CrashTranslator: Automatically Reproducing Mobile Application Crashes Directly from Stack Trace

Yuchao Huang[*†‡]
yuchao2019@iscas.ac.cn
Institute of Software, Chinese
Academy of Sciences
Beijing, China

Junjie Wang[*†‡§]
junjie@iscas.ac.cn
Institute of Software, Chinese
Academy of Sciences
Beijing, China

Zhe Liu[*†‡]
liuzhe181@mails.ucas.edu.cn
Institute of Software, Chinese
Academy of Sciences
Beijing, China

Yawen Wang[*†‡]
yawen2018@iscas.ac.cn
Institute of Software, Chinese
Academy of Sciences
Beijing, China

Song Wang
wangsong@yorku.ca
York University
Toronto, Canada

Chunyang Chen
Chunyang.chen@monash.edu
Monash University
Melbourne, Australia

Yuanzhe Hu[*†‡]
yuanzhe@iscas.ac.cn
Institute of Software, Chinese
Academy of Sciences
Beijing, China

Qing Wang[*†‡§]
wq@iscas.ac.cn
Institute of Software, Chinese
Academy of Sciences
Beijing, China

ABSTRACT

Crash reports are vital for software maintenance since they allow the developers to be informed of the problems encountered in the mobile application. Before fixing, developers need to reproduce the crash, which is an extremely time-consuming and tedious task. Existing studies conducted the automatic crash reproduction with the natural language described reproducing steps. Yet we find a non-neglectable portion of crash reports only contain the stack trace when the crash occurs. Such stack-trace-only crashes merely reveal the last GUI page when the crash occurs, and lack step-by-step guidance. Developers tend to spend more effort in understanding the problem and reproducing the crash, and existing techniques cannot work on this, thus calling for a greater need for automatic support. This paper proposes an approach named CrashTranslator to automatically reproduce mobile application crashes directly from the stack trace. It accomplishes this by leveraging a pre-trained Large Language Model to predict the exploration steps for triggering the crash, and designing a reinforcement learning based technique to mitigate the inaccurate prediction and guide the search holistically. We evaluate CrashTranslator on 75 crash reports involving 58 popular Android apps, and it successfully reproduces 61.3% of the crashes, outperforming the state-of-the-art baselines by 109% to 206%. Besides, the average reproducing time is 68.7 seconds, outperforming the baselines by 302% to 1611%. We also evaluate the usefulness of CrashTranslator with promising results.

CCS CONCEPTS

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

KEYWORDS

Bug reproduction, Stack trace, Mobile application testing

[*]Also With State Key Laboratory of Intelligent Game, Institute of Software, CAS
[†]Also With Science and Technology on Integrated Information System Laboratory, Institute of Software, CAS
[‡]Also With University of Chinese Academy of Sciences
[§]Corresponding author

1 INTRODUCTION

New mobile applications are being developed and released continuously via app stores since the market for mobile devices is both growing and diversifying. Recent statistics show that more than 5 million mobile apps are available in popular marketplaces like Apple App Store and Google Play Store, for over 140 billion downloads in 2022 and 12 million mobile developers are maintaining them [55]. 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 quite important for an app's success. As shown in a survey, 88% of app users would abandon an app if they repeatedly

encountered a functionality issue [2]. This motivates developers to identify and resolve issues rapidly, or risk losing users otherwise.

An important mechanism for ensuring app quality is the online bug reporting systems, e.g., GitHub Issue Tracker [21], Bugzilla [29], Google Code Issue Tracker [9]. These systems enable users to create bug reports in which they can describe their observed failure; developers can then use this information to help debug their apps. These bug reports are becoming a non-neglectable source of information for improving app quality and user satisfaction.

Once developers receive a bug report, one of the first steps to debugging the reported issue is to reproduce the issue following the reproducing steps. There are several existing studies focusing on crash[1] reproduction from the natural language described reproducing steps [72–74]. They typically apply natural language processing techniques to match the reproducing steps with the app's GUI events (i.e., operations on GUI widgets, e.g., clicking the *search* button of an app), and employ guided exploration strategies with the matched information for bug reproduction. However, not all crash submitters would strictly follow the report template to provide the reproducing steps when reporting the crash.

Our motivational study (Section 2) reveals that a non-neglectable portion (20.2%) of crash reports contain only the stack trace. Such stack-trace-only reports can be submitted by crash reporting tools such as Crashlytics [10], which automatically collects crash logs and upload them, and this is also the commonly-used practice in large commercial software. These reports can also be submitted by the app users who accidentally trigger the crash yet fail to figure out the reproducing steps. Due to the insufficient information provided in these reports, developers tend to spend extra effort in understanding and reproducing the issues, which brings in a longer fixing duration, i.e., the average fixing duration of these stack-trace-only reports is 26% larger than the crash reports with reproducing steps. Besides, the aforementioned existing approaches would fail to work on stack-trace-only reports. This further implies the necessity for the automatic crash reproduction approach directly from stack traces.

Existing studies on stack trace analysis relate with stack trace similarity [30, 49, 60], fault localization from stack trace [23, 26, 42, 65, 67], test code generation from stack trace [7, 50, 52–54, 68], duplicate crash reports detection with stack trace [11, 48], etc. Although these approaches can facilitate the understanding and analysis of the stack trace, none of them can tackle the problem of automatic crash reproduction from the stack trace. There are two challenges in the automatic reproduction of these crashes.

First, as mentioned above, existing studies generally utilize the textual-described reproducing steps for crash reproduction, yet stack-trace-only crashes lack step-by-step guidance. In other words, the stack trace does not record the exploration sequence from the entry page to the crash-occurring page, while the most useful information might be the last GUI page when the crash occurs. However, there can be 1 to 8 exploration steps for reaching the last crash-occurring GUI page (based on our experimental data), which can be quite inefficient if explored randomly. Although the static or dynamic analysis techniques [19, 56, 70] can infer the transition between activities and plan the exploration path, yet they can be

quite incomplete or inaccurate [69]. Second, even when the last crash-occurring page is reached, it may still need certain interactions with the app to finally trigger the crash, e.g., clicking a certain button. Nevertheless, there can be an average of 6.6 interactive widgets on a GUI page (based on our experimental data) and the stack trace does not implicitly provide which widget to interact with for crash triggering, which further complicates the automated crash reproduction problem.

Nevertheless, we also find two clues for facilitating the reproduction of stack-trace-only crashes. The first is the last crash-occurring GUI page before triggering the crash, which offers the target for the planned exploration. The second is the involved APIs in the programming code when conducting the operations in the crash-occurring page, which can help find the widget with which to interact so the crash can finally be triggered.

Motivated by these clues, we propose an approach named Crash-Translator to automatically reproduce mobile application crashes directly from the stack trace. It accomplishes this by leveraging a pre-trained Large Language Model (LLM) to predict the exploration steps for triggering the crash, and designing a reinforcement learning based technique to mitigate the inaccurate prediction and guide the search holistically.

In detail, we first extract crash-related information from the stack trace, i.e., the crash-occurring GUI page and crash-involved APIs. Second, we design three scorers to assign the exploration priority for each GUI widget on the current page: 1) Page reaching scorer, which leverages the LLM to choose the GUI widget that may lead to the crash-occurring page; 2) Widget hitting scorer, which utilizes the heuristic method to find the crash-triggering widget by matching the crash-involved APIs; 3) Exploration optimization scorer, which assigns scores based on previous interaction records of reinforcement learning technique, in order to bridge the gap by inaccurate prediction of the first two scorers and plan the exploration holistically. CrashTranslator selects the GUI widget based on the three scorers and continues the process iteratively until the target crash is triggered. It finally generates the replay script (for direct replay) and the textual-described reproducing steps with the step-by-step image instructions (for facilitating understanding).

To evaluate the effectiveness and efficiency of our approach, we run CrashTranslator on 75 crash reports collected from 58 popular Android apps involving three datasets. CrashTranslator successfully reproduces 61.3% (46/75) of the crashes, which outperforms the state-of-the-art baselines by 109% to 206%. Besides, the average reproducing time is 68.7 seconds, outperforming the baselines by 302% to 1611%. Furthermore, the results also show that both the designed page reaching scorer and widget hitting scorer greatly contribute to the reproduction performance. The usefulness evaluation shows that CrashTranslator's generated reproducing steps can make the crashes easily reproduced (215% faster).

