contains two sentences. While tools such as spaCy [143] have the capability of detecting sentences, they are not accurate because they are not intended to deal with bug report text.

To address this problem, S2Rminer designs several heuristics to identify sentences from each line of the text: 1) one text line contains at least one sentence; 2) a text segment ending with a full stop "." is a sentence; 3) if a full stop is preceded by a number (e.g., "1.") or a part of ellipsis, it is not considered to be the end of a sentence.

Extracting text features. S2Rminer employs a well-known NLP tool spaCy [143] to extract text features from each sentence. We consider three types of features. The first type of feature is *stemming*, which transforms each word in the sentence to its stem. Stemming is the process of removing the ending of a derived word to get its root form. For example, "clicking", "clicks", and "clicked" become "click". Without stemming, multiple words with the same meaning would be used as different features, resulting in too many features and thus a low quality machine learning model.

The second type of feature is *part-of-speech (POS)* tags, which labels each word with a POS tag. The features used by S2Rminer are words labeled as "noun", "verb", and "adjective".

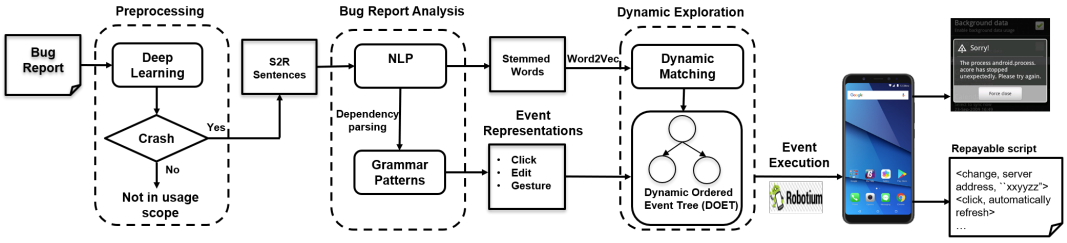


Figure 6.4: Overview of the ReCDroid+ Framework.

The third type of features is *dependency parsing*, which analyzes the grammar structure of the sentence. Specifically, words labeled as root, predicate, and object are considered as features.

Building a text classifier. We use n-grams and CountVectorizer [144] to transform text features into numerical features, which is easy to process by a machine learning tool. A n-gram a contiguous sequence of n items from a given sequence of text. For example, 1-gram (or unigram) indicates single word tokens and 2-gram (or bigrams) indicates two consecutive word tokens.

The current implementation of S2Rminer uses Support Vector Machines [145] (SVM) to do binary classification given the extracted text features. SVM outputs a "1" if a sentence is a S2R and a "0" otherwise. S2Rminer saves the sentence labeled with "1" into the result file for each bug report.

6.3 ReCDroid+ Approach

The architecture of ReCDroid+ is shown in Fig. 6.4. ReCDroid+ consists of three major phases — preprocessing, bug report analysis, and dynamic exploration. The preprocessing phase employs NLP and deep learning techniques to identify crash bug reports and S2R sentences. To carry out the bug report analysis, ReCDroid+ employs NLP techniques and heuristics to summarize a set of grammar patterns for different types of events. It then uses these grammar patterns 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.

6.3.1 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.

6.3.1.1 HTML parsing

In order 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, in a bug report with only 10 lines of bug description [146], the associated noisy information contains more than a thousand of 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 tag "//td". ReCDroid+ utilizes an HTML parsing tool called `lxml` [15] 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 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. Therefore, ReCDroid+ detects and annotates such tags, which will be later used in extracting S2R sentences (Section 6.3.1.2).

6.3.1.2 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 [121] 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 [147, 148], 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 Fig. 5.2, a sentence right before or right after a S2R sentence is more likely to be a S2R sentence than the others. In addition, sentences that follow the text indicating steps to reproduce (e.g., a word "steps") has a higher change 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 with the adjacent context, especially in the java error message written in multiple adjacent sentences.

