Feedback-Driven Automated Whole Bug Report Reproduction for Android Apps

Dingbang Wang
University of Connecticut
USA
dingbang.wang@uconn.edu

Yu Zhao
University of Cincinnati
USA
zhao3y3@ucmail.uc.edu

Sidong Feng
Monash University
Australia
sidong.feng@monash.edu

Zhaoxu Zhang
University of Southern California
USA
zhaoxuzh@usc.edu

William G.J. Halfond
University of Southern California
USA
halfond@usc.edu

Chunyang Chen
Monash University
Australia
chunyang.chen@monash.edu

Xiaoxia Sun
China Mobile(Suzhou) Software Technology Co., Ltd.
China
18896724798@139.com

Jiangfan Shi
Zhejiang University
China
shijiangfan@dragontesting.cn

Tingting Yu
University of Connecticut
USA
tingting.yu@uconn.edu

Abstract

In software development, bug report reproduction is a challenging task. This paper introduces ReBL, a novel feedback-driven approach that leverages GPT-4, a large-scale language model, to automatically reproduce Android bug reports. Unlike traditional methods, ReBL bypasses the use of Step to Reproduce (S2R) entities. Instead, it leverages the entire textual bug report and employs innovative prompts to enhance GPT's contextual reasoning. This approach is more flexible and context-aware than the traditional step-by-step entity matching approach, resulting in improved accuracy and effectiveness. In addition to handling crash reports, ReBL has the capability of handling non-crash bug reports. Our evaluation of 96 Android bug reports (73 crash and 23 non-crash) demonstrates that ReBL successfully reproduced 90.63% of these reports, averaging only 74.98 seconds per bug report. Additionally, ReBL outperformed three existing tools in both success rate and speed.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Android, Automated Bug Reproduction, Large Language Model, Prompt Engineering

ISSTA 2024, 16-20 September, 2024, Vienna, Austria

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Dingbang Wang, Yu Zhao, Sidong Feng, Zhaoxu Zhang, William G.J. Halfond, Chunyang Chen, Xiaoxia Sun, Jiangfan Shi, and Tingting Yu. 2024. Feedback-Driven Automated Whole Bug Report Reproduction for Android Apps. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In software development, debugging and fixing are crucial, especially in the mobile app marketplace. According to [11], 88% of app users are likely to abandon an app if they encounter recurring issues, underlining the need for swift issue resolution to retain users. One major challenge developers face is effectively reproducing bugs reported by users, which often lack crucial details like the sequence of user interactions [17, 20, 33, 39]. To address this, the software engineering community is increasingly interested in automating the bug reproduction process.

Several existing approaches have been developed to automate bug reproduction [26, 28, 52, 54, 55]. These methods follow two phases in bug reproduction: 1) extracting entities from steps to reproduce (S2Rs), and 2) explicitly matching the extracted entities with the actual app UI to find the sequence of events that replays the S2Rs or reproduces the reported bug. However, these approaches have limitations. First, S2Rs are often unclear, imprecise, or ambiguous, posing a significant challenge to state-of-the-art NLP techniques [28]. Second, explicitly matching bug reports with app UI can result in missing reproduction steps due to incomplete bug reports. Existing techniques use resource-intensive dynamic exploration algorithms and human-defined heuristics to address this issue, leading to reduced effectiveness and higher costs.

Recent work, AdbGPT [28] utilizes large language models (LLMs), i.e., GPT-3.5, to extract S2R entities from bug reports and then iteratively employs ChatGPT [14] to make decisions for selecting UI widgets to replay the extracted S2Rs. In S2R Entity Extraction, few-shot learning is applied to guide the LLM in recognizing entities

related to bug reproduction. The Guided Replay phase involves dynamically guiding LLMs based on GUI screens. This approach leverages the remarkable capabilities of GPT to comprehend natural language and act as an expert developer of extracting S2Rs and guiding GUI exploration.

While AdbGPT represents improvements over previous approaches in effectiveness and efficiency, it still faces several challenges. First, like many existing approaches [52–54], it strictly adheres to the use of S2R entities, following a two-phase structure: the S2R Entity Extraction phase and subsequent matching with UI widgets. As acknowledged in existing work [20, 25, 39], bug reports often suffer from a substantial cognitive and lexical gap between reporters and developers, leading to ineffective communication of crucial reproduction steps and inconsistent report quality. The use of S2R entities can exacerbate this situation because (i) the Entity Extraction phase may omit essential details; (ii) the original S2R may be ambiguous or inaccurate; and (iii) extracted entities overlook the actual UI context encountered during bug reproduction. Second, when implicit input values are involved, AdbGPT uses a "TEST" placeholder for text fields, risking invalid GUI exploration. However, text inputs are crucial for triggering some bugs and significantly influence testing and bug detection. Third, it overlooks the inherent randomness in large language models' outputs, potentially resulting in inaccuracies in matching GUI widgets and diminishing effectiveness. Finally, AdbGPT terminates UI exploration when all S2Rs are covered, lacking the capacity to assess whether the bug is being triggered or not.

In this paper, we introduce ReBL, a novel feedback-driven approach utilizing GPT to automate bug reproduction. Unlike existing methods, ReBL utilizes the entire bug report, eliminating the need to use S2R entities. This streamlines the process and ensures the original bug report's description remains intact, avoiding potential omissions during the S2R Extraction phase. The feedback-driven design enriches the GPT model with rich UI context, enabling flexible, context-aware actions crucial for accurate bug reproduction. During reproduction, ReBL diverges from a rigid step-by-step approach, unlike AdbGPT. Instead, it employs a feedback-driven methodology that considers the bug report, the current app state, and the reproduction history to make informed decisions. ReBL uses innovative techniques to capture UI context, addressing incomplete and ambiguous bug report information. Additionally, it integrates novel strategies to mitigate randomness in LLMs' responses and automatically adapt to correct behavior.

Overall, ReBL is a tool that average software developers can easily adopt and benefit from without requiring in-depth knowledge of LLMs technology. First, it is designed to be end-to-end, requiring users to input only the bug report and APK file. Second, it automatically generates feedback based on the bug report, app state, and action status, eliminating the need for manual input. Third, ReBL integrates seamlessly into existing workflows and bug tracking systems, allowing for easy incorporation into bug resolution processes without disruption.

ReBL has been implemented as a powerful software tool built on top of GPT-4 [15] and UI Automator2 [9]. To assess the effectiveness of our approach, we conducted extensive experiments by running ReBL on a substantial dataset comprising 73 crash bug reports and 23 non-crash bug reports. The results show that ReBL demonstrated an impressive success rate by successfully reproducing 87 bugs (69 crash and 18 non-crash bugs), accounting for 90.63% of the total bug reports in the dataset, with an average reproduction time of 74.98 seconds. To provide further insights into the advantages of our approach over the state-of-the-art, we conducted a comparative analysis of 73 crash bug reports against three existing tools: ReCDroid [55], ReproBot [52], and AdbGPT [28]. The success rates for these tools are 45.21%, 65.75%, and 73.97%, respectively, while ReBL achieved an impressive 94.52%. Moreover, our approach stands out as the fastest, with an average time of 72.11 seconds, compared to ReCDroid (534.92s), ReproBot (413.72s), and AdbGPT (89.80s).

In summary, our paper makes the following contributions:

- ReBL, the first tool capable of reproducing bugs using the whole bug reports without the use of S2R entities and specific bug type domains, streamlining the debugging process.
- An empirical study demonstrating the effectiveness of ReBL in reproducing both crash and non-crash bugs for Android bug reports.
- We made the implementation and dataset publicly available for future research work [16].

2 Preliminaries and Motivation

In this section, we introduce the essential preliminaries for automated bug reproduction and provide motivating examples. These examples highlight the limitations of existing approaches and showcase the advantages of our approach.