The contributions of this paper are as follows:

- **Dimension.** The first work of the automatic crash reproduction of mobile applications directly from the stack trace.
- **Technique.** An automatic approach CrashTranslator for stack-trace-only crash reproduction by leveraging the LLM to predict the exploration steps for triggering the crash, and

---

[1]Existing studies focus on the crash reports since they have the observable oracle compared with other bugs, and following them, this work also focuses on the crash reproduction, and we will use crash and bug interchangeably.

designing a reinforcement learning based technique to mitigate the inaccurate prediction and guide the search holistically.

- **Evaluation.** Experimental evaluation of effectiveness, efficiency and usefulness of CrashTranslator with promising performance, outperforming the state-of-the-art techniques and human reproduction.
- **Data.** Public released source code of CrashTranslator and the dataset of our experiments to facilitate the replication and extension of this study[2].

2 MOTIVATIONAL STUDY

We conduct a motivational study to investigate whether it is common for stack-trace-only crash reports, their characteristics, and the challenges of reproducing these reports.

In detail, we choose GitHub as the data source since it contains a large number of publicly available valid bug reports. We use the web crawler provided by Wendland et al. [63] to automatically crawl the bug reports from Android projects and focus on the reports created from Jan. 2015 to May. 2022, resulting in 96,451 bug reports. We then filter the bug reports involving application crashes with the keywords such as *crash, exception* following existing studies [63, 74]. As a result, we acquire 10,843 Android crash reports for our motivational study.

2.1 Is it Common for Stack-trace-only Crash?

A well-formulated crash report usually contains the crash overview, textual-described reproducing steps, stack trace when the crash occurs, and visual recordings (screenshots/GIFs) about how the crash occurs. However, not all issue submitters would strictly follow the report template and provide all crash-related information when reporting the crash.

We utilize keywords and heuristic pattern matching[2] to automatically examine whether the crash report contains reproducing steps and stack traces, following existing studies [63, 74]. Results reveal that 20.2% (2,187 / 10,843) of crash reports contain only stack traces, i.e., stack-trace-only crash reports. Such reports can be submitted by crash reporting tools such as Crashtics [10], which automatically collect crash logs and upload them to the issue server when the app crashes, as shown in Figure 2 (a). Furthermore, commercial software can also have such auto-generated stack-trace-only crash reports [14, 27] , which further indicates the universality of such reports. Besides, these reports can also be submitted by the app users or developers who accidentally trigger the crash yet fail to figure out the reproducing steps, as shown in Figure 2 (b). This also implies the necessity for the automatic crash reproduction approach.

2.2 Is it Difficult to Handle Such Crash?

We go a step further to investigate whether it is difficult to handle (e.g., reproduce, fix) such crashes. Since it can hardly obtain the time or effort for crash reproducing from GitHub, we turn to the commonly-used issue fixing duration, i.e., the duration between the issue creation time and the closing time [12], to indicate the difficulty of handling such crashes.

For crash reports with reproducing steps, the average issue fixing duration is 57 days, while for stack-trace-only crash reports, such duration is increased to 72 days (26% higher). We assume that due to the insufficient information provided in the stack-trace-only crash report, developers need to spend extra effort in understanding the problem and reproducing the issue, and thus have a low willingness and take more time to fix the issue. Therefore it would be highly expected to automate the reproduction of stack-trace-only crash reports to save the effort and facilitate follow-up issue fixing.

2.3 Why is it Difficult?

Challenge 1: Lack of step-by-step guidance. Existing approaches would take the textual-described reproducing steps as input and conduct the bug reproduction guided by the steps [72–74]. However, for the stack-trace-only crash report, we cannot fetch the reproducing steps and thus lacks the step-by-step guidance to trigger the crash. By comparison, we can possibly derive the crash-occurring activity or fragment, i.e., the last interactive GUI page when the crash occurs, e.g., *InstalledSearchEnginesSettingsFragment* as demonstrated in Figure 1. But the automatic approach still needs to speculate the exploration steps for navigating to the crash-occurring page. In our experimental dataset (shown in Section 4.1), there can be 1 to 8 exploration steps for reaching the crash-occurring GUI page, which can be quite inefficient if explored randomly.

Challenge 2: Need specific interactions even reaching the last GUI page. The second challenge is that even if the crash-occurring GUI page is reached, it may still need certain interactions with the app to finally trigger the crash. As shown in Figure 1, to trigger the crash, after reaching the GUI page, one still needs to click *Search engine* in the crash-occurring GUI page at step 5. In our experimental dataset (shown in Section 4.1), there can be an average of 6.6 interactive widgets, which further complicates the reproduction of stack-trace-only crashes.

2.4 Are There any Clues for Reproduction?

While facing the above challenges, stack traces do provide clues for automated reproduction. Specifically, as mentioned in challenge 1, we can derive the crash-occurring GUI page, which offers the target for the planned exploration. Besides, when conducting the operations in the crash-occurring GUI page, there can be corresponding invocations of the programming code, and the stack trace would output the involved APIs, e.g., *refetchSearchEngines* and *onResume* in Figure 1.

To summarize, the stack trace offers two clues, i.e., the crash-occurring GUI page and the involved APIs when interacting with the app at the crash-occurring page. We can utilize the first clue for predicting the exploration steps for navigating to the crash-occurring GUI page; and then use the second clue to find the widget by interacting with which the crash can finally be triggered.
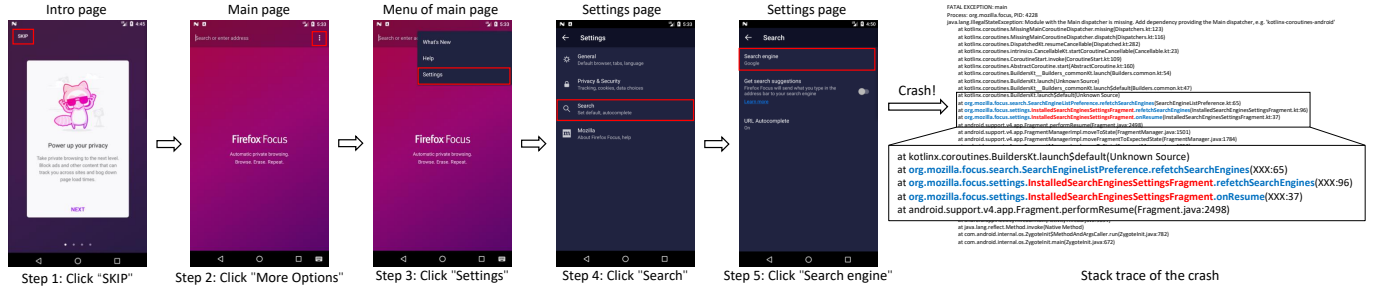
---

[2]Details are in our website: https://github.com/wuchiuwong/CrashTranslator

Figure 1: Examples of crash reproducing

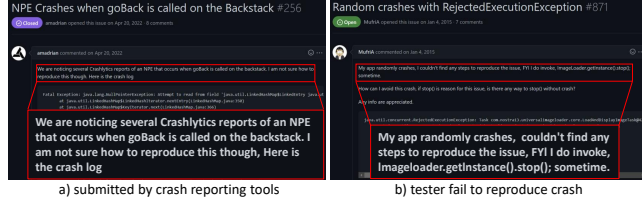a) submitted by crash reporting tools   b) tester fail to reproduce crash

Figure 2: Examples of stack-trace-only crash reports

Summary: Our analysis on 10,843 crash reports of Android apps from GitHub shows that 20.2% of crash reports only contain the stack trace, and these stack-trace-only reports consume 26% more time for issue fixing. This can be because they lack step-by-step guidance for planning the reproduction. Our findings confirm the necessity and challenges of the crash reproduction directly from the stack trace. We also observe two clues to motivate our approach development for automated crash reproduction.

3 APPROACH

Motivated by the above findings, we propose an automated approach named *CrashTranslator* to reproduce crash reports directly from the stack trace of mobile apps. It accomplishes this by leveraging a pre-trained Large Language Model (LLM) to predict the exploration steps for triggering the crash, and designing a reinforcement learning based technique to mitigate the inaccurate prediction and guide the search holistically.