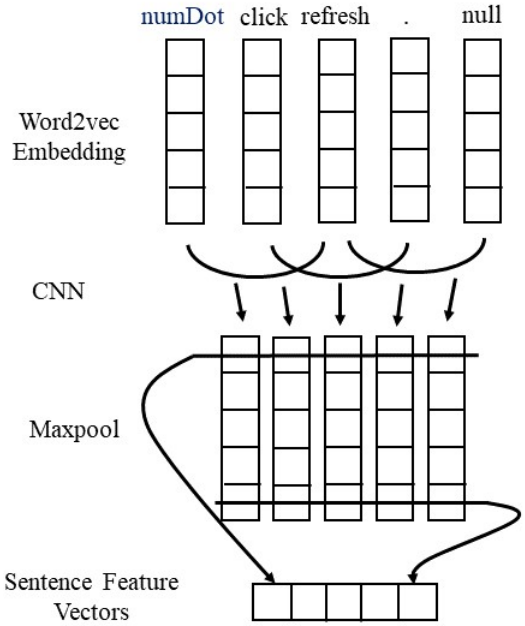


Figure 6.5: The convolutional neural network extracts sentence features from each word. The word embedding vector are per-trained through word2vec.

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 Fig. 6.5. Next, based on the extracted *word* feature vector, a CNN and max layer are used to 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 Fig. 6.6. We next describe the three steps used to detect S2R and crash sentences.

Word embedding. ReCDroid+ uses Word2vec [22] 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

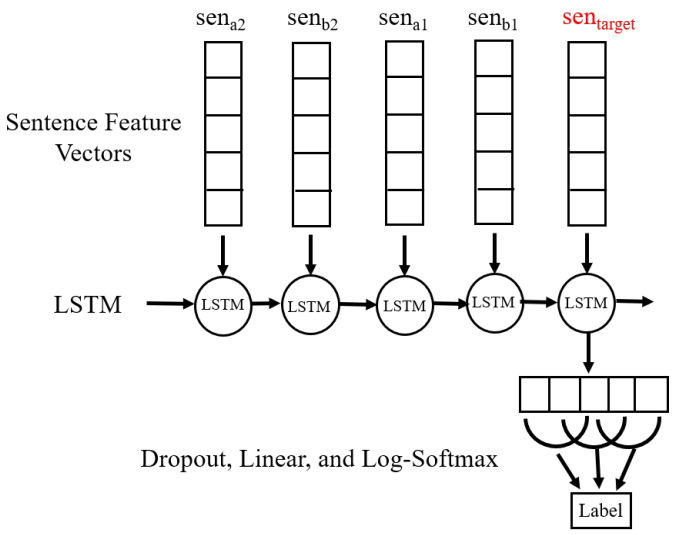


Figure 6.6: The LSTM classifies the sentence with the dependence information from neighbor sentences.

learning, we set the number of epochs [2] to 2000, which is empirically determined in our evaluation.

Sentence feature vector extraction. A CNN and a maxpool layer (i.e., an output layer in a neural network [150]) are used to extract the sentence feature vector from the pre-trained word vectors. Fig. 6.5 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 [16]. 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 Fig. 6.5 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".

Sentece dependence modeling. ReCDroid+ uses an LSTM to model sentence dependence, as shown in Fig. 6.6. 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

[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 [149]

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. [151]. 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 $\{\text{sen}_{b_2}, \text{sen}_{b_1}, \text{sen}_{target}, \text{sen}_{a_1}, \text{sen}_{a_2}\}$, 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) [152], The well-known dropout method is used to prevent overfitting of the LSTM model [153]. 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).

6.3.1.3 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 a 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 6.1 lists the eleven rules, their description, and the rationale behind each rule. In the equation representation of each rule, "cmt_i" suggests sentences in the i^{th} comment,"$M()$" suggests the S2R extraction deep learning model, "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,

Table 6.1: 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	$sentIndex, line_i =$ $\quad findFirst(\text{"toreproduce"}, line)$ $S2R_r = extafter(extra2Neib($ $\quad M(line_i), line_i)), sentIndex)$	(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.

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 limited 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. Each bug report in the dataset is recorded as whether it was successfully reproduced and 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 from the successful cases .