2.1 Preliminaries

A UI *widget* is a graphical element of an app, such as a button, a text field, and a check box. A UI *action* is the action performed by the app. It can either be an explicit action on a UI widget (e.g., click) or an implicit action (e.g., wait, phone call). In our setting, a *state* represents an app page (i.e., a set of widgets shown on the current screen. If the set of widgets is different, we have another state). *UI information* represents the content of the widgets extracted from the current app state. *Successful reproduction* is defined as the scenario in which the buggy behavior specified in the bug report is accurately triggered during the bug reproduction process.

2.2 Comparison with Existing Techniques

Current state-of-the-art bug report reproduction techniques typically focus on using steps to reproduce (S2Rs) as the initial input for reproduction [26, 28, 52, 53]. Some approaches propose automated techniques to extract S2Rs. For example, ReCDRoid+ [54] uses a deep learning algorithm to extract S2Rs from the full bug reports. The extracted S2Rs are typically represented as <action, target UI widget, input values>.

For the actual bug report reproduction, existing approaches use various techniques and algorithms to explicitly match S2R with the app UI. This matching process determines the priority of UI widgets in the reproduction approaches' exploration. For example, ReCDroid [53] uses Word2Vec to match S2R entities (e.g., the target UI widget) with the UI widgets in the app and then employs a guided DFS to find the most relevant GUI widget iteratively. Zhang et al.[52] calculate a similarity score to measure the similarity between an action's UI event and S2R. It then uses Q-learning to learn

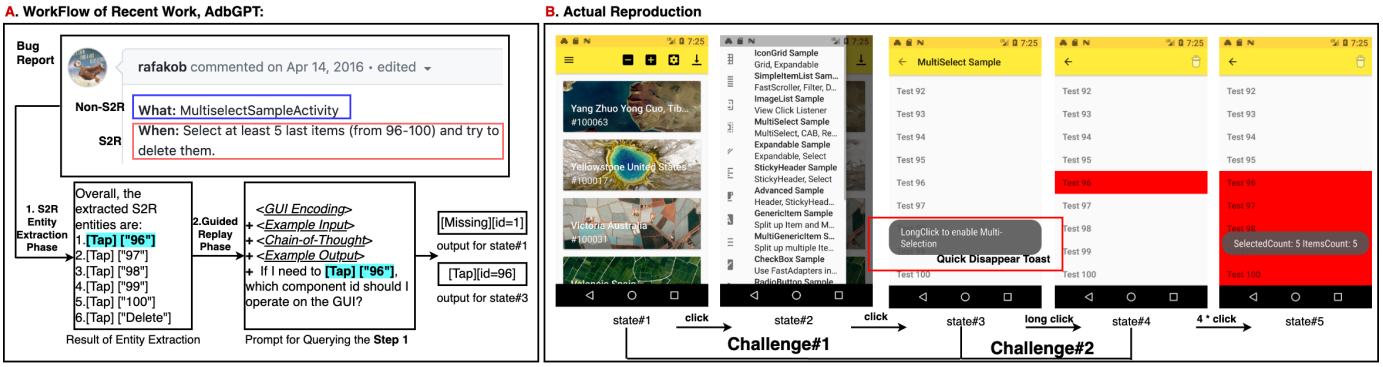


Figure 1: Motivation Example

how to match UI events with the S2Rs and bridge missing steps to calculate a UI event sequence that can lead to the observed failure. ScopeDroid [31] matches the S2R with the state transition graph (STG) generated from the app. The matching results are used to plan a path in STG to guide bug report reproduction. Despite their contributions, they have inherent limitations, which manifest in at least one of the following aspects: (1) Difficulties in accurately extracting S2R from bug reports, mainly due to the complexity and diversity of sentence structures found in these reports; (2) Challenges in inferring missing steps due to the limited knowledge or understanding of the bug reproduction context; (3) Significant costs associated with the matching and dynamic analysis phases, making the bug reproduction process resource-intensive.

The most recent work, AdbGPT [28], addresses the above limitations using large language models (LLMs). By leveraging their advanced natural language understanding and decision-making capabilities, AdbGPT significantly enhances the accuracy and efficiency of reproducing Android bugs automatically. Similar to conventional approaches, AdbGPT initially identifies S2R entities from manually supplied S2R sentences and then use these extracted entities as prompts against the UI widgets to determine the optimal widget for replaying the S2Rs. Nevertheless, like other techniques, AdbGPT exhibits the following limitations:

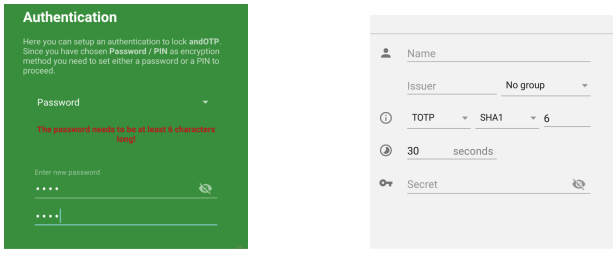
Challenge 1: Overlooking non-S2R information. Existing approaches focus on extracting entities from the S2R segment, which risks overlooking other essential information for bug reproduction. Figure 1 exemplifies a bug report that failed to be reproduced by existing approaches with S2R entity extraction. Figure 1A displays the entity extraction result from AdbGPT [28]. This result reveals an omission of crucial information, "What:MultiselectSampleActivity", because it is not in the S2R segment. The oversight of this non-S2R information results in inadequate information to bridge the missing steps from the home page (state#1) to this specific page (state#3), especially since many pages in this app feature a similar UI structure as state#3, e.g., many pages have items numbered 1-100. Failing to consider this non-S2R information during the reproduction process makes it challenging to determine the exact location where the S2R should be performed. Conversely, our approach utilizes the whole bug report and bypasses the Entity Extraction phase, comprehending the bug report as a cohesive whole.

Challenge 2: Incapable of handling incomplete or ambiguous S2Rs. The S2Rs written by users might be incomplete or ambiguous, and further extraction of S2R entities could potentially exacerbate these issues. Additionally, the extracted entities might lack flexibility, as they are derived from the S2Rs without considering the actual UI context. Continuing with the example from Figure 1, assume that Challenge 1 has been addressed, allowing AdbGPT to proceed with the reproduction process from state#3. In the Entity Extraction phase, "Select at least 5 last items (from 96-100)" has been broken down by AdbGPT into 5 steps, with the first step being ["tap"] ["96"]. As presented in Figure 1A, we demonstrate how AdbGPT uses prompt engineering queries in LLMs for suggestions on executing this step. However, while ["tap"] ["96"] may appear valid according to the bug report and the extraction process, it proves to be invalid in the actual reproduction process. As can be seen in state#3 in Figure 1B, a toast, which is a widget that disappears quickly, suggests "LongClick to enable Multi-Selection", implying that to multi-select items 96-100, one must first *long-click* on item 96. Since AdbGPT is only looking to match the extracted entities <["tap"] ["96"]> to the UI page, it will not be able to perform the long click/tab action.

To address this problem, our approach eliminates the use of S2R entities, thereby avoiding presuming the action for each step and instead relying on a holistic approach, considering both the complete bug report and the rich UI context to determine the most appropriate action. In this scenario, it recognizes the presence of the quickly disappearing toast message and takes into account its context to perform a long click on the target widget.

Challenge 3: Less sophisticated input generation. Existing bug reproduction tools adopt less sophisticated strategies in filling text fields when explicit inputs are lacking in S2Rs, which can lead to invalid inputs, potentially resulting in failed reproduction. These methods include generating random text [31], employing the generic placeholder [52], relying on predefined dictionaries to fill text boxes, and defaulting to placeholders when unsuitable [53]. AdbGPT [28] specifically uses the generic placeholder "TEST." Particularly challenging are scenarios involving text fields with complex requirements, such as password fields illustrated in Figure 2A. These password fields necessitate a minimum length, and it is common for such fields to demand a combination of letters and numbers, making simplistic placeholders insufficient. Furthermore, password confirmation fields require matching inputs, rendering random generation methods unsuitable. Figure 2B is another example with

fields requiring numeric, alphabetic, and unspecified types of input(e.g., the *Secret* text field) on the same page. Therefore, a more versatile fill-blank method is necessitated. Addressing these, our approach, REBL, intelligently infers input values to fill in blanks leveraging the UI context of the current page. Furthermore, due to the feedback mechanism and flexibility of REBL, it is also capable of correcting any invalid inputs, provided there is a warning message, thereby ensuring the accuracy and appropriateness of the inputs.