As demonstrated in Figure 3, given a stack-trace-only crash report, CrashTranslator first derives the crash-occurring GUI page and crash-involved APIs for triggering the crash. It designs three scorers to assign the exploration priority for each interactable GUI widget on the current page, i.e., 1) **Page reaching scorer** leverages the LLM to choose the next GUI page for reaching the crash-occurring GUI page, and the widget for transferring to the next page (Section 3.2); 2) **Widget hitting scorer** utilizes a heuristic method to find the crash-triggering widget by matching the crash-involved APIs (Section 3.3); and 3) **Exploration optimization scorer** assigns scores based on the previous interaction records of reinforcement learning technique, which aims at bridging the gap by inaccurate prediction of the first two scorers and plans the exploration holistically (Section 3.4). CrashTranslator selects the GUI widget based on the three scorers and repeats the process iteratively until the target crash is triggered. It finally generates a replay script (for direct replay) and textual-described reproducing steps with step-by-step image instructions (for facilitating understanding).

3.1 Preprocessing

We first conduct preprocessing for the stack trace and the target app, to prepare the information for reproducing the crash. Specifically, three types of information will be used in the crash reproduction.

App's package name and the names of all activities in the app. We first decompile the target app and get its configuration file (*AndroidManifest.xml*) which records the package name of the app and the names of all activities it contains. In this paper, we consider the activity name as the GUI page name.

Crash-involved APIs. We extract the lines that contain the app's package name from the stack trace, which indicate the crash-involved APIs in the app, e.g., *org.mozilla.focus.search.SearchEngine-ListPreference.refetchSearchEngines* in blue color in Figure 1.

Crash-occurring page. From the crash-involved APIs extracted in the second step, we then check whether the terms coincide with the app's activity name (extracted in the first step). If so, we treat the activity as the crash-occurring page; otherwise, the crash may occur in a fragment, and we simply extract the name with the keywords *Fragment* from the stack trace, e.g., *InstalledSearchEngi-nesSettingsFragment* in red color in Figure 1.

For better understanding and reducing noise, we further tokenize the extracted page names (activity names), crash-involved APIs, and the crash-occurring page by the underscore and Camel Case, and remove the stop words, for follow-up usage.

3.2 Page Reaching Scorer

To reach the crash-occurring GUI page, intuitively, we can interact with the GUI widgets that share similar names with the crash-occurring page, e.g., click *Search* in step 4 can reach the crash-occurring *installed search engines settings fragment* as shown in Figure 1. However, there can be a long sequence of exploration steps for reaching the crash-occurring page, and the widgets in the prior and middle part of the sequence tend to be irrelevant to the crash-occurring page, e.g., *More options* in step 2 looks totally different from the crash-occurring page as shown in Figure 1. To tackle this, we leverage the LLM to predict the exploration steps for reaching the crash-occurring page iteratively. Specifically, as shown in Figure 4, we first ask the LLM to predict the next page for reaching the crash-occurring page; then ask the LLM to choose the widget for transferring to the predicted next page; and iterate the process. This step would assign scores for the widgets, indicating their probabilities of reaching the crash-occurring GUI page, i.e., priorities of being chosen.
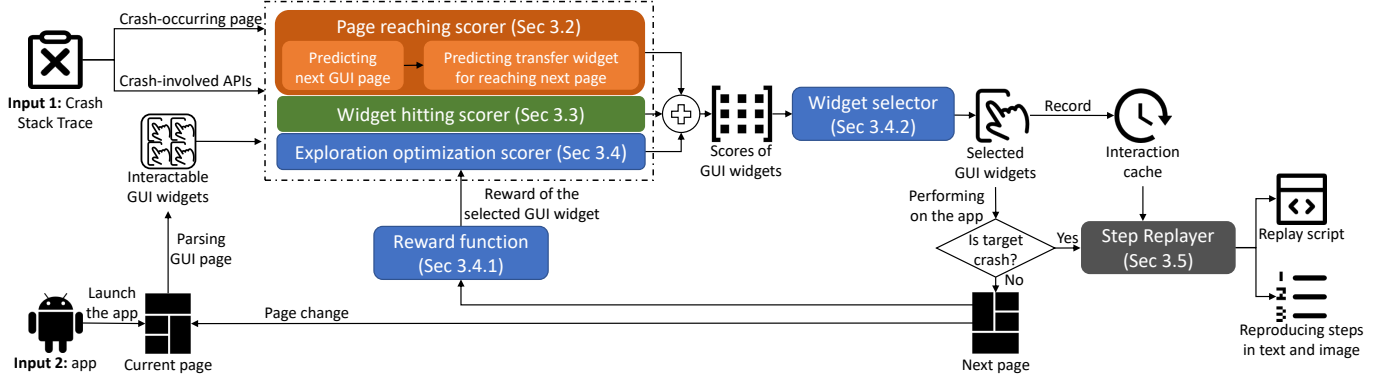
Figure 3: Overview of CrashTranslator. *Page reaching (Sec 3.2)* means arriving at the last GUI page when the crash occurs, and *widget hitting (Sec 3.3)* means clicking the correct widget in the last GUI page for triggering the crash.

Table 1: The example of prompt generation rules

		Input		
ID	**Attribute**	**Description**		**Examples**
I1	PageNames	List of names of all activities in the app, extracted from Android-Manifest.xml file		PageNames = ["intro", "main", "setting", …]
I2	CurrentPage	The activity name of the current page		CurrentPage = menu of main
I3	CrashPage	The name of the crash-occurring page, obtained from stack trace		CrashPage = installed search engines settings
I4	Interactable-Widgets	List of names for all interactable GUI widgets on the current page, obtained from parsing the view hierarchy of the current page		InteractableWidgets = ["what's new", "help", "settings", …]

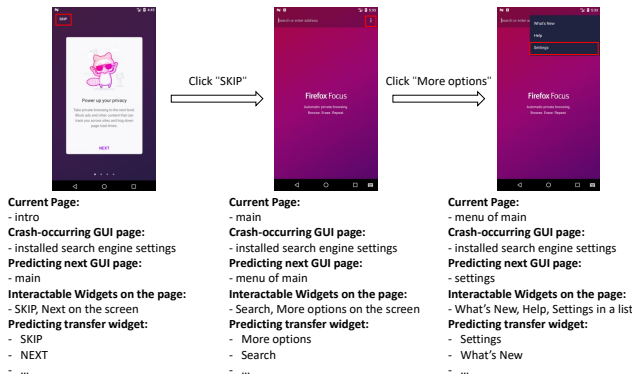|		Prompt		
ID	**Task**	**Prompt**	**Instantiation**	**Prediction**
P1	Predicting next GUI page	There are *<#NumOfPageNames>* pages in the app, named: *<PageNames>* . I want to go from the *<CurrentPage>* page to the *<CrashPage>* page. What is the next page?	There are 8 pages in the app, named: intro, main, setting, … . I want to go from the menu of main page to the installed search engines settings page. What is the next page?	*<NextPage>* = setting
P2	Predicting transfer widget for reaching next page	There are *<#NumOfPageNames>* pages in the app, named: *<PageNames>* . I want to go from the *<CurrentPage>* page to the *<CrashPage>* page. The next page may be the *<NextPage>* page. Here are widgets I can click:*<InteractableWidgets>*. What should I click?	There are 8 pages in the app, named: intro, main, setting, … . I want to go from the menu of main page to the installed search engines settings page. The next page may be the setting page. Here are widgets I can click: what's new, help, settings in a list, ... What should I click?	*<TransferWidget>* = settings

Figure 4: Example of navigating from the app entry page to the crash-occurring page

3.2.1 *Predicting Next Page for Reaching Crash-occurring Page.* We provide the LLM with all app's pages and the current

GUI page, and ask the LLM to speculate the next page to reach the crash-occurring page, by which the whole exploration sequence is generated iteratively.

Input. There are three inputs, i.e., I1-I3 as shown in Table 1. 1) PageNames, i.e., the name of all activities, are extracted from the *AndroidManifest.xml* as described in Section 3.1; 2) CurrentPage is the activity name of the current GUI page during the iterative process. To distinguish between pages with the same activity name but with different widgets on them (e.g., the pages in steps 2 and 3 in Figure 1), we divide the pages into three types: menu, dialog, and general pages. For menu or dialog page, we name the CurrentPage by "*page_type of activity_name*" (e.g., *menu of main* in step 3 in Figure 1). 3) CrashPage, i.e., the name of the crash-occurring page, is extracted from the crash stack trace as described in Section 3.1;

Prompt generation. To design the prompt, we follow the regular prompt template [5, 8, 25], and each of the three authors is asked to write the prompt sentence for this task with 10 trial apps, and conduct a discussion to derive the final prompt pattern, i.e., P1

as shown in Table 1. Take the last step in Figure 4 as an example, we first tell the LLM what pages are contained in the app (*There are 8 pages in the app, named: intro, main, setting, ...*), then tell the LLM what we want to do with the information about the current page and crash-occurring page (*I want to go from the menu of main page to the installed search engines settings page*), and finally ask what the next page should be (*What is the next page?*). The LLM will predict the name of the next page (*setting*).