We use mean-shift [137], 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 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.

6.4 S2Rminer Evaluation

To evaluate S2Rminer, we consider two research questions:

RQ1: What is the performance of S2Rminer in extracting S2R from bug reports?

RQ2: Which types of text features has the best performance in extracting S2R from bug reports?

RQ1 lets us evaluate the effectiveness of S2Rminer in extracting S2R. RQ2 lets us investigate how different types of text features influence of the performance of S2Rminer.

6.4.1 Datasets

We evaluated S2Rminer on bug reports from GitHub [12] and Google Code [118]. To prepare the training set, we randomly crawled 500 bug reports from GitHub and 500 reports from Google Code. We hired two undergraduate students to label the sentences of each bug report as S2R and non-S2R. During the labeling process, the inspector read the reports with sufficient details in the bug descriptions to identify S2R sentences. To ensure the correctness of our results, the manual inspections were performed independently by the two undergraduate students. Any time there was dissension, the authors and the inspectors discussed to reach a consensus.

We randomly divided the 500 bug reports from both datasets into two sets — 400 for training 100 for testing. Each bug report contains one or more sentences, which are the instances for building machine learning models.

6.4.2 Experiment Design

The experiment was conducted on a physical x86 machine running with Ubuntu 14.04 installed. The NLP techniques of S2Rminer was implemented by the spaCy

Table 6.2: Types of Text Features

No NLP techniques	only use original words as features
Stem(1 gram)	only stem of the word as features
Stem(3 gram)	only stem of the word but consider 3 gram relationship
Stem(3 gram)+pos	combine stem and part of speech as features
Stem(3 gram)+dep	combine stem and dependency as features
Stem(3 gram)+pos+dep	combine three of them as features
(Stem+pos+dep)(3 gram)	add 3 gram relationship to all of features

dependency parser [143]. The classifier was implemented by Scikit-Learn [137].

6.4.2.1 Performance Metrics.

We chose performance metrics allowing us to answer each of our two research questions. Specifically, we employ accuracy, precision, recall, and F1-measure. A sentence can be classified as: S2R when it is truly a S2R sentence (true positive, TP); it can be classified as a S2R sentence when it is actually not (false positive, FP); it can be classified as a non-S2R sentence when it is actually a S2R sentence (false negative, FN); or it can be correctly classified as a non-S2R sentence (true negative, TN).

- **Accuracy:** the number of instances correctly classified over the total number of instances.

$$Accuracy = \frac{TP+TN}{TP+FP+FN+TN}$$

- **Precision:** the number of instances correctly classified as S2R over the number of all instances classified as S2R.

$$P = \frac{TP}{TP+FP}$$

- **Recall:** the number of instances correctly classified as S2R over the total number of S2R instances.

$$R = \frac{TP}{TP+FN}$$

- **F-measure:** a composite measure of precision and recall for buggy instances.

$$F(b) = \frac{2*P*R}{P+R}$$

6.4.2.2 Combinations of Different Text Features.

RQ2 aims to evaluate how S2Rminer performs when using the combinations of different types of text features. Table 6.2 shows the features used for evaluation.

6.4.3 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subjects and bug reports. Other subjects may exhibit different behaviors. Data recorded in bug tracking systems can have a systematic bias relative to the full population of bug reports [154] and can be incomplete or incorrect [155]. However, we do reduce this threat to some extent by using two well studied open source projects and bug sources for our study. We cannot claim that our results can be generalized to all systems of all domains though.

The primary threat to internal validity involves the use of manual inspection to identify the S2R sentences To minimize the risk of incorrect results given by manual inspection, the sentences are labeled independently by two people.

The primary threat to construct validity involves the dataset and metrics used in the study. To mitigate this threat, we used bug reports from two bug tracking systems, which are publicly available and generally well understood. We also used the well known, accepted, and validated measures of accuracy, recall, precision, and F-measure.

6.5 S2Rminer Results and Analysis

Table 6.3 and Table 6.4 summarizes the results of the two datasets.