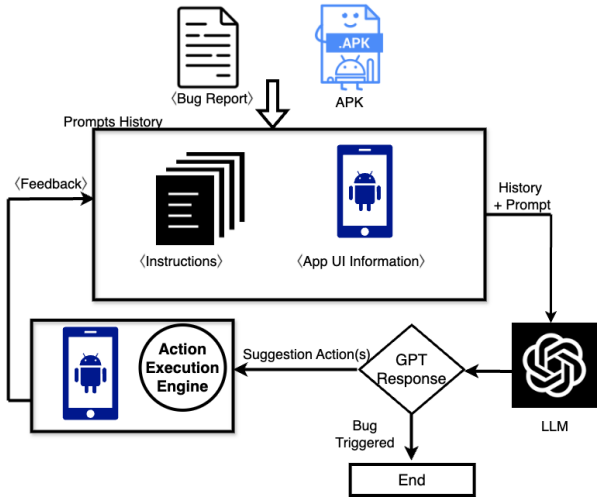
A. Fill Blank #1 B. Fill Blank #2

Figure 2: Example of Inferring Input Values

Challenge 4: Lack of LLM response management. AdbGPT [28] leverages the advanced capabilities of LLMs but lacks effective mechanisms for managing their outputs. When integrating the LLM's output into a program, defining a custom format for automated interpretation is crucial. However, due to the inherent randomness of LLMs, the output might not always adhere to the desired format or could be in the correct format but contain incorrect information, leading to execution failures or program errors. Despite the potential benefits of setting temperature, complete control is not achievable. Moreover, AdbGPT lacks mechanisms to capture specific output patterns for enhancing bug report reproduction. For instance, the reproduction process may stall when encountering a repeated sequence of actions. To address these concerns, our approach uses a feedback mechanism to consistently update the LLMs with feedback on the effectiveness of their responses. This process guides them toward more accurate and relevant outputs for subsequent actions, thereby enhancing the consistency and reliability of the bug reproduction process.

Challenge 5: Incapable of handling non-crash bug reports. The wide range of non-crash bug symptoms poses a substantial challenge in bug reproduction. Existing works are primarily focused on crash bug reports [26, 52–54], or S2Rs replay without concern for automatically verifying the symptoms, such as AdbGPT [28]. Crash symptoms are often easy to identify through error messages in Logcat or UI changes, while non-crash bug symptoms are diverse and may need different test oracles for detection. [45, 50]. Given the advanced text comprehension capabilities of LLMs, we see LLMs' potential to recognize non-crash bug symptoms, although they may not be able to fully address all kinds of non-crash issues at this moment. For example, a non-crash symptom described as "See no results" can be effectively determined by LLMs based on the warning "No data" displayed on the screen. Therefore, in this paper, we also investigate whether LLMs can accurately identify content-related non-crash bug symptoms based on the UI context and the symptom described in the bug report, excluding issues related to images such as blurriness, size, and color variations.

Figure 3: ReBL Approach Overview



3 REBL Approach

REBL is a feedback-driven approach for automated *whole bug report* reproduction in Android apps utilizing the capabilities of LLMs. The architectural framework of REBL is illustrated in Figure 3. REBL is end-to-end, requiring users to input the bug report and APK file. Therefore, developers without knowledge of LLMs can conveniently utilize the tool. The ultimate objective is to generate an event sequence that precisely reproduces the reported bug. The instructions transform the general-purpose LLM into a bug reproduction tool [12, 41], adhering to our design. App UI Information offers the app's UI context. The inherent feedback mechanism of REBL is fully automated to generate feedback and provide a richer context of the reproduction process by offering additional observations related to the format of responses, UI context, or actions.

This is an iterative process. In each iteration, REBL leverages the above information to generate prompts and update the prompt history, which are then used to query the LLM for a response. Upon receiving the response, REBL interprets it and utilizes the execution engine to perform the suggested actions. Following this, it updates the feedback and app's state, informing the generation of the next prompt and updating the prompt history. This process continues until the LLM determines the reproduction process should be concluded.

The prompt history preserves all information during the reproduction process, enabling LLMs to maintain a consistent understanding and to reference any detail, such as the bug report and previous UI information, at any time. By utilizing the entire textual bug report, REBL bypasses the two traditional phases, S2R entity extraction and S2R entity matching. It directly addresses Challenge#1. It ensures that every piece of textual information in the bug report is considered. Moreover, the description of bug symptoms in the whole bug report combined with the App UI Information aids in determining whether the bug has been triggered, effectively tackling Challenge#5.

In contrast to AdbGPT, which employs prompts to inquire about precise actions and target S2R entities for each step, our approach

takes a significantly different path. REBL utilizes the LLM in a distinctive, feedback-driven manner. By providing comprehensive App UI Information and thorough feedback, the LLM is empowered to make well-informed decisions relevant to the current page. This strategy significantly enhances flexibility in bug reproduction, effectively addressing both Challenge#2, Challenge#3, and Challenge#4. Furthermore, in Challenge#3, if the input for filling a blank generates an invalid response, preventing page progression, the design of feedback can aid in correcting the input.

3.1 Instructions Description

The instructions serve as a foundational guide for the LLM in the workflow of automated bug reproduction. Although LLMs possess extensive knowledge, they lack the tailored specialization required for specific tasks such as bug reproduction. The instructions transform the general-purpose LLM into a bug reproduction tool, adhering to the design of our approach. They clearly define the *objective* and *workflow* of the task, followed by a comprehensive *explanation* of the workflow, ensuring that the LLM can perform this specialized task effectively. The structure and components of the instructions are shown in Table 1.

Equipped with this meticulous prompt instructions, REBL acquires the capability to conduct feedback-driven bug report reproduction. This includes supporting advanced actions, executing these actions, gathering UI context, interpreting the LLM response for feedback provision, and establishing criteria for termination. The prompt instructions act as a guiding beacon, steering the entire reproduction process.

COMPONENT	DETAILS
⟨Objective⟩	I need your assistance in reproducing bug reports for Android apps. Our goal is not just to follow the steps leading to where the bug occurs in the app, but also to verify that the buggy behavior specified in the bug report is indeed triggered.
⟨Workflow⟩	To initiate the reproduction process, I will provide the app name, bug report, and initial UI information. Your role will be to offer one suggestion at a time, such as clicking a button. After executing your suggestion, I will update you with feedback and the current UI state. This iterative process will continue until either triggering the bug or determining reproduction has failed.
⟨Explanation⟩	1. Available Actions: click, long_click,set_text, scroll,...; 2. Your Response Rormat:...; 3. Termination criteria: 2. Your response format should be...; 3. The condition to terminate: (a) Successful Reproduction (b) Failed Reproduction.

[1] Due to space constraints, this table, this table aims to present the structure and components of the instructions. Full details are available in [16].

Table 1: Instructions Description

3.2 Extracting App UI Information

App UI Information showcases the app's current state. It can be used to validate the effectiveness of previous actions and to help make the decision for the next step. In our design, the UI information is composed of two distinct parts: the activity name and the UI widget information.

3.2.1 Activity Name. The activity name serves as a unique identifier for the activity within the app. It offers extra context about the functionality or purpose of the current page, aiding in progress checking of bug reproduction.