Fine-tuning. To achieve better performance, we build a fine-tuning dataset and fine-tune the LLM for learning the transition relations between app pages. Specifically, we collect 1,000 apps of different categories from F-Droid [17] and extract their activity transition graph (ATG) with Gator [19], one of the state-of-the-art static analysis tools. We then utilize these transition relations as fine-tuning data for model fine-tuning. Although the static analysis tool like Gator cannot obtain the full ATG, which is why we do not directly use it for path planning, yet through inputting the incomplete ATG from different apps, the LLM has the potential to combine the diversified viewpoints together and speculate the desired transitions. Note that the fine-tuning process is a one-time requirement and does not necessitate individual fine-tuning for each app.

3.2.2 *Predicting Transfer Widget for Reaching Next Page*.
After knowing the next page, we need to know how to reach there, e.g., which button to click. We provide the LLM with all interactable GUI widgets on the current page, then ask the LLM to choose the widget for transferring to the next page.

Input. Besides the three inputs used in the previous section, this step needs the fourth input, i.e., InteractableWidgets (I4 as shown in Table 1). It is the list of names for all widgets which can be interacted with on the current GUI page, and we obtain it from the view hierarchy file of the current page. Specifically, we first filter the interactive widgets from all the widgets based on whether the *clickable* or *long-clickable* property is true. Then, we leverage heuristic rules to extract a representative name for each widget following existing studies [38, 39, 74]. In detail, for text-like widgets (e.g., *Buttons*, *TextView*, etc.), we obtain their textual attributes (e.g., *text*, *content-description*, *resource-id*) and utilize the first non-empty one. For icon-like widgets (e.g., *ImageButton*, *ImageView*, etc.), we extract their name from their contextual text information (e.g., *nearby text*, *sibling text*, *child text*) and use the first non-empty one. Finally, we tokenize the extracted widget name by the underscore and Camel Case, and group widgets according to their container widgets to express the current page's layout, e.g., *What's New, Help, Settings in a list* in step 3 of Figure 4.

Prompt generation. Following the same procedure as the previous section, we come out with the prompt pattern, i.e., P2 in Table 1. Take the last step in Figure 4 as an example. Like the prompt in the previous section, we first tell the LLM what pages are contained in the app and our ultimate goal; we then tell it the predicted next page (*The next page may be the setting page*), provide the names of all widgets on the current page (*Here are widgets I can click: what's new, help, settings in a list, ...*), and finally ask LLM to choose the optimal transferring widget (*What should I click?*). The LLM will output a widget which it thinks could lead to the target page (*settings*).

Since the LLM might not always output the correct transfer widget, we would let it provide a ranked list of candidate widgets (i.e., top 5), and CrashTranslator would consider these ranked widgets during the exploration optimization in Section 3.4. To realize this, we repeat the widget prediction by utilizing a new prompt in which we remove the interactable widgets that have already been predicted, and let the LLM provide a new answer. This process is repeated five times, and we obtain five distinct widgets in order. We then assign numerical scores to each widget in the manner of $1/(rank + 2)$ (e.g., Top 1 scores 0.33, Top 2 scores 0.25).

Fine-tuning. Similar to the prior section, we fine-tune the LLM for better performance. We use the commonly-used RICO dataset [13], which contains plenty of the targeted GUI page and transfer widget for reaching it. We randomly sample 1,000 such data pairs (involving 629 apps) as the fine-tuning dataset. Note that the fine-tuning process here also only needs to be conducted once.

3.3 Widget Hitting Scorer
Some crashes can be triggered as soon as arriving at the crash-occurring page, yet in most cases, specific events or event combinations are needed to perform on the crash-occurring page to finally trigger the crash, e.g., although we reach the *installed search engine settings* page at step 4 in Figure 1, the crash has not yet been triggered until we click the *Search engine* at step 5. We utilize the crash-involved APIs to infer the crash-triggering widgets. Take Figure 1 as an example, the stack trace involves API of *refetch-SearchEngines*, from which we can infer that clicking the *Search engine* on the crash-occurring page would likely trigger the crash.

To automatically find these crash-triggering widgets, we propose a lightweight matching method between widgets and crash-involved APIs extracted from the stack trace. Specifically, we first tokenize the name of widgets and crash-involved APIs by the underscore and Camel Case. If there is an overlap at the token level between the name of a widget and an API, i.e. at least one token is the same after stemming or an abbreviation of the other, we assume the widget is a candidate crash-triggering widget and assign a score based on the percentage of overlapping tokens.

3.4 Exploration Optimization Scorer
Ideally, the crash can be reproduced with the predicted transfer widget for arriving at the crash-occurring page (in Section 3.2) and the crash-triggering widget after reaching the crash-occurring page (in Section 3.3). Yet, in practice, the prediction can be inaccurate, and there might be complex transitions in the app which the prediction model could not capture. Therefore we design the exploration optimization scorer to help plan the exploration holistically and bridge the gap by inaccurate prediction of the first two scorers.

We leverage Q-learning [62], a reinforcement learning method, to help conduct the exploration. The basic idea is to maintain a Q-table, which stores all widgets' value, to record the crash-related and exploration information and memorize the valuable widgets during trial-and-error exploration. When we first reach a new page, the value of each widget on the page is initialized to 0 and updated by reward or penalty after the widget's interaction (details below).

3.4.1 **Formulating Exploration as MDP**. In our approach, we define an instance of the Markov decision process (MDP) to describe the exploration process and adopt Q-learning to optimize the exploration. The MDP can be defined with a 4-tuple, $\langle S, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, where S refers to the set of **states**, $\mathcal{A}$ refers to the set of **actions**, $\mathcal{P}$ refers to the **transition function**, and $\mathcal{R}$ refers to the **reward function**. In the context of crash reproduction, each page in the app represents an individual state. We extract interactable GUI widgets on each page, and interactions (click, long click or type text) with widgets constitute the action set of the state. When we perform an action a_t (an interaction on a widget), the app's state will be changed from state s_t to state s_{t+1}. We first record the transition $\langle s_t, a_t, s_{t+1} \rangle$ to the transition function $\mathcal{P}$, and then assign a reward r_t based on the reward function $\mathcal{R}$. The reward function $\mathcal{R}$ generates a value indicating the quality of a performed action, e.g., whether interacting with a certain widget relates to the crash. Our reward function evaluates an action (i.e., the corresponding widget) as a sum of the following three aspects.

Crash triggering reward. When an action involves the crash-related elements, i.e., reaching the crash-occurring page or triggering crash-involved APIs, the corresponding widget will receive a large positive reward since the exploration in these widgets has a larger possibility of triggering the crash. Next time when the approach reaches the page, it might choose the widget again, and the combination containing the widget might trigger the crash.

New state reward. When an action explores a new GUI page, the corresponding widget will receive a small positive reward to encourage exploiting new states, especially at the beginning of the exploration. With the exploration going on, the new state reward of a widget can be balanced out by the duplicate state penalty.

Duplicate or failure state penalty. When a widget transfers to a known page, it will receive a small penalty since it is less likely to trigger the crash. Besides, when an action transfers to a page out of the app (e.g., opening a browser) or triggers a non-target crash, the corresponding widget will receive a large penalty. It will not be chosen again since it is impossible to trigger the crash.

Next, we will update the value of the widget interaction $Q(s_t, a_t)$ recorded in the Q-table with the Bellman function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q^*(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (1)$$

where α refers to the learning rate and is set to 0.1, γ refers to the discount factor and is set to 0.9, $Q^*(s_{t+1}, a_{t+1})$ refers to the maximum value of all actions in state s_{t+1}.

3.4.2 **Widget selector**. To efficiently focus on the potentially correct planning and also break out the local optimum, we leverage the ϵ-greedy policy [59] to select the widget to be performed following existing studies [46, 72]. Specifically, for each widget, we sum the assigned score from the three scorers (i.e., page reaching scorer, widget hitting scorer, exploration optimization scorer); then choose the widget with the highest sum score with a high probability $1 - \epsilon$, or randomly select other widgets with a low probability ϵ. In practice, ϵ is initially set as a small number close to 0 to enable CrashTranslator to focus on the predicted optimal widget (i.e., the one with the highest sum score). During exploration, the optimal widget may be wrongly predicted and lead the exploration stuck in a repetitive exploration between several pages; at this point, ϵ will

be changed to a big number close to 1, leading CrashTranslator to select the non-optimal widget to break the local optimum.