6.5.1 RQ1: Performance of S2Rminer.

Table 6.3 and Table 6.4 show that S2Rminer is able to extract S2R from bug reports in GitHub and Google Code. The accuracy is above 0.85 for GitHub and above 0.92 for Google code. Regarding the precision and recall, the best F-score is above 0.6 for both Github and Google code. We consider F-measures over 0.6 to be good [156].

In both bug tracking systems, the precision scores are better than the scores of recall. We analyzed the results and found that the low recall could be due to 1) the small training set; 2) incorrect labels. As part of the future, we intend to expand the training set and perform more robust labeling work.

In summary, the above results imply that *S2Rminer is effective at extracting S2R.*

6.5.2 RQ2: Comparison of Different Types of Text Features.

As we can see from the two tables, comparing the text feature without using NLP technique (F_n) with the other feature combinations in the GitHub dataset, F_n performed the worst. However, this is not true in the Google Code dataset. When comparing different feature combinations with NLP applied, the dependency parsing feature type slightly improved the performance in terms of F-measures.

Overall, these results imply that the *the stemming and dependency parsing feature types can potentially improve the performance of S2Rminer.*

Table 6.3: GitHub Result

GitHub Result	TP	TN	FP	FN	Accuracy	Precision	Recall	F-score
No NLP techniques	79	612	33	84	0.85	0.69	0.47	0.56
Stem(1 gram)	99	596	49	69	0.86	0.66	0.62	0.64
Stem(3 gram)	88	612	33	71	0.87	0.72	0.55	0.63
Stem(3 gram)+pos	94	605	40	65	0.86	0.70	0.59	0.64
Stem(3 gram)+dep	94	609	36	65	0.87	0.72	0.59	0.65
Stem(3 gram)+pos+dep	90	607	38	69	0.86	0.70	0.57	0.63
(Stem+pos+dep)(3 gram)	87	605	40	72	0.86	0.69	0.55	0.61

Table 6.4: Google Code Result

Google Result	TP	TN	FP	FN	Accuracy	Precision	Recall	F-score
No NLP techniques	40	516	13	32	0.92	0.75	0.56	0.64
Stem(1 gram)	44	507	22	28	0.92	0.67	0.61	0.64
Stem(3 gram)	38	519	10	34	0.93	0.79	0.53	0.63
Stem(3 gram)+pos	36	518	11	36	0.92	0.77	0.5	0.61
Stem(3 gram)+dep	39	520	9	33	0.93	0.81	0.54	0.65
Stem(3 gram)+pos+dep	40	517	12	32	0.93	0.77	0.56	0.65
(Stem+pos+dep)(3 gram)	36	518	11	36	0.92	0.76	0.5	0.61

6.6 ReCDroid+ Evaluation

To evaluate ReCDroid+, we consider four 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?

6.6.1 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 [82] and 25 bug reports from a recent paper on translating Android bug reports into test cases [83]. 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 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 a S2R or not. To ensure the correctness of our results, the manual labeling process was performed independently by two graduate students. 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.

6.6.2 Implementation

We conducted our experiment on a physical x86 machine running with Ubuntu 16.04. The NLP techniques of ReCDroid+ was implemented based on the spaCy dependency parser [121]. The dynamic exploration component was implemented on top of two execution engines, Robotium [55] and UI Automator [112], 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.

6.6.3 Experiment Design

6.6.3.1 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., 3 hours) and efficiency in terms of the time it took to reproduce each crash.

6.6.3.2 RQ2: Effectiveness and Efficiency of ReCDroid+ in extracting S2R and crash sentences

We performed a five cross-validation on the 4000 labeled bug reports. Of these 5 folds, 4 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 [42]. 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 6.3.1.3. We showed the results on reproducing success and time on each single refining rule.

6.6.3.3 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 6.8). The time limits for running ReCDroid+$_N$ and ReCDroid+$_D$ were also set to three hours.

6.6.3.4 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 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 54 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 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? Note that to avoid bias, the participants were not aware of the purpose of this user study.

6.6.3.5 RQ5: Handling Low-Quality Bug Reports

The goal of RQ5 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. 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.

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.

6.6.3.6 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 5.2 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.