3.2.2 UI Widgets Information. UI widgets showcase the app's current state. Existing reproduction approaches [28, 52, 54, 55] leverage individual UI widgets to perform entity matching between the UI widgets' identifiers (either by their resource-id, content-description, or text) and the extracted S2R entities. We have followed the same method of selecting identifiers to represent single widgets, such as [class: identifier]. If all three identifier fields are empty, then coordinates are used. However, focusing solely on individual UI widgets can lead to a lack of context, making it difficult to manage complex UI pages, such as those with too many widgets or widgets that have identical or semantically similar identifiers on the same page. It is crucial to group widgets to enhance understanding and facilitate automation tasks [48, 49], such as bug reproduction.

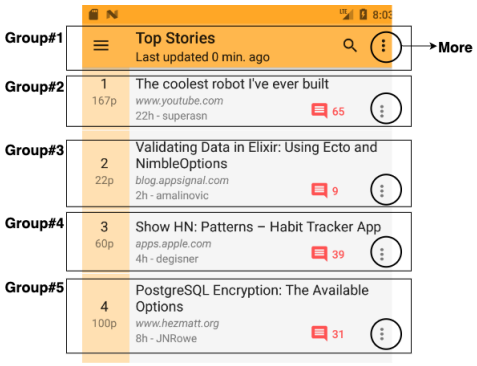


Figure 4: Illustration of Grouping

Figure 4 shows an app page featuring many widgets. Viewing these widgets in isolation makes it challenging to discern their specific functions and relationships. Even for the common widget "More" (known as "more options"), the presence of five such widgets complicates identifying their distinctions when viewed separately. However, grouping them adds organizational context, which eases the differentiation and prediction of widget relationships and functions. In this example, after grouping, there is a group containing a widget "Top Stories" followed by four uniformly structured groups. For instance, each group contains the same number of widgets, including a long text and a URL-like text, suggesting that each group represents a story. The lengthy text likely serves as the title, and the URL-like text acts as the link to that story. Furthermore, one "More" widget in the header group suggests global functionalities for the page, while the other four "More" widgets, distributed among the story groups, indicate local functionalities related to their respective story groups.

3.2.3 Grouping UI Widgets. To group UI widgets, we analyze the XML of the app page and use the layout structure to systematically group widgets. This approach primarily follows the app developer's intention to organize and contextualize them. In the XML, a *layout* is an element whose class type is ViewGroup or a subclass of ViewGroup, such as LinearLayout or FrameLayout, designed to contain and organize UI components (views) on the screen. A *clickable layout* is a layout with its clickable attribute set to true, meaning that interacting with any part of the layout triggers a response. A *child* is an element that is directly contained within another element and is exactly one level below it in the hierarchy. A *leaf* is a

child that does not have any children. A *nested layout* is a layout element and serves as a child of another layout element. Widgets on a page are organized following the format "Group #[Number]: [List of Widgets]", where each individual widget within the group maintains the format of a single widget. The rules of grouping work as follows, with the order of steps being critical:

1. For a clickable layout that contains no nested clickable layouts, all widgets within it are considered a group. This rule is adapted because interacting with any part of the layout triggers a response. Widgets within this group collectively convey the group's overall functionality.
2. If a non-clickable layout has all its children as leaves, and at least one of them is clickable, all widgets within this layout are considered a group. This rule is based on the understanding that non-clickable widgets can serve as supplementary explanations for clickable widgets in the same group.
3. If the previous rules do not apply, a clickable widget can be a group by itself. This ensures that widgets not covered by the previous two rules are also taken into consideration.

Figure 5 depicts a UI page utilized for adjusting app display settings. The layout encompassing the "Text size" and "Small" widgets constitutes a group. This classification arises from the fact that clicking anywhere within this layout will prompt the item list for font size. Following the first rule, this group is established because the clickable layout contains no nested clickable layouts. All widgets (i.e., "Text size", "Small") within this clickable layout are grouped. For the second rule, a common example is a non-clickable label next to an editable text box.

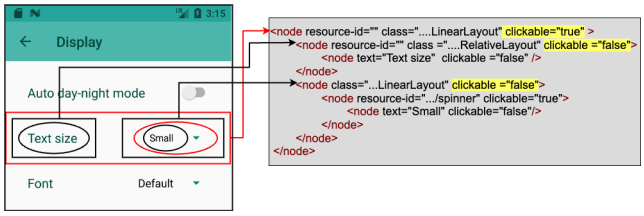


Figure 5: Grouping Example

3.3 Interpretation and Feedback on LLM Responses

3.3.1 Interpreting the Response. During each iteration of the bug reproduction process, REBL relies on the LLM's response to execute action(s) on the current page. The response can be a single action, such as [$a1$], or a sequence of actions taken in order, such as [$a1$, $a2$, $a3$]. Here, a represents a single action, each paired with its necessary components, such as the target UI widget, input value, direction, and duration. See Table 2 for the list of available actions.

Compared with existing approaches, REBL can handle multiple actions on one page in a single response, which speeds up the reproduction process, saving time from conducting multiple interactions (e.g., sending subsequent prompts and waiting for responses). This proves particularly effective when the bug report requires the selection of multiple items or filling out several text fields on the same page. Moreover, there are scenarios where rapid execution of multiple actions is needed to trigger a bug, such as "Do multiple fast clicks on Play/Stop button" [5]. In these instances, existing approaches that execute actions step by step may prove inadequate, while REBL, utilizing the multi-actions response from the LLM, is able to handle the rapid execution of a sequence of actions.

Table 2: Actions and LLM Responses Format

UI Actions		
Action a	**Required Format**	**Example**
back	[action]	['back']
click *long − click*	[action, target]	['click', 'theme']
scroll *swipe* *rotate*	[action, direction]	['scroll', 'up']
set_text	[action, target, input]	['set_text', 'name', 'joh']
System Actions		
restart	[action]	['back']
sleep	[action, duration]	['sleep', 0.5]
Termination Actions		
success	[action]	['success']
fail	[action]	['fail']
LLM Response		
Example1	[a1]	[['click ', 'A']]
Example2	[a1, a2]	[['set_text', 'email', 'conf@test.com'], ['set_text', 'password', '123456']]

3.3.2 Feedback on the Response. Our approach's design emphasizes the necessity of providing extra feedback on the LLMs' responses at every iteration. This includes (i) execution status, confirming whether the actions were successfully executed; (ii) analysis of whether actions might cause repetition or loops; (iii) observing if actions trigger quick-disappearing widgets that appear and disappear quickly, often unnoticed but might be crucial.

Action execution status. The Execution Result handler informs the LLM models of the execution status, indicating whether the previously suggested actions were executed successfully, thereby aiding in the checking of reproduction progress. Due to the probabilistic nature of language models, the LLM might occasionally produce unexpected responses. For instance, it might suggest performing an action on a UI widget that does not currently exist on the app's current page, leading to a failure in execution as the target widget cannot be located. Alternatively, the LLM might identify the correct target but output the response in an incorrect format, leading to execution failure as the response cannot be interpreted accurately. To address this, we include the execution result in the prompts. This feedback enables the LLM to acknowledge the current status, indicating whether it is appropriate to proceed to the next step or necessary to reformulate the response due to a failed execution.

Repeated sequence. The Repeated Sequence handler detects patterns where a sequence of actions has been executed at least twice, leading to a situation where the reproduction process might get stuck on the same page or enter a loop. This handling is particularly crucial because our approach is feedback-driven and does not ask the LLMs to match specific entities with UI widgets. Consequently, there is an inherent potential for the process to become stuck on a page, necessitating the need for this handler to monitor the history of actions, providing crucial oversight to prevent repetitive loops and ensure a smoother reproduction process. Our algorithm checks if the newly suggested action(s) cause any sequence of actions to repeat from some point until the new actions in each iteration. For example, if the current action history is A → B → C → D → B

→ C, and the LLM suggests D as the next action, then a repeated sequence is detected: B → C → D → B → C →D. When a repeated sequence is detected, we remind the LLM models. This reminder helps the LLM decide if ReBL should avoid these repetitions in future explorations.