3.5 Step Replayer

During crash reproducing, CrashTranslator uses an interaction cache to record all interactions with widgets when the app is launched (and the cache will be cleared after the app restart). After the target crash is triggered, the interaction recorded in the cache is crash-reproducing operations from app launch to crash. However, the interactions recorded in the cache may not be the most straightforward way to trigger the crash and might contain some redundant interactions. To make the generated reproducing steps more concise, we automatically eliminate the redundant interactions that lead to repeated or looped transitions. Finally, we convert the concise interaction history into an auto-replay script and human-readable reproducing steps, i.e., "image + text" reproducing steps shown in Figure 1. For each interaction, we highlight the widget to be interacted with on the page's screenshot by a red box and generate the textual reproducing step in the form of "event type + widget name".

3.6 Implementation

We implement our approach in Python and extend functionalities from the following libraries: Appium [1] to interact with Android apps and obtain the view hierarchy of the current page; NLTK [43] to stem word, which is used in the widget hitting scorer (Section 3.3); Ella [16] to check whether crash-involved APIs are triggered, which is used in the exploration optimization scorer (Section 3.4); OpenCV [45] to mark widgets that need to be interacted with on screenshots, which is used in generating reproducing steps (Section 3.5). We run CrashTranslator and perform experiments on a physical x86 Ubuntu 20.04 machine with Android emulators (Android 4.4-7.0).

For the LLM leveraged in the page reaching scorer (Section 3.2), we adopt the pre-trained GPT-3 [4] model from OpenAI[3]. We choose the Curie model as the base model and fine-tune the model through official APIs as described in Section 3.2. Developers only need to set up their OpenAI account, complete the fine-tuning process according to the instructions provided on our website[2], and subsequently utilize our tool to automatically. On average, reproducing one crash requires sending approximately 42.6 prompts (4939.7 tokens), with an estimated cost of around 0.01 USD.

4 EXPERIMENT DESIGN

To evaluate CrashTranslator, we consider the following three research questions:

RQ1: How effective and efficient is CrashTranslator in reproducing crashes from stack trace?

RQ2: What is the contribution of the designed scorers in CrashTranslator for reproduction?

RQ3: Can the reproducing steps generated by CrashTranslator help developers to reproduce crashes?

4.1 Experimental Dataset

In this work, We collect 75 crash reports involving 58 apps from three sources for evaluation, i.e., ReCDroid's dataset [74], AndroR2 dataset [63], and GitHub. ReCDroid is an approach for crash replay based on the textual-described reproducing steps, and we utilize all 33 crash reports in its replicate package. AndroR2 is a dataset of manually-reproduced bug reports for Android apps, and we use all its 22 crash reports. Other reports, e.g., display bug reports, are out of the scope of this study. Note that the above 55 (33+22) reports do not necessarily contain stack traces. For those reports without a stack trace, we manually reproduce the crash following the reproducing steps and then extract the stack trace from the log.

We also collect a third dataset from GitHub to further prove the effectiveness of CrashTranslator. In detail, we first crawl and filter 3,566 crash reports with the stack trace (may also contain reproducing steps) from GitHub as described in Section 2, and randomly sample 300 crash reports for manual checking to retrieve the reproducible ones. It is performed independently by three graduate students with 2-4 years of Android development experience, and each report is manually reproduced by two of them. We exclude those that cannot be reproduced (e.g., lack of apks, failed-to-compile apps, environment issues) or require special conditions (e.g., account, hardware). This results in 20 crash reports involving 15 apps, and we refer to this dataset as CrashTranslator's dataset.

Note that the crash reports used in the experiments may contain reproducing steps or screenshots, but CrashTranslator will not use such information and only uses the stack trace to reproduce crashes. Due to space limitations, the details of the dataset can be viewed on our website[2].

4.2 Baselines

To the best of our knowledge, CrashTranslator is the first work to reproduce crashes directly from the stack trace. Existing studies which reproduce crashes from the natural language described reproducing steps, e.g., RecDroid [74] and ReproBot [72], can not work for our task. Nevertheless, we forcefully apply ReCDroid as a baseline for our task and donate as **ReCDroid**$_{ST}$. Specifically, We provide the crash stack trace as its input rather than the reproducing steps, irrespective of whether ReCDroid comprehends the stack trace or not.

In addition, there are automated GUI testing approaches [15, 24, 34, 36, 37, 40, 46, 57, 61] which also explore the app and try to reveal the crashes; hence we utilize these approaches as the baselines to better prove our effectiveness. We choose the following four state-of-the-art approaches from different categories, i.e., Monkey, Humanoid, APE, and Q-testing:

Monkey [40] is a widely-used random-based GUI testing tool that tests the target app with purely random sequences of GUI events or system events. The advantages of Monkey are its ability to perform lots of GUI events quickly and its good compatibility.

Humanoid [34] is a novel deep learning-based GUI testing tool. It trains a deep neural network model from a large-scale crowdsourced human interactions dataset to predict which GUI widgets on the current page are more likely to be interacted with by testers.

Ape [24] is one of the state-of-the-art model-based GUI testing tools. It models the app's behavior by building a finite state machine dynamically. The advantage of Ape is its ability to balance the size and precision of the modelling by using dynamic GUI abstraction.

Q-testing [46] uses reinforcement learning to guide testing toward new pages to find crashes. It rewards GUI events that reach a new page and penalizes events that transfer to an explored page.

4.3 Experimental Setup and Evaluation Metrics

For RQ1, we verify the effectiveness and efficiency of CrashTranslator in two aspects: (1) the percentage of reports that can be successfully reproduced in a given time (denoted as *success rate*). We set one hour following the existing study [46]. (2) The time required for successful reproduction (denoted as *reproducing time*). To mitigate the bias from randomness, we run our approach and the baselines three times and record the average reproducing time.

For RQ2, to investigate the contribution of the designed scorers, we would remove each of them and evaluate the performance for crash reproduction. Note that since the exploration optimization scorer is responsible for the exploration, if removing it, the exploration might be stuck in a local dilemma and could not finish the reproduction. Therefore, we only evaluate the contribution of the other two scorers. In detail, we create two variants of CrashTranslator, i.e., 1) CT_{wp}, the variant **w**ithout the **p**age reaching scorer described in Section 3.2. 2) CT_{ww}, the variant **w**ithout the **w**idget hitting scorer described in Section 3.3. The relative contribution of each scorer is measured by comparing each variant with the original approach in terms of success rate and reproducing time, which is also based on the average of three runs.

For RQ3, we verify the usefulness of reproducing steps generated by CrashTranslator. We invite 14 postgraduate students to participate in this experiment. All of them have experience in mobile application testing, 8 are Android developers with at least 3 years of development experience, and 7 work in the crowdtesting platform. For the 46 crash reports that can be reproduced by CrashTranslator, we ask participants to reproduce the crash manually based on the stack trace or reproducing steps (generated by CrashTranslator). Specifically, each participant is assigned 20 different reports, 10 of which only contain the stack trace, while the other 10 only contain the reproducing steps, thus ensuring that each report is reproduced by 3 participants based on the stack trace and 3 others based on steps. If a participant can reproduce the crash within 30 minutes following the existing study [74], we record the reproducing time; otherwise, we mark it as failing to reproduce. Finally, we compare the success rate and reproducing time based on the stack trace and CrashTranslator-generated reproducing steps.

5 RESULTS AND ANALYSIS

5.1 RQ1: Effectiveness and Efficiency

Table 2 shows the success rate of reproducing crash reports from three datasets. Overall, CrashTranslator can reproduce 61.3% of them (46 out of 75), outperforming the baselines by a large margin, i.e., 171% (61.3% vs. 22.6%) higher than ReCDroid$_{ST}$, 206% (61.3% vs. 20%) higher than Monkey, 142% (61.3% vs. 25.3%) higher than Humanoid, 109% (61.3% vs. 29.3%) higher than Ape, 206% (61.3% vs. 20%) higher than Q-Testing. This shows that our tool can effectively reproduce crashes based on the corresponding stack trace. Specifically, CrashTranslator successfully reproduces 28 (84.8%) reports on

Table 2: Reproduction success rate (Effectiveness). CT, R, M, H, A, Q in the table header refers to CrashTranslator, ReCDroid$_{ST}$, Monkey, Humanoid, Ape and Q-testing respectively.