6.7 ReCDroid+ Results and Analysis

The results of applying ReCDroid+, ReCDroid+$_N$, and ReCDroid+$_D$ in 54 out of the 66 bug reports are summarized in Table 6.5. 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., [130]). Columns 2 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.

6.7.1 RQ1: Effectiveness and Efficiency of ReCDroid+

As Table 6.5 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 $\star$) from the FUSION paper [82] and nine bug reports (marked with $*$) from Yakusu [83] were successfully reproduced. The results indicate that *ReCDroid+ is effective in*

Table 6.5: 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+. "✓"=Crash reproduced. "N"=Crash not reproduced. "-"=Not applicable. ">"=exceeded time limit (3 hours).

reproducing crashes from bug reports. The 12 cases where ReCDroid+ failed will be discussed in Section 6.8.

For the six 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). Third, ReCDroid+ cannot handle time-sensitive crashes. To trigger the crash for ODK-1796 (a concurrency bug), requires waiting on one screen for seconds and then clicking the next screen at a very fast speed. The limitations are summarized in Section 6.8.

6.7.2 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 6.6 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 6.6 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

Table 6.6: RQ3: Different policies

#BR.	Time (sec)										
	P1	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*
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	-

"P1"=Policy 1. "N"=Crash not reproduced. "-"=Not applicable(empty extraction under the policy). ">"=Crash not reproduced in exceeded time limit (3 hours).

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 that is on another. For example, rule-7 cost less time on timeTrack-35 than rule-1, but the situation is opposite on ODK-2601.

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 training set and testing set. In each iteration, we randomly selected 32 bug reports as training set and the other 10 bug reports as 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 [157] to measure the significant difference between the average time cost of mean-shift and the random. The results that, in average, mean-shift took 854 seconds to reproduce a crash, which was almost half of time taken by random (i.e., 1594 seconds), and the difference is statistically significant.

6.7.3 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 2). 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.

6.7.4 RQ4: Usefulness of ReCDroid+

The last column of Table 6.5 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 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 2 and 9 in Table 6.5 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 [131] 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.*

Table 6.7: 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)

6.7.4.1 RQ5: Handling Low-Quality Bug Reports

Columns 4–9 of Table 6.7 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.*

6.7.4.2 RQ6: Handling Bug Reports Generated by Different Reporters

The last two columns of Table 6.7 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.*

6.8 Discussion

Comparing with Android Testing Tools. As a generic GUI exploration and testing tool, ReCDroid+$_D$ is similar to existing Android testing tools [63, 5, 135, 69, 9, 7], which detect crashes in an unguided manner. ReCDroid+$_D$ was shown to be competitive with Monkey [5], Sapienz [63], and the recent work Stoat [69] on our experiment subjects. Sapienz can not work on the app which android sdk version is not Android 4.4, so ReCDroid+ only compare the android 4.4 app in our data set with Sapienz. Specifically, as Table 6.5 shows, ReCDroid+$_D$ reproduced 31 more crashes than Stoat, 20 more crashes than Sapienz, and 28 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.

6.9 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.

Chapter 7

Conclusion and Future Work

In this dissertation, we have presented AI-based mobile testing and bug reproducing technologies that allow testers to automatically test mobile app and reproduce mobile bugs from bug reports. We have introduced DinoDroid, a mobile testing tool that utilizes AI technologies to learn to understand the text content of the android app. It can also know how to test android applications rather than depending on heuristic rules. We have evaluated DinoDroid on 64 apps from a widely used benchmark and showed that DinoDroid outperforms the state-of-the-art and state-of-practice Android GUI testing tools in both code coverage and bug detection. We have developed another tool ReCDroid+ that is an automated approach to reproducing crashes from bug reports for Android applications. ReCDroid+ leverages AI technology natural language processing techniques and heuristics to analyze bug reports and identify GUI events that are necessary for crash reproduction. ReCDroid+ can successfully reproduce 42 bugs in 66 raw bug reports.

