Automated Recognition of Buggy Behaviors from Mobile Bug Reports

ZHAOXU ZHANG, University of Southern California, USA
KOMEI RYU, University of Southern California, USA
TINGTING YU, University of Connecticut, USA
WILLIAM G.J. HALFOND, University of Southern California, USA

Bug report reproduction is a crucial but time-consuming task to be carried out during mobile app maintenance. To accelerate this process, researchers have developed automated techniques for reproducing mobile app bug reports. However, due to the lack of an effective mechanism to recognize different buggy behaviors described in the report, existing work is limited to reproducing crash bug reports, or requires developers to manually analyze execution traces to determine if a bug was successfully reproduced. To address this limitation, we introduce a novel technique to automatically identify and extract the buggy behavior from the bug report and detect it during the automated reproduction process. To accommodate various buggy behaviors of mobile app bugs, we conducted an empirical study and created a standardized representation for expressing the bug behavior identified from our study. Given a report, our approach first transforms the documented buggy behavior into this standardized representation, then matches it against real-time device and UI information during the reproduction to recognize the bug. Our empirical evaluation demonstrated that our approach achieved over 90% precision and recall in generating the standardized representation of buggy behaviors. It correctly identified bugs in 83% of the bug reports and enhanced existing reproduction techniques, allowing them to reproduce four times more bug reports.

CCS Concepts: • **Software and its engineering → Maintaining software**.

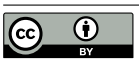
Additional Key Words and Phrases: Mobile Bug Report Reproduction, Buggy Behavior Recognition

ACM Reference Format:
Zhaoxu Zhang, Komei Ryu, Tingting Yu, and William G.J. Halfond. 2025. Automated Recognition of Buggy Behaviors from Mobile Bug Reports. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE100 (July 2025), 24 pages. https://doi.org/10.1145/3729370

1 Introduction

Bug reporting is an important mechanism for mobile app developers to receive user feedback and address bugs in their apps. Upon receiving a bug report, mobile app developers need to first analyze and manually reproduce the bug so they can identify the underlying faults. An accurate and efficient reproduction is therefore crucial for developers to efficiently locate the faulty code and fix the issue [32, 80]. However, recent studies have found that mobile app bug reports are numerous and generally have low quality writing [34, 35, 59]. This can make manual reproduction time-consuming and error-prone, slowing down the debugging progress. Therefore, automated techniques to assist developers in reproducing bugs are of high importance.

Authors' Contact Information: Zhaoxu Zhang, University of Southern California, Los Angeles, USA, zhaoxuzh@usc.edu; Komei Ryu, University of Southern California, Los Angeles, USA, eryu@usc.edu; Tingting Yu, University of Connecticut, Storrs, USA, tingting.yu@uconn.edu; William G.J. Halfond, University of Southern California, Los Angeles, USA, halfond@usc.edu.



ACM 2994-970X/2025/7-ARTFSE100
https://doi.org/10.1145/3729370

To address the need for improving the bug report reproduction process, several techniques [39, 41, 75, 76, 78] have been introduced by Software Engineering researchers to automate the reproduction process. To reproduce a bug report, these techniques continuously execute User Interface (UI) events in the app to reproduce the steps in the report (such as "click button"). Such a process terminates once the target bug is triggered on the mobile device. Assuming the correct UI events can be found and executed, existing works' capabilities are still limited by their mechanisms for recognizing the buggy behavior