# Dataset	CT	R	M	H	A	Q
ReCDroid's Dataset (33)	27 (81.8%)	14 (42.4%)	11 (33.3%)	15 (45.5%)	17 (51.5%)	10 (30.3%)
AndroR2 Dataset (22)	10 (45.5%)	2 (9.1%)	0 (0%)	2 (9.1%)	3 (13.6%)	3 (13.6%)
CrashTranslator's Dataset (20)	9 (45%)	1 (5%)	2 (10%)	2 (10%)	2 (10%)	2 (10%)
total (75)	46 (61.3%)	17 (22.7%)	13 (17.3%)	19 (25.3%)	22 (29.3%)	15 (20%)

Table 3: Reproducing time on successfully reproduced reports (Efficiency)

Reproducing Time (s)	Average	Min	Q_1	Median	Q_3	Max
ReCDroid$_{ST}$ (17 reports)	988	10	94	996	1436	2719
Monkey (15 reports)	669	5	48	152	985	2525
Humanoid (19 reports)	532	12	43	216	1094	1642
Ape (22 reports)	531	5	33	84	809	2166
Q-testing (15 reports)	368	6	30	118	548	1506
CrashTranslator (46 reports)	68.7	8	19	44	87	640

the ReCDroid's dataset, which outperforms ReCDroid$_{ST}$ and other automated GUI testing baselines (30.3%-51.5%). While on the AndroR2 dataset and the CrashTranslator's dataset, CrashTranslator can achieve a success rate of 40.9%-45%. In contrast, ReCDroid$_{ST}$ and other automated GUI testing baselines can only successfully reproduce a small portion of reports, with a success rate of 5% to 13.6%. The difference in the success rate of the three datasets is might because that crash reports in the ReCDroid's dataset tend to involve fewer exploration steps for reproduction, while the other two datasets require a longer exploration sequence for reproducing the crash.

For the 29 reports that CrashTranslator fails to reproduce, there are three main reasons for hindering the reproduction: 1) Some apps (3 from the ReCDroid's dataset, 1 from the AndroR2 dataset) could not run in our environment, e.g., the server is down, incompatibility with our emulator. 2) CrashTranslator does not cover all the interactive GUI actions. It already supports such GUI actions as tapping, long pressing and typing, and rotating the screen like existing studies [73, 74], yet some crashes require other types of actions to trigger (e.g. *C-4 Alarmio-47* requires scrolling up and down the screen). 3) Other unsolved technical challenges, which are further discussed in the discussion (Section 6.1).

Table 3 and Table 4 show the reproducing time of CrashTranslator and four baselines on the successfully reproduced reports. For the 46 crash reports that CrashTranslator can reproduce, the average reproducing time of CrashTranslator is 68.7 seconds, which indicates that CrashTranslator can automatically reproduce crashes based on the corresponding stack trace within an acceptable time cost. Compared with the baselines, CrashTranslator performs better than ReCDroid$_{ST}$ and the automated GUI testing techniques in most of the cases. Since different tools can succeed in different crashes, for the average reproducing time, we compare CrashTranslator with each baseline on the crashes that both of them can reproduce. The results show that CrashTranslator is 1110% faster than ReCDroid$_{ST}$ (81.6s vs. 988s), 1611% faster than Monkey (39.1s vs. 669s), 620% faster than Humanoid (73.9s vs. 532s), 705% faster than Ape (66s vs. 531s), and 302% faster than Q-testing (89.6s vs. 360s).

5.2 RQ2: Contribution of Different Scorers

Columns CT$_{wp}$ and CT$_{ww}$ in Table 4 show the reproduction results for the variants of CrashTranslator without the page reaching scorer and without the widget hitting scorer, respectively. Overall, CT$_{wp}$ can only successfully reproduce 42 crash reports, 4 fewer than CrashTranslator. Meanwhile, the average reproducing time of CT$_{wp}$ is 127.1 seconds, which is 82% slower than CrashTranslator (69.7s vs. 127.1s) considering the reports both can reproduce. For CT$_{ww}$, it can successfully reproduce 44 crashes (2 fewer than CrashTranslator), and the average reproducing time is 113.2s (63% slower than CrashTranslator). The inferior performance of CT$_{wp}$/CT$_{ww}$ indicates that the two scorers can significantly improve the effectiveness and efficiency of crash reproducing.

We further examine the detailed difference between the results of CT$_{wp}$/CT$_{ww}$ and CrashTranslator for a thorough understanding. For the reports which involve shorter exploration sequences for reaching the crash-occurring page, e.g., *R-10 FastAdapter-394*, we find that excluding the page reaching scorer or widget hitting scorer would not largely influence the reproduction. This might be because, in these cases, the widget transferring to the crash-occurring page can be found by one of the scorers or even by traversal. By comparison, for the reports which involve longer exploration sequences for reproduction (e.g., *A-5 andOTP-500*), or the entry page has dozens of candidate widgets (e.g., *R-5 AnyMemo-18*), it is difficult to find the reproducing steps solely by the traversal or the widget hitting scorer. In this case, the page reaching scorer which is accomplished by LLM contributes significantly to the crash reproduction by providing step-by-step guidance to the crash-occurring page. Besides, for the reports which have many candidate widgets on the crash-occurring page (e.g., *R-14 SMSsync-464*), the efficiency can be largely improved by the widget hitting scorer that predicts which widgets should be interacted with to trigger the crash.

Still, we need to admit that on some reports like *R-11 LibreNews-22*, removing the scorer would improve the crash reproduction efficiency. This is mainly because the scorers can occasionally fail to conduct an accurate prediction, and with the wrong guidance, the exploration might go in the wrong direction and waste time.

5.3 RQ3: Usefulness of CrashTranslator

Columns P$_{trace}$ and P$_{step}$ of Table 4 show the results of participants' manual reproduction based on stack traces and reproducing steps (generated by CrashTranslator), respectively. Following the reproducing steps generated by CrashTranslator, 100% of reports are successfully reproduced by at least two participants. As a comparison, when we only provide the stack trace, this percentage drops to 63% (29/46), and there are 6 reports that none of the participants can reproduce. Besides, the average reproducing time with CrashTranslator is 66.7 seconds, which is 215% faster than reproducing from stack traces (57.3s vs. 180.5s) considering the reports that both have at least one participant can reproduce. This indicates the usefulness of our proposed CrashTranslator, whose generated reproducing steps can supplement the stack-trace-only crash reports and make the crashes easily to be reproduced.

Furthermore, on 91.3% (42/46) reports, the reproduction is faster for the automatic approach CrashTranslator, when compared with the average time of human reproduction directly from the stack

Table 4: Reproduction details on three datasets. CT, R, M, H, A, Q in the table header refers to CrashTranslator, ReCDroid$_{ST}$, Monkey, Humanoid, Ape and Q-testing respectively. CT$_{wp}$ and CT$_{ww}$ refer to variants of CrashTranslator without the page reaching scorer and without the widget hitting scorer, respectively. If the above approach successfully reproduces the crash, we record the reproduction time (in seconds) in the table, and if it fails, we record ×. Columns P$_{trace}$ and P$_{step}$ show the average time for participants to manually reproduce the crash based on the stack trace or reproducing steps generated by CrashTranslator, respectively, and the number in parentheses is the number of participants who reproduce crash successfully (out of 3). Note that in order to save space, reports that cannot be reproduced by either approach are omitted.