For future work, we intend to improve both of our existing approaches DinoDroid and ReCDroid+. Given a test case as input, the testing tool will run the tested software. It should have the ability to distinguish whether the running behavior of the tested software is normal or not. Otherwise, a human should be involved in the process to judge the behavior manually. The automatic judgment is called "Oracle". Yet, automatic Oracle generation is still an open question in software testing. If it is still an open question, the software testing can not be a whole automatic process [158]. In the current version DinoDroid and ReCDroid+, we evade the oracle generation by only focusing on the crash problem. The crash is a kind of Oracle which can be easily detected. I am planning to detect the bug report's Oracle type by a machine learning method or key work matching. If a sentence is identified as an Oracle sentence in a bug report, this sentence will be processed by an Nlp technology part. The Nlp technology may extract the Oracle information from a bug report into an Oracle pattern. The Oracle pattern will be transferred to the Oracle in the test case. The steps above combine an automatic Oracle generation method.

Bibliography

[1] Librenews-android issues. https://github.com/milesmcc/LibreNews-Android/issues.

[2] Android activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.

[3] Google play data. `https://en.wikipedia.org/wiki/Google_Play`.

[4] Hewlett Packard. Failing to meet mobile app user expectations: A mobile user survey. *Tech. rep.*, 2015.

[5] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html.

[6] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[7] 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 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[8] 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.

[9] 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. ACM, 2013.

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

[11] gooledoc. `https://code.google.com`.

[12] github. `https://github.com`.

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

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

[15] lxml.etree. https://lxml.de/tutorial.html, 2014.

[16] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

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

[18] Dinodroid. https://github.com/softwareTesting123/DinoDroid.

[19] Recdroid. `https://github.com/AndroidTestBugReport/ReCDroid`.

[20] Rémi Philippe Lebret. Word embeddings for natural language processing. Technical report, EPFL, 2016.

[21] Word Embedding. `http://semanticgeek.com/technical/a-count-based-and-predictive-vector-models-in-the-semantic-age/`, 2020.

[22] word2vec. https://github.com/dav/word2vec.

[23] William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, volume 161175. Citeseer, 1994.

[24] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. Text classification algorithms: A survey. *Information*, 10(4):150, 2019.

[25] Ralf Bender and Ulrich Grouven. Ordinal logistic regression in medical research. *Journal of the Royal College of physicians of London*, 31(5):546, 1997.

[26] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[27] Larry M Manevitz and Malik Yousef. One-class svms for document classification. *Journal of machine Learning research*, 2(Dec):139–154, 2001.

[28] Andreas Kamilaris and Francesc X Prenafeta-Boldú. Deep learning in agriculture: A survey. *Computers and electronics in agriculture*, 147:70–90, 2018.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[30] dependency-parse. https://spacy.io/usage/linguistic-features.

[31] Angelina Ivanova, Stephan Oepen, and Lilja Øvrelid. Survey on parsing three dependency representations for english. In *51st Annual Meeting of the Association for Computational Linguistics Proceedings of the Student Research Workshop*, pages 31–37, 2013.

[32] Yuan Ding and Martha Palmer. Machine translation using probabilistic synchronous dependency insertion grammars. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 541–548, 2005.

[33] Akane Yakushiji, Yusuke Miyao, Tomoko Ohta, Yuka Tateisi, and Jun'ichi Tsujii. Automatic construction of predicate-argument structure patterns for biomedical information extraction. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 284–292, 2006.

[34] Lilja Øvrelid, Jonas Kuhn, and Kathrin Spreyer. Cross-framework parser stacking for data-driven dependency parsing. *TAL*, 50(3):109–138, 2009.

[35] PJ Antony and KP Soman. Parts of speech tagging for indian languages: a literature survey. *International Journal of Computer Applications*, 34(8):0975–8887, 2011.

[36] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[37] Richard Bellman. On the theory of dynamic programming. *National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[38] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 153–164, 2020.

[39] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[40] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[41] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. Improving bug tracking systems. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 247–250. IEEE, 2009.

[42] 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 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407. ACM, 2017.

[43] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. Why are some bugs non-reproducible?

[44] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 62–71. ACM, 2014.