Quick-disappearing widgets. A quick-disappearing widget refers to a widget that appears in the UI when it is relevant, often following the execution of an action, but then disappears quickly. Common examples include pop-up notifications, toast, and data-loading dialogs. Existing approaches typically involve a brief waiting period (e.g., 5 seconds) after an action to gather information from the stable UI page, frequently neglecting the existence of quick-disappearing widgets. However, quick-disappear widgets can be crucial in various aspects of bug reproduction. First, quick-disappearing widgets are crucial for providing information for S2Rs. As exemplified in Challenge 2 in Section 2.2, a quick-disappearing toast provides critical information for choosing the correct action. Second, they act as termination indicators, particularly when bug symptoms, such as error messages, are presented as quick-disappearing toast messages. Third, there are situations where the target of reproduction steps is a quick-disappearing widget. To address this challenge, ReBL is adept at considering the presence and context of quick-disappearing UI widgets, thereby effectively managing scenarios that involve this kind of widget and enhancing the accuracy of the bug reproduction process.

3.4 Handling Token Limit

The token limit in LLMs restricts the total amount of text that can be included in both the prompt and the model's response, highlighting the necessity for effective management strategies. For instance, GPT-4 offers options for specifying token limits — 8K and 32K, depending on user preference and budget availability [15]. In our experiment, we opted for a token limit of 8K.

Our approach employs summarization to address the token limit. With the max token limit (L), ReBL continuously monitors the token count of the prompt history, denoted as C. When C surpasses a set threshold ($C > L \times TH$ with TH being the threshold proportion, e.g., 0.7), ReBL queries the underlying LLM to condense the current prompt history. Due to the robust semantic comprehension of LLMs, this method preserves crucial information while significantly reducing the size of prompt history without losing context and eliminating less relevant details for assisting the bug reproduction. To the best of our knowledge, none of the existing literature has explicitly mentioned specific strategies for addressing the issue of token limits. Our experiment validates the effectiveness of the summarization strategy.

3.5 Termination

The reproduction process terminates, signaling either a successful or failed reproduction.

3.5.1 Successful Reproduction. The successful reproduction of a bug report is defined by the manifestation of the reported bug symptom. ReBL leverages the underlying LLM to assess whether a bug has been triggered, utilizing the bug symptom outlined in the report, in conjunction with the current UI information (Section 3.2) and the prompt history. The actions executed in the prompt history are used to ensure the mentioned steps in the bug reports have been executed, while the current UI information is employed to verify the presence of bug symptom mentioned in the bug report. For example, ReBL effectively resolves a *non-crash* bug report reproduction related to an invalid search. This was accomplished by utilizing the LLM's capability to identify the *non-crash* symptom described by the reporter in the bug report as "See no results after search," and subsequently correlating this symptom with the content "No data" displayed on the current UI page. The bug arose due to the expected search yielding no valid results. This termination strategy also applies to *crash* bugs, where the symptoms are readily apparent.

3.5.2 Failed Reproduction. Failed reproduction occurs when ReBL fails to trigger the described bug. In such cases, ReBL continues to explore the application and makes repeated attempts to reproduce the bug. This persistent exploration often results in surpassing the token limit of the prompt history as information accumulates over time. Consequently, token limit control mechanisms (Section 3.4) are activated. To optimize resource use and limit endless exploration, failure is determined either after a specified duration (e.g., one hour) or upon reaching a predefined token summarization threshold (e.g., three times). The one-hour time constraint is aligned with the settings commonly used in state-of-the-art approaches [52–54]. Our empirical study sets the token summarization threshold at three because we found that typically, one summarization is sufficient to free up space for continued reproduction, leading to successful outcomes. Therefore, we chose 3 attempts to ensure a sufficient number of opportunities. For cases that fail to reproduce after three attempts at summarization, the primary reasons for the failures are not due to the token limit, but rather factors such as the underlying tool's capabilities or insufficient information in the original bug report, as detailed in Section 5.1.

4 Empirical Study

To evaluate ReBL, we address three key research questions:

RQ1: How effective and efficient is ReBL at reproducing bug reports?

RQ2: How do individual components contribute to the overall effectiveness and efficiency of ReBL?

RQ3: How does ReBL's effectiveness and efficiency in reproducing bug reports compare to that of three baseline approaches?

4.1 Datasets

To collect datasets, we adopted the established practice for gathering real-world bug reports for bug reproduction [26, 33, 46, 52, 54, 55]. Specifically, our dataset integrates evaluation datasets from four state-of-the-art tools: AdbGPT [28], ReproBot [52], ReCDroid/ReCDroid+ [54, 55], and Yakusu [26], and AndroR2 [46], a dataset of manually-reproduced Android bug reports, and an empirical study on Android bug report reproduction [33]. We refined our dataset by excluding duplicates, reports related to inaccessible or non-installable APK files, and reports no longer reproducible (e.g., server issues, invalid login). This process resulted in a concise set of 96 unique bug reports, of which 73 are crash reports and 23 are non-crash reports.

4.2 Implementation

We conducted our experiment on a physical x86 machine running Ubuntu 16.04, equipped with an i7-4790 CPU @ 3.60GHz and 32 GB of memory. Notably, this machine did not have a GPU. For gathering GitHub issues, we utilized the GitHub REST API crawling [13]. However, for issues present on other platforms like F-Droid, we relied on BeautifulSoup [2] for data crawling. Our approach, ReBL, integrates the underlying language model GPT-4 [15]. To facilitate interaction with UI widgets on the device, we implemented UI Automator2 [9] as our execution engine. We executed our approach three times to ensure the robustness and consistency of results, and we calculated the average for measuring the execution times. The implementation of our approach is publicly available, along with the experiment data [16].

4.3 Study Design

4.3.1 RQ1: How effective and efficient is ReBL at reproducing bug reports? Effectiveness is determined by the ratio of successful reproductions to the total number of bug reports examined. Additionally, we assess the effectiveness of reproducing crash and non-crash bug reports separately. Efficiency, on the other hand, is measured by the average time taken for successful reproductions. (Refer to Section 3.5 for the criteria of successful reproduction.) The evaluation is conducted within a one-hour timeframe, with the summarization threshold set to three times—a decision explained in Section 3.5.2. Reproductions that exceed this time limit or summarization threshold are considered a failure. To ensure the accuracy of our results and avoid false positives, we conducted a manual inspection to confirm whether the described crash or non-crash bug symptom occurs on the specified target page once ReBL terminates after executing the given S2Rs in the bug report.

4.3.2 RQ2: How do the individual components enhance ReBL's overall effectiveness and efficiency? We conducted an ablation study to systematically evaluate the impact of individual components on the functionality and performance of ReBL by comparing it against a fully functional version. The study includes the following three ablations: 1) $\textbf{ReBL}_{\textbf{S2R}}$ uses only S2Rs provided in the bug report, excluding title and other details beyond S2R segment, unlike the full version, which considers the entire textual report; 2) $\textbf{ReBL}_{\textbf{Indiv}}$ views UI widgets individually, while the full version groups widgets to process UI information; and 3) $\textbf{ReBL}_{\textbf{NoFB}}$ retains the simple guidelines in the instructions and does not incorporate the feedback mechanism.

4.3.3 RQ3: How does ReBL's effectiveness and efficiency in reproducing bug reports compare to that of three baseline approaches? We have established three baselines: three state-of-the-art automated bug reproduction approaches: AdbGPT [28], ReproBot [52], and ReCDroid [55]. Throughout our evaluation, we set a time limit of one hour for all techniques to complete the bug reproduction process. This allowed us to assess their performance under consistent conditions and determine their effectiveness and efficiency in reproducing bugs within a reasonable timeframe. Manual inspection is also integral to validate the results, ensuring the reliability of our comparative analysis.