ID	#Crash Reports	RQ 1						RQ 2		RQ 3	
		CT	R	M	H	A	Q	CT$_{wp}$	CT$_{ww}$	P$_{trace}$	P$_{step}$
R-1	NewsBlur-1053	32	176	2024	×	68	×	53	69	201 (3)	69 (3)
R-2	Markor-194	88	×	×	×	2166	1156	52	94	540 (1)	120 (3)
R-3	Birthdroid-13	112	1776	×	×	1564	×	100	127	230 (2)	73 (3)
R-4	Car Report-43	46	2558	×	×	×	×	58	46	475 (2)	92 (3)
R-5	AnyMemo-18	19	×	125	644	111	×	×	23	63 (3)	16 (3)
R-6	AnyMemo-440	90	×	×	×	950	×	×	×	342 (2)	91 (3)
R-7	Notepad-23	64	2719	×	1223	855	×	120	70	× (0)	116 (3)
R-8	Olam-2	10	×	1288	×	101	×	10	10	62 (3)	23 (3)
R-9	Olam-1	10	×	×	×	37	×	10	10	24 (3)	13 (3)
R-10	FastAdapter-394	8	526	10	1642	13	1506	8	8	61 (3)	36 (3)
R-11	LibreNews-22	93	1261	×	×	×	×	31	139	107 (1)	42 (3)
R-12	LibreNews-23	71	×	126	104	49	518	×	51	× (0)	83 (3)
R-13	LibreNews-27	93	1272	×	197	×	×	31	139	11 (1)	64 (3)
R-14	SMSsync-464	118	×	×	443	×	63	186	254	213 (2)	68 (3)
R-15	Transistor-63	41	94	×	216	×	17	10	48	151 (3)	63 (3)
R-16	Zom-271	50	536	45	×	50	24	52	59	120 (1)	80 (3)
R-17	Pix-Art-125	48	1436	×	977	541	219	48	176	246 (3)	34 (2)
R-18	Pix-Art-127	56	1270	×	1416	1972	53	57	164	300 (1)	49 (3)
R-19	ScreenCam-25	33	×	152	28	43	×	40	33	69 (2)	68 (3)
R-20	ownCloud-487	60	996	303	262	×	118	62	60	228 (1)	87 (3)
R-21	OBDReader-22	52	×	2525	1211	×	×	×	53	210 (2)	76 (3)
R-22	Dagger-46	8	18	8	12	5	×	8	8	180 (1)	14 (3)
R-23	ODK-2086	640	2064	×	1538	531	725	939	1016	180 (1)	41 (3)
R-24	K-9Mail-3255	13	×	×	×	×	×	16	18	256 (3)	43 (3)
R-25	K-9Mail-2612	11	×	×	×	×	×	22	×	194 (3)	65 (3)
R-26	K-9Mail-2019	12	×	×	×	×	×	16	21	185 (2)	68 (3)
R-27	TagMo-12	42	×	52	47	12	×	25	42	176 (2)	43 (2)
R-28	FlashCards-13	23	×	×	×	×	×	21	21	73 (2)	18 (3)
A-1	HABPanel-25	21	×	×	×	672	25	28	29	44 (3)	19 (3)
A-2	Noad Player-1	8	10	5	12	5	6	10	10	13 (3)	14 (3)
A-3	Weather-61	31	×	×	×	×	578	40	34	47 (2)	16 (3)
A-4	Berkeley-82	8	50	683	39	22	×	8	8	41 (3)	23 (3)
A-5	andOTP-500	120	×	×	×	×	×	425	323	104 (3)	103 (3)
A-6	K-9Mail-3255	14	×	×	×	×	×	15	19	45 (2)	41 (3)
A-7	K-9Mail-3971	66	×	×	×	×	×	95	85	× (0)	152 (3)
A-8	Firefox-3932	61	×	×	×	×	×	107	83	× (0)	47 (3)
A-9	Aegis-3932	117	×	×	×	×	×	159	129	205 (3)	146 (2)
C-1	NewPipe-7825	32	×	×	×	×	×	46	32	426 (1)	64 (3)
C-2	SDBViewer-10	15	×	×	68	1900	36	20	192	98 (3)	25 (3)
C-3	Anki-10584	180	×	×	×	×	×	878	312	175 (2)	148 (2)
C-4	Alarmio-47	×	×	×	×	×	487	×	×	A- (0)	A- (0)
C-5	Shuttle-456	87	×	×	×	×	×	96	97	476 (1)	132 (3)
C-6	Anki-3370	35	×	×	×	×	×	611	35	183 (2)	41 (3)
C-7	WhereUGo-368	165	×	2524	×	×	×	372	181	× (0)	157 (3)
C-8	GrowTracker-87	210	×	×	×	×	×	271	605	× (0)	221 (3)
C-9	FakeStandby-30	27	×	×	×	×	×	27	27	386 (1)	38 (3)
C-10	getodk-219	20	45	166	29	32	×	157	20	80 (3)	25 (3)

trace. This further implies the usefulness of CrashTranslator which provides an automatic solution and can be faster than humans.

After practitioners have manually reproduced the bug reports, we also conduct an unstructured interview about the challenges they encountered in reproducing crashes based on stack traces only. Most participants complain that crash-related information in stack traces is too limited to infer how to reproduce crashes. They usually need to spend a long time trying to reach crash-occurring pages and finding crash-triggering widgets, and after many attempts, they may lose interest and assume that crashes are not reproducible. By comparison, they agree that CrashTranslator can automatically conduct the tedious process of exploring the app and finding paths to crash-occurring pages and crash-triggering widgets.

6 DISCUSSION

6.1 Limitations

Except for the two engineering limitations discussed in Section 5.1, there are three other technical limitations of CrashTranslator which hinder it from reproducing all bug reports, This also indicates the challenges in crash reproduction from stack traces and calls for further research.

First, CrashTranslator may fail to reproduce crashes that do not contain crash-occurring pages or crash-involved APIs in stack traces. In some special cases, crashes do not occur in a specific activity or fragment, e.g., faults related to network communication. For these cases, crash-occurring pages and crash-involved APIs are not available from the stack trace, CrashTranslator may degenerate into an automated GUI testing tool for aimless exploration due to the lack of guidance from the stack trace.

Second, CrashTranslator cannot reproduce crashes requiring valid input contents, e.g., performing a crash-triggering log-in requires a valid username and password. Such crashes require human intervention and cannot be fully automated since input contents are not present in the stack trace. Nevertheless, by asking the users to provide the information in advance, CrashTranslator can still conduct the automatic reproduction for these cases.

Third, CrashTranslator cannot capture special preconditions that trigger crashes. For example, a display GUI page works fine with default settings, but if switching to a dark theme, a crash would occur when reaching the page. In this case, CrashTranslator can correctly understand that the display page is a crash-occurring page, but it could not successfully reproduce the crash even if the correct page is reached. This is because CrashTranslator can hardly capture the triggering precondition of *switching to the dark theme* from the stack trace. In the future, we will investigate incorporating the crash-triggering preconditions into CrashTranslator with the help of static code analysis and other techniques.

6.2 Threats to Validity

The first threat relates to the randomness of CrashTranslator. Our widget selector (Section 3.4.2) will select a non-optimal widget with a certain probability. To reduce this threat, we run CrashTranslator and its variants (CT$_{wp}$ and CT$_{ww}$) three times and record the average reproducing time in experiment results.

The second threat relates to the choice of parameter settings of CrashTranslator that may affect the effectiveness and efficiency of crash reproduction. In order to mitigate the threat, we conduct small-scale experiments on several "crash" reports that are excluded from our experimental data to determine suitable settings before

the evaluation. These reports come from the data collection (Section 2) when we find some "crash" reports with stack traces, but they can not be triggered by ourselves due to incompatibility with our environment.

The third threat relates to the confounding effects of participants. Following the existing approach [74], we assume that students with Android programming experience can be substituted for testers, and their reproducing time and success rate are representative.

7 RELATED WORK

Mobile Bug Reports Analysis and Reproducing. There are several studies which utilize natural language processing techniques to extract critical information from mobile bug reports, such as summarizing and classifying bug reports [20, 47], facilitating dynamic analysis [28, 66], augmenting bug reports for mobile apps [35, 41] and generating test cases [6, 41].

Several studies focus on crash reproduction of mobile bug reports with step-by-step guidance, i.e., textual described reproducing steps [72–74] and visual recordings [18]. Specifically, ReCDroid [74] and ReCDroid+ [73] leveraged the natural language described reproducing steps to perform reproduction. It designed a set of predefined grammar patterns to extract events and objects from textual reproducing steps and then adopted a greedy-based dynamic exploration to synthesize event sequences. ReproBot [72] went a step further by analyzing the reproducing steps more accurately and designing a new exploration strategy to find the best match between steps and GUI actions. GIFdroid [18] leveraged the visual screen record to perform reproduction. It adopted image-processing techniques to map the keyframes in recording to GUI states and generated reproducing traces based on the transition graph. Compared to the aforementioned crash reproduction tools, CrashTranslator achieves the following advances: 1) Our work enables the reproduction of crashes from stack traces without relying on step-by-step guidance. This is a more challenging task, and existing tools perform poorly; 2) We propose a novel approach which leverages LLM and reinforcement learning to predict and guide the exploration steps for trigger the crash, which is more effective and efficient than previous techniques.