[45] App manifest. https://developer.android.com/guide/topics/manifest/manifest-intro.

[46] Application fundamentals. https://developer.android.com/guide/topics/manifest/manifest-intro.

[47] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 233–244, 2017.

[48] Android broadcasts. https://developer.android.com/guide/components/broadcasts.

[49] Android emulator. https://developer.android.com/studio/run/emulator.

[50] Android debug bridge. https://developer.android.com/studio/command-line/adb.

[51] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[52] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Sciences and Applications*, 3(6):12–1, 2012.

[53] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 29–35. IEEE Press, 2012.

[54] Mirzaei Alvari and Nariman Mirzaei. *Automated Input Generation Techniques for Testing Android Applications*. PhD thesis, 2016.

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

[56] uiautomator. `https://developer.android.com/training/testing/ui-automator.html`.

[57] uiautomatorview. `https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html`.

[58] uiautomatorshortcoming. `https://developer.android.com/reference/android/support/test/uiautomator/UiDevice.html`.

[59] robotium. `https://plugins.jetbrains.com/plugin/7513-robotium-recorder`.

[60] monkeyrunner. `https://developer.android.com/studio/test/monkeyrunner/index.html`.

[61] appiumr. `https://github.com/appium/appium`.

[62] espresso. `https://developer.android.com/training/testing/espresso/index.html`.

[63] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[64] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the International Conference on Automated Software Engineering*, pages 238–249, 2016.

[65] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 103–114, 2017.

[66] Jiwei Yan, Linjie Pan, Yaqi Li, Jun Yan, and Jian Zhang. Land: a user-friendly and customizable test generation tool for android apps. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 360–363, 2018.

[67] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. Widget-sensitive and back-stack-aware gui exploration for testing android apps. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 42–53, 2017.

[68] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *FASE*, volume 13, pages 250–265. Springer, 2013.

[69] 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 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

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

[71] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. Android testing via synthetic symbolic execution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 419–429, 2018.

[72] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. QBE: Qlearning-based exploration of android applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115, 2018.

[73] Nataniel P Borges, Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 133–143. IEEE, 2018.

[74] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: a deep learning-based approach to automated black-box android app testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 1070–1073, 2019.

[75] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. Test transfer across mobile apps through semantic mapping. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 42–53, 2019.

[76] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. Learning user interface element interactions. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 296–306, 2019.

[77] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. Reinforcement learning for android gui testing. In *Proceedings of the International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 2–8, 2018.

[78] Thi Anh Tuyet Vuong and Shingo Takada. Semantic analysis for deep q-network in android gui testing. In *SEKE*, pages 123–170, 2019.

[79] Thi Anh Tuyet Vuong and Shingo Takada. A reinforcement learning based approach to automated testing of android applications. In *Proceedings of the International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 31–37, 2018.

[80] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012.

[81] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 772–784, 2019.

[82] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 673–686. ACM, 2015.

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

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

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

[86] Xue Han, Tingting Yu, and David Lo. Learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd International Conference on Automated Software Engineering*, ICSE '17, 2018.

[87] 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.

[88] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 620–631. IEEE, 2015.

[89] Dongpu Jin, Myra B Cohen, Xiao Qu, and Brian Robinson. Preffinder: getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 151–162. ACM, 2014.

[90] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 86–96, 2019.

[91] Younes Bennani and Khalid Benabdeslem. Dendogram-based svm for multi-class classification. *Journal of Computing and Information Technology*, 14(4):283–289, 2006.

[92] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, pages 155–166, 2010.

[93] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.

[94] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484, 2012.

[95] 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 2015 IEEE 23rd International Conference on Program Comprehension*, pages 48–59, 2015.

[96] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *ICSE*, pages 102–112, 2012.

[97] Tingting Yu, Tarannum S Zaman, and Chao Wang. Descry: reproducing system-level concurrency failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 694–704. ACM, 2017.

[98] 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.

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

[100] M Mainegra Hing, Aart van Harten, and PC Schuur. Reinforcement learning versus heuristics for order acceptance on a single resource. *Journal of Heuristics*, 13(2):167–187, 2007.