5 Results and Analysis

5.1 RQ1: Effectiveness and Efficiency of ReBL

5.1.1 Effectiveness. ReBL successfully reproduced 87 out of 96 bugs, including 69 crash bugs of 73 (94.52%) and 18 non-crash bugs out of 23 (78.26%), achieving an impressive overall success rate of 90.63%. The details of bug reproductions are shown in [16]. This performance highlights ReBL's robustness and versatility in reproducing bug reports. Within the 8k token limit constraint, the summarization mechanism was activated for 11 bug reports to manage token constraints. Among these cases, two [1, 8] triggered this mechanism, resulting in successful bug reproduction that would have otherwise failed. However, the remaining nine reports also activated summarization but failed to reproduce the bug due to reasons other than the token limit.

Reasons for failed reproductions. Out of the nine cases where ReBL failed to reproduce the bugs, four were crash bug reports and five were non-crash bug reports. For the crash bug reports, we identified the following reasons for the failed reproductions:

First, the inability to reproduce bug report when involving third-party services, such as Google Drive. ReBL lacks the capability to navigate between different apps, which limits its effectiveness in cases like "ODK-360" [3], where interaction with Google Drive is essential.

Second, the limitation of ReBL's underlying testing framework, UI Automator2, appears to be particularly evident in specific apps. It fails to extract custom views from the hierarchy, preventing ReBL from accessing the necessary UI widgets to reproduce the specified bug. A notable example of this is "Memento-169" [7]. Another critical issue is the framework's limited ability to execute certain actions. For example, in "osmeditor-637" [4], the framework struggled with *set_text*, leading to a failure in the reproduction. These issues may be specific due to the compatibility of our underlying testing framework rather than a widespread problem.

Third, a severe lack of information significantly impedes accurate bug reproduction, as exemplified by the "Anki-6432" case [10], where the bug report omitted 20 out of the 26 required steps [52]. ReBL struggles to identify the bug due to extremely insufficient information in the bug report. Future improvements could include using static analysis [30, 51] to gain comprehensive domain knowledge about the app. This could potentially improve LLMs' understanding and enable more precise predictions, even with limited information in bug reports.

For the five non-crash bug reports, ReBL faces a unique challenge due to their subtler symptoms compared to crash bugs. This subtlety might lead to false conclusions that a bug has been triggered. For example, in the bug report "LrkFM-34" [6], the S2Rs are outlined as follows: "1. Move any file, 2. Try to paste the file, and 3. Observe that nothing happens." However, ReBL fails to reproduce the actual bug because it performs the "move" (essentially cutting a file from one location) and "paste" actions within the same folder. Although this results in "nothing happens" within the same folder – a symptom that seems to match the bug report – the actual issue involves failing to paste the item into a different folder, where no files are added after the paste action.

Successfully reproduced non-crash bug reports. Figure 6 illustrates some non crash bug scenarios adeptly addressed by ReBL.

First, leveraging the semantic capabilities of the underlying LLM, REBL excels in *recognizing textual nuances and correlations*. Figure 6-A illustrates an example where the bug report provides an error message. REBL identifies that the bug is triggered through the association between the given message and the actual error message. Another example, as introduced in earlier section, REBL can associate the "See no results after search" symptom and the "No data" text displayed.

Second, REBL can *verify the existence of widgets and their states*. As demonstrated in Figure 6-B, REBL can determine the presence of widgets of birthday year and verify the state of the checkbox to confirm whether the bug has been triggered. Similarly, Figure 6-C shows REBL handling mismatches between widget types and dialogue selections.

Third, REBL excels in *comparing changes across pages* by utilizing the historical data in prompt history. Figure 6-E showcases a bug, the symptom of which is "sound remaining unchanged." REBL identifies this bug symptom by accessing and comparing the previous sound settings from the prompt history with the current settings. Likewise, the bug symptom in Figure 6-F, "nothing happens in the list," requires analysis of the UI's state before and after the action.

5.1.2 Efficiency. REBL showcases an impressive level of speed in bug reproduction. The time required to reproduce the bugs varied between 19.99 to 243.3 seconds, with a low average time of 74.98 seconds. Notably, the bulk of this time is spent interacting with the GPT model, such as making API calls and waiting for responses. In contrast, the processes of action execution, feedback collection, and prompt generation are extremely swift, each taking less than 0.5 seconds.

> **RQ1:** REBL successfully reproduced 90.63% of the 96 bug reports with each bug report taking an average of 74.98 seconds. Specifically, it achieved a 94.52% (69/73) success rate for crash bug reports and 78.26% (18/23) for non-crash bug reports, demonstrating efficient and adaptable bug reproduction capabilities.

5.2 RQ2: The Roles of Individual Components

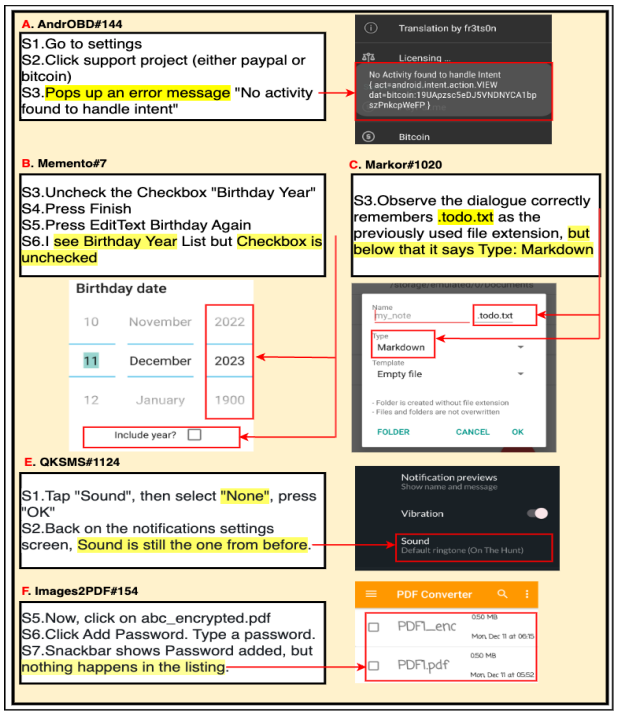
Table 3 shows the results of the three ablations compared with the fully functional REBL.

REBL$_{S2R}$ achieves an overall success rate of 81.25%, including 90.41% (66/73) for crash reports and 52.17% (12/23) for non-crash bug reports. This overall success rate is 9.38% lower than that of REBL. The performance of REBL$_{S2R}$ is impacted by the insufficient information when considering only S2Rs. This shortfall is especially severe for non-crash bug reports because symptoms of non-crash bugs are often found in the title or observed behavior sections, leading to greater oversight. Lacking the symptom of a non-crash bug makes it impossible to verify its occurrence.

	REBL_S2R	REBL_Indiv	REBL_NoFB	REBL
Effectiveness	81.25%	77.08%	73.96%	**90.63%**
Efficiency	75.50s	78.23s	87.16 s	**74.98s**

Table 3: Ablation study

Figure 6: Examples of None-Crash Bug Reports



REBL$_{Indiv}$ achieves a success rate of 77.08%, which is 13.55% lower than REBL. REBL$_{Indiv}$ excels when target widget identifiers are clear, similar to existing tools that perform S2R entity match. However, it struggles when the page contains numerous or semantically similar widgets. For example, "open the context menu for an item" targets a group of widgets comprising title and URL, rather than individual ones. Section 3.2 details limitations regarding individual widgets.

REBL$_{NoFB}$ reports a success rate of 73.96%, reflecting a reduction of 16.67% compared to REBL. Its primary challenge is the absence of detailed instructions. Without explanations for actions, REBL$_{NoFB}$ may misinterpret actions in the instructions. For instance, "select A" in S2R might be considered a suggestion for "select" rather than "click," which is the correct action. Furthermore, the lack of feedback exacerbates the issue, as there is no system to indicate failure when attempting to execute "select."

Efficiency remains consistent across both ablated and fully featured versions, as the absence of certain details—such as non-S2R information in REBL$_{S2R}$ or differences in format—such as widget format in REBL$_{Indiv}$—does not significantly alter prompt size or complexity, thereby not impacting the LLM's processing time.