There are also studies which record and replay bugs in mobile and web apps by using running information [22, 33, 44, 64, 71] and textual description [3, 31]. Among these studies, CrashDroid [64] generated reproducing steps by translating the call stack, which contains all method calls from app launch to app crash. It requires the call stack collected by the specific mechanism throughout the app's run, while our approaches only need the automatically generated stack trace when the crash occurs.

Stack Trace Analysis. Stack traces offer exception-related information about an app. Schroter et al. [51] empirically indicated that the stack trace information was very helpful to developers when debugging. Subsequently, several automatic approaches were proposed to recover the links between the crashes and their cause functions and assist developers in locating crashing faults [23, 26, 42, 65, 67]. Going a step further, researchers started to explore how to utilize the located faulty functions to help developers fix bugs, e.g., generating test cases for fault functions [7, 50, 52–54, 68], and finding the best developer to fix the bug [32, 58]. Besides, there

were some studies focused on calculating the similarity between the stack trace, which could be used to distinguish duplicate crash reports [11, 30, 48, 49, 60]. This study opens a new direction, i.e., the crash reproduction directly from the stack trace.

8 CONCLUSION

Crash reports from open-source platforms are vital for ensuring mobile application quality. Still, the crash-related information they record is not always complete, e.g., they may only contain the crash stack trace but lack the reproducing steps, which hinders developers from fixing issues. This paper proposes a novel reproduction approach named CrashTranslator which automatically reproduces crashes from stack traces of mobile bug reports. It adopts three scorers, i.e., page reaching scorer, widget hitting scorer, and exploration optimization scorer, to select crash-related widgets iteratively until the target crash is triggered. We evaluate the CrashTranslator on 75 bug reports, and it can successfully reproduce 46 (61.3%) of the crashes within an acceptable time cost, largely outperforming automated GUI testing baselines.

REFERENCES

[1] Appium. 2023. https://appium.io.
[2] APPLAUSE. 2023. https://www.applause.com/blog/app-abandonment-bug-testing.
[3] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight recording to reproduce field failures. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 362–371.
[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[5] Andrew Cantino. 2016. Prompt Engineering Tips and Tricks with GPT-3. https://blog.andrewcantino.com/blog/2021/04/21/prompt-engineering-tips-and-tricks/.
[6] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 86–96.
[7] Ning Chen and Sunghun Kim. 2014. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering* 41, 2 (2014), 198–220.
[8] Xiang Chen, Ningyu Zhang, Xin Xie, Shumin Deng, Yunzhi Yao, Chuanqi Tan, Fei Huang, Luo Si, and Huajun Chen. 2022. Knowprompt: Knowledge-aware prompt-tuning with synergistic optimization for relation extraction. In *Proceedings of the ACM Web Conference 2022*. 2778–2788.
[9] Google Code. 2023. https://code.google.com.
[10] Firebase Crashlytics. 2023. https://firebase.google.com/products/crashlytics.
[11] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1084–1093.
[12] Carlos DA de Almeida, Diego N Feijó, and Lincoln S Rocha. 2022. Studying the impact of continuous delivery adoption on bug-fixing time in apache's open-source projects. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 132–136.
[13] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 845–854.

[14] Ivan Dimoski. 2014. Automated Android Crash Reports with ACRA and Cloudant. https://www.toptal.com/android/automated-android-crash-reports-with-acra-and-cloudant.

[15] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 481–492.

[16] ella. 2019. https://github.com/saswatanand/ella.

[17] F-Droid. 2023. https://f-droid.org/.

[18] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1045–1057.

[19] GATOR. 2019. http://web.cse.ohio-state.edu/presto/software/gator/.

[20] Michael Gegick, Pete Rotella, and Tao Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 11–20.

[21] GitHub. 2023. https://github.com.

[22] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*. 72–81.

[23] Liang Gong, Hongyu Zhang, Hyunmin Seo, and Sunghun Kim. 2014. Locating crashing faults based on crash stack traces. *arXiv preprint arXiv:1404.4100* (2014).

[24] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.

[25] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. Ppt: Pre-trained prompt tuning for few-shot learning. (2021).

[26] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. 2019. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software* 148 (2019), 88–104.

[27] Tri Huynh, Alessio Gambi, and Gordon Fraser. 2019. AC3R: Automatically Reconstructing Car Crashes from Police Reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 31–34. https://doi.org/10.1109/ICSE-Companion.2019.00031

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

[29] Bugzilla keyword descriptions. 2023. https://bugzilla.mozilla.org/describekeywords.cgi.

[30] Aleksandr Khvorov, Roman Vasiliev, George Chernishev, Irving Muller Rodrigues, Dmitrij Koznov, and Nikita Povarov. 2021. S3M: Siamese stack (trace) similarity measure. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 266–270.

[31] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, and Paolo Tonella. 2014. Reproducing field failures for programs with complex grammar-based input. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 163–172.

[32] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 486–493.

[33] Amy J Ko and Brad A Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 387–396.

[34] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[35] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated classification of actions in bug reports of mobile apps. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 128–140.

[36] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[37] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.

[38] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of gui events for test reuse: are we there yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–190.

[39] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–118.

[40] Monkey. 2023. http://developer.android.com/tools/help/monkey.html.

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

[42] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 151–160.

[43] NLTK. 2023. https://www.nltk.org.

[44] Dmitry Nurmuradov and Renee Bryce. 2017. Caret-HM: recording and replaying Android user sessions with heat map generation using UI state clustering. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–403.

[45] opencv. 2023. https://opencv.org.

[46] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164.

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

[48] Irving Muller Rodrigues, Daniel Aloise, and Eraldo Rezende Fernandes. 2022. FaST: A linear time stack trace alignment heuristic for crash report deduplication. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 549–560.

[49] Irving Muller Rodrigues, Aleksandr Khvorov, Daniel Aloise, Roman Vasiliev, Dmitrij Koznov, Eraldo Rezende Fernandes, George Chernishev, Dmitry Luciv, and Nikita Povarov. 2022. TraceSim: An Alignment Method for Computing Stack Trace Similarity. *Empirical Software Engineering* 27, 2 (2022), 53.

[50] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. Reconstructing core dumps. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 114–123.

[51] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 118–121.

[52] Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie Van Deursen. 2020. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering* 25 (2020), 96–138.

[53] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2016. Evolutionary testing for crash reproduction. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*. 1–4.

[54] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2017. A guided genetic algorithm for automated crash reproduction. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 209–220.

[55] Statista. 2022. Combined global Apple App Store and Google Play app downloads from 1st quarter 2015 to 4th quarter 2022. https://www.statista.com/statistics/604343/number-of-apple-app-store-and-google-play-app-downloads-worldwide/.

[56] StoryDistiller. 2022. https://github.com/tjusenchen/StoryDistiller.

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

[58] Denis Sushentsev, Aleksandr Khvorov, Roman Vasiliev, Yaroslav Golubev, and Timofey Bryksin. 2022. DAPSTEP: Deep Assignee Prediction for Stack Trace Error rePresentation. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 184–195.

[59] Sebastian Thrun and Michael L Littman. 2000. Reinforcement learning: an introduction. *AI Magazine* 21, 1 (2000), 103–103.

[60] Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. 2020. TraceSim: a method for calculating stack trace similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*. 25–30.

[61] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 469–480.

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

[63] Tyler Wendland, Jingyang Sun, Junayed Mahmud, SM Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A dataset of manually-reproduced bug reports for android apps. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 600–604.

[64] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating reproducible and replayable bug reports from android application crashes. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 48–59.

[65] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE international conference on software maintenance and evolution*. IEEE, 181–190.

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

[67] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 204–214.

[68] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 910–913.

[69] Jiwei Yan, Shixin Zhang, Yepang Liu, Xi Deng, Jun Yan, and Jian Zhang. 2022. A Comprehensive Evaluation of Android ICC Resolution Techniques. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[70] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 105–109. https://doi.org/10.1145/3510454.3516864

[71] Jiwei Yan, Hao Zhou, Xi Deng, Ping Wang, Rongjie Yan, Jun Yan, and Jian Zhang. 2021. Efficient testing of GUI applications by event sequence reduction. *Science of Computer Programming* 201 (2021), 102522.

[72] Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William GJ Halfond. 2023. Automatically Reproducing Android Bug Reports Using Natural Language Processing and Reinforcement Learning. *arXiv preprint arXiv:2301.07775* (2023).

[73] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–33.

[74] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 128–139.