[101] lockpatterngenerator. https://github.com/dharmik/lockpatterngenerator.

[102] 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.

[103] Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260, 2003.

[104] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. *ACM SIGPLAN Notices*, 49(10):33–47, 2014.

[105] Android activity. https://developer.android.com/reference/android/app/Activity.

[106] Conv1d layer. https://keras.io/api/layers/convolution_layers/convolution1d.

[107] Dense layer. https://keras.io/api/layers/core_layers/dense/.

[108] adam. https://keras.io/api/optimizers/adam/.

[109] Arryon D Tijsma, Madalina M Drugan, and Marco A Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *Proceedings of the Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2016.

[110] Michel Tokic and Günther Palm. Value-difference based exploration: adaptive control between epsilon-greedy and softmax. In *Proceedings of the Annual Conference on Artificial Intelligence*, pages 335–346, 2011.

[111] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[112] Ui automator. https://developer.android.com/training/testing/ui-automator.

[113] Androguard. https://github.com/androguard/androguard.

[114] Emma. http://emma.sourceforge.net/.

[115] Keras. https://keras.io/.

[116] nectroid. https://github.com/cknave/nectroid.

[117] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. Recdroid: Automatically reproducing android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE -19, pages 128–139, 2019.

[118] Google code archive. https://code.google.com/archive/.

[119] 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.

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

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

[122] 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 12th annual international conference on Mobile systems, applications, and services*, pages 204–217, 2014.

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

[124] 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.

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

[126] 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.

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

[128] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 253–262. IEEE, 2011.

[129] B Ashok, Joseph Joy, Hongkang Liang, Sriram K Rajamani, Gopal Srinivasa, and Vipindeep Vangala. Debugadvisor: a recommender system for debugging. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 373–382, 2009.

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

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

[132] 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.

[133] 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 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 408–419, 2018.

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

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

[136] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 666–676, 2015.

[137] 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. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[138] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[139] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[140] Charu C Aggarwal and ChengXiang Zhai. A survey of text classification algorithms. In *Mining text data*, pages 163–222. Springer, 2012.

[141] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. Automatically extracting bug reproducing steps from android bug reports. In *International Conference on Software and Systems Reuse*, pages 100–111. Springer, 2019.

[142] Jill Burstein, Daniel Marcu, Slava Andreyev, and Martin Chodorow. Towards automatic classification of discourse elements in essays. In *Proceedings of the 39th annual Meeting on Association for Computational Linguistics*, pages 98–105. Association for Computational Linguistics, 2001.

[143] spacy. https://spacy.io/.

[144] Akshay Kulkarni and Adarsha Shivananda. Converting text to features. In *Natural Language Processing Recipes*, pages 67–96. Springer, 2019.

[145] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer, 1998.

[146] acv-11. https://github.com/robotmedia/droid-comic-viewer/issues/11, 2013.

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

[148] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. A c-lstm neural network for text classification. *arXiv preprint arXiv:1511.08630*, 2015.

[149] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research*, 14(Feb):567–599, 2013.

[150] maxpool. https://keras.io/api/layers/pooling_layers/, 2020.

[151] Jos Van Der Westhuizen and Joan Lasenby. The unreasonable effectiveness of the forget gate. *arXiv preprint arXiv:1804.04849*, 2018.

[152] Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *Transactions of the Association for Computational Linguistics*, 4:357–370, 2016.

[153] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[154] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *ESEC*, pages 121–130, 2009.

[155] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE*, pages 298–308, 2009.

[156] Geoff Dougherty. *Pattern recognition and classification: an introduction*. Springer Science & Business Media, 2012.

[157] Nadim Nachar et al. 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):13–20, 2008.

[158] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

VITA

Yu Zhao started his Ph.D. with the Department of Computer Science at the University of Kentucky in 2016. His current research mainly focuses on software testing and computer networks. Yu Zhao received his Bachelor's degree in Telecom Engineering from Jilin University in Changchun, China in 2009 and respectively his M.S. Degree in Computer Science from Changchun University of Science and Technology in Telecom Engineering, China in 2012.