> **RQ2:** REBL significantly outperformed its three ablated versions, emphasizing the necessity of its full feature set for optimal performance. This highlights the importance of using the whole textual bug report to furnish more comprehensive information, grouping widget to provide structured and organized UI context, providing detailed instructions to ensure smooth interaction with LLMs and implementing a feedback mechanism enables timely measures when unexpected responses occur.

5.3 RQ3: Comparison with the State-of-the-Art Approaches

Table 4 presents the overview of the comparison results, comparing our approach with three state-of-the-art baselines across a dataset of 73 crash bug reports.

	ReCDroid	ReproBot	AdbGPT	REBL
Effectiveness	45.21%	65.75%	73.97%	**94.52%**
Efficiency	534.92s	413.72s	89.80s	**72.11s**

Table 4: Comparison with Baselines

5.3.1 Effectiveness. As shown in Table 4, REBL successfully reproduces 69 crash bug reports, surpassing the numbers achieved by ReCDroid, ReproBot, and AdbGPT, which reproduce 33, 48, and 54 bug reports, respectively. For bug reports where REBL successfully reproduces while the three state-of-the-art tools fail to address, we have conducted a thorough analysis of each case. Our findings reveal four main reasons for these failures, aligning with the challenges we outlined in the motivation section (Section 2.2). It is critical to recognize that often, it is usually not a single isolated reason leading to the failure, but rather a combination of them.

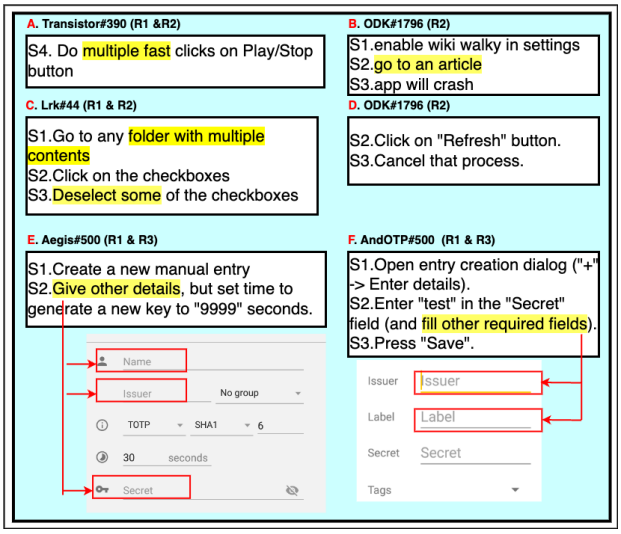
Reason#1: Overlooking non-S2R information. In Challenge 1 (Section 2.2), we provided an example where the S2R Entity Extraction phase omits crucial non-S2R information needed for bug reproduction. Another example that highlights this issue is [5], where one S2R states: "Add URL with the stream". However, the original post does not contain the specific URL; instead, it is provided as a comment in a different section. Baseline approaches that rely solely on S2R are unable to associate the provided URL.

Reason#2: Limitation of S2R entity extraction. The S2R Entity Extraction phase sometimes falls short in extracting precise reproduction steps due to its reliance solely on bug report content, neglecting actual context encountered during reproduction. Refer to Figures 7-A and C. They contain ambiguous S2R that make entity extraction challenging. It is challenging to identify the exact number of actions and targets without the context of the actual UI context because "select some checkboxes" and "multiple" lack specificity. Similarly, "Given other details" in Figure 7-E and "fill other required fields" in Figure 7-F face the same limitation.

Reason#3: Challenges in handling incomplete and ambiguous S2Rs. As demonstrated in Challenge 2 of the motivating example (Section 2.2), we provided an example to underscore this limitation. The experimental results showcase similar instances, primarily attributed to the limited context-awareness of UI information and an inability to handle rapid UI actions.

(i) Limited context-aware: In Figure 7-B, the second step "go to an article" implies the need to leave the settings page and return to the home page to find an article. Existing methods strictly follow sequential steps on the current page, attempting to locate a UI widget within the settings page labeled "article". However, they may miss the need to navigate elsewhere. If unsuccessful on the current page, these methods typically explore widgets on the same page, rarely considering navigating to a different page unless explicit heuristics guide them. In contrast, REBL considers the broader UI context, including the current page and navigational history, while recognizing the current reproduction progress. It prioritizes context-aware actions to achieve the goal of "go to an article," rather than rigidly focusing on finding a specific S2R entity, "article," on the current page. *(ii) Quick actions:* Quick actions are common in using mobile apps and can potentially trigger bugs, as illustrated in Figures 7-A and C. In Figure 7-C, triggering the bug requires "multiple fast clicks", presenting a scenario where traditional single-action-per-step approaches fall short. These approaches execute one action per iteration, often followed by a waiting period (e.g., 5 seconds) for UI stabilization. Additionally, processing the next step, such as AdbGPT [28] querying LLMs, can take 3-10 seconds, and NLP matching approaches also require time. This time gap between actions hinders the triggering of bugs requiring rapid, consecutive interactions, like multiple clicks without interruption. Figure 7-C depicts a concurrency bug where the "refresh" button is part of a rapidly disappearing loading dialog. To trigger this bug, detecting the quick-disappearing widget "Cancel" and successfully clicking on it are necessary, requiring immediate quick action. Our approach, monitoring UI pages to detect rapidly disappearing widgets and having the flexibility to perform more than one action per step, is well-suited for handling such bug scenarios.

Figure 7: Examples of Other Approaches' Failure Cases



Reason#4: Fail to generate valid input. As noted in Challenge 3 of the motivation section (Section 2.2), the inability to generate valid input has been a significant issue in existing works. Consider Figures 7-E and F, where even when the three state-of-the-art tools overcome the above limitation and recognize the need to fill in these blanks, they often input invalid input. This is primarily because these tools adapt simpler methods for filling text fields when explicit inputs are absent in the S2R. Consequently, they lack the capability to generate context-specific inputs and are unable to correct invalid inputs when warned.

5.3.2 Efficiency. Regarding efficiency, REBL demonstrates a significant advantage, with an average reproduction time of 72.11 seconds per bug report. In comparison, ReproBot, ReCDroid, and AdbGPT exhibit considerably longer average times of approximately 413.72 seconds, 534.92 seconds, and 89.80 seconds, respectively. Thus, REBL is 7.42 times faster than ReCDroid, 5.74 times faster than ReproBot, and 1.25 times faster than AdbGPT in reproducing bug reports.

RQ3: Our analysis of 73 crash bug reports shows that REBL significantly outperforms ReCDroid, ReproBot, and AdbGPT in effectiveness and efficiency. Specifically, their success rates are 45.21%, 65.75%, and 73.97% respectively. REBL outperforms them with a remarkable success rate of 94.52%. In terms of efficiency, REBL reproduces bug reports in an average time of 72.11 seconds, which is 7.42 times faster than ReCDroid, 5.74 times faster than ReproBot, and 1.25 times faster than AdbGPT in reproducing bug reports.

6 Threats to Validity

The primary external validity concern in this study revolves around the representativeness of the apps, bug reports, and tools utilized. In our evaluation, we aimed to create realistic settings using real bug reports and Android apps. The emulator and execution engine (UI Automator) are widely adopted in both industry and academia, consistent with other Android testing works [28, 44, 52, 54]. Furthermore, existing approaches (e.g.,ReCDroid [55] , AdbGPT [28]]) have demonstrated the effectiveness of such automated reproduction tools over manual reproduction by real-world developers. However, we acknowledge that our results may not be fully generalizable to all bug reports in different domains. Additionally, the relatively small number of non-crash bug reports presents an additional constraint, potentially impacting the breadth of our conclusions.

Regarding internal validity, a notable threat arises from the inherent randomness in the responses generated by LLMs. To address this concern, we ran our experiments three times, thereby reducing the impact of random variations. However, it is essential to recognize that complete consistency in results cannot be guaranteed in all instances. REBL utilizes GPT-4 as the underlying LLM implementation. While other LLMs [21, 24, 37, 42] could be employed, variations in their design and training data may result in different performance outcomes, potentially impacting REBL's effectiveness and accuracy. In the future, we plan to systematically examine the actual impact of various LLMs on our approach.

7 Related Work

Automated Bug Reproduction. As discussed in Sections 1 and 2.2, there are existing approaches that specifically target automatically reproducing Android bug reports, including Yakusu [26], ReCDroid/ReCDroid+ [54, 55], MACA [36], DroidScope [31], and ReproBot [52]. The most recent work, AdbGPT [28], uses LLMs to extract S2R entities for guiding bug report reproduction. However, similar to other existing techniques, AdbGPT's employment of the traditional two-phase structure—consisting of the S2R Entity Extraction phase and the Entity Matching phase—suffers from the same limitations as described in Section 2.2. Our results demonstrate that REBL outperforms AdbGPT.

There are other works that approach bug reproduction from different aspects, such as recording and replaying bugs [18, 19, 27, 29, 40], analyzing stack traces [32], and leveraging the call stack [47]. Among these, GIFdroid [27] utilizes screen recordings to automate bug reproduction by adopting image processing techniques. CrashTranslator [32] reproduces crash reports directly from stack traces by leveraging a pre-trained LLM to predict the steps necessary for reproduction.

Bug Report Study. There have been several research efforts dedicated to studying and analyzing Android bug reports. For instance, Johnson et al. [33] conducted an empirical study on 180 Android bug reports to examine their reproduction challenges and the quality of reported details. Chaparro et al. [23] conducted an empirical study on user-reported behaviors, reproduction steps, and expected behaviors, identifying discourse patterns used by reporters. Chaparro et al. also developed Euler [22], an automatic technique to assess the quality of S2R in Android bug reports, using simple grammar patterns. Liu et al. introduced Maca [36], a machine learning-based classifier that categorizes action words of S2R into standard categories. However, these techniques all focus on improving the accuracy of identifying S2Rs.

Some research aims to facilitate the reporting process. For example, Fusion, developed by Moran et al. [39], employs dynamic analysis to obtain UI events of the app to enhance bug reports during testing. Additionally, Fazzini et al.[25] assist reporters in writing more accurate reproduction steps using information from the static and dynamic analysis of the app to predict the next step. Also, Yang et al. [43] provide a guided reporting system with instant feedback and graphical suggestions to improve the quality of bug reports. These approaches improve bug report quality. Though not aimed at reproduction, the improvement in bug report quality could potentially enhance LLMs' comprehension in bug reproduction.

LLMs in Analyzing Bug Reports. There has been some work on using LLMs to analyze and understand bug reports. Lee et al. [35] use LLMs to analyze bug reports for bug triage. Messaoud et al. [38] use the BERT model for duplicate bug report detection. Kang et al. [34] propose an approach to use LLMs to generate test methods for Java programs from bug reports. This approach focuses on JUnit tests, which differ from Android UI testing that requires different modeling and iterative exploration processes.

8 Conclusions

In conclusion, REBL is an advanced automated approach for reproducing both crash and non-crash bug reports in Android apps. Leveraging GPT-4 and well-designed prompts, REBL interacts effectively with the GPT model for bug reproduction. Our evaluation, conducted on 96 bug reports from 54 Android apps, showcases REBL's proficiency in successfully reproducing 87 bug reports in an average time of 74.98 seconds, surpassing three existing tools in success rate and efficiency. REBL stands out as a lightweight and streamlined solution, offering developers a powerful tool to tackle bug reports efficiently and effectively. In the future, we aim to enhance REBL's performance in handling a wider range of non-crash bug symptoms, potentially through static analysis and fine-tuning. Another goal is to expand REBL's capabilities to tackle more complex and ambiguous S2Rs to solve increasingly intricate scenarios.

9 Acknowledgments

This work was supported in part by U.S. National Science Foundation (NSF) under grants CCF-2402103, CCF-2403617, CCF-2403747, CCF-2342355, CCF-2211454.

References

[1] 2016. AIMSICD-816. https://github.com/CellularPrivacy/Android-IMSI-Catcher-Detector/issues/816.

[2] 2016. Beautiful Soup Documentation. https://tedboy.github.io/bs4_doc/.

[3] 2017. ODK-360. https://github.com/getodk/collect/issues/360.

[4] 2017. Osmeditor-637. https://github.com/MarcusWolschon/osmeditor4android/issues/637.

[5] 2017. transistor-149. https://github.com/y20k/transistor/issues/149.

[6] 2018. lrkFM-34. https://github.com/lfuelling/lrkFM/issues/34.

[7] 2018. Memento-169. https://github.com/alexstyl/Memento-Calendar/issues/169.

[8] 2019. Fdroidclient-1821. https://gitlab.com/fdroid/fdroidclient/-/issues/1821.

[9] 2019. UI Automator2. https://github.com/openatx/uiautomator2.

[10] 2020. Anki-6432. https://github.com/ankidroid/Anki-Android/issues/6432.

[11] 2020. APPLAUSE. https://www.applause.com/blog/app-abandonment-bug-testing.

[12] 2022. Aligning language models to follow instructions. https://openai.com/research/instruction-following.

[13] 2022. GitHub REST API documentation. https://docs.github.com/en/rest.

[14] 2022. Introducing ChatGPT. https://chat.openai.com.

[15] 2022. Models -OpenAI API. https://platform.openai.com/docs/models/overview.

[16] 2024. Replication Package. https://github.com/datareviewtest/ReBL.

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

[18] 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.

[19] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 309–321.

[20] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 308–318.

[21] 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.

[22] 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*. ACM, Tallinn Estonia, 86–96. https://doi.org/10.1145/3338906.3338947

[23] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 396–407. https://doi.org/10.1145/3106237.3106285

[24] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[25] Mattia Fazzini, Kevin Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2022. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1246–1272.

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

[27] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.

[28] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[29] 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)*. IEEE, 72–81.

[30] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. 2020. Improving automated GUI exploration of android apps via static dependency analysis. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 557–568.

[31] Yuchao Huang, Junjie Wang, Zhe Liu, Song Wang, Chunyang Chen, Mingyang Li, and Qing Wang. 2023. Context-aware Bug Reproduction for Mobile Apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2336–2348.

[32] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[33] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Honolulu, HI, USA, 321–322. https://doi.org/10.1109/SANER53432.2022.00048

[34] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[35] Jaehyung Lee, Kisun Han, and Hwanjo Yu. 2022. A Light Bug Triage Framework for Applying Large Pre-trained Language Model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–11.

[36] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated Classification of Actions in Bug Reports of Mobile Apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 128–140. https://doi.org/10.1145/3395363.3397355

[37] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[38] Montassar Ben Messaoud, Asma Miladi, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Lobna Ghadhab. 2022. Duplicate bug report detection using an attention-based neural language model. *IEEE Transactions on Reliability* (2022).

[39] 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.

[40] 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.

[41] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[43] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 344–356.

[44] Dingbang Wang, Yu Zhao, Lu Xiao, and Tingting Yu. 2023. An Empirical Study of Regression Testing for Android Apps in Continuous Integration Environment. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.

[45] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 434–446.

[46] 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.

[47] 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.

[48] Shuhong Xiao, Yunnong Chen, Yaxuan Song, Liuqing Chen, Lingyun Sun, Yankun Zhen, Yanfang Chang, and Tingting Zhou. 2024. UI semantic component group detection: Grouping UI elements with similar semantics in mobile graphical user interface. *Displays* (2024), 102679.

[49] Mulong Xie, Zhenchang Xing, Sidong Feng, Xiwei Xu, Liming Zhu, and Chunyang Chen. 2022. Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 332–343.

[50] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1319–1331.

[51] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25 (2018), 833–873.

[52] 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. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 411–422.

[53] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *International Conference on Software and Systems Reuse*. Springer, 100–111.

[54] 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.

[55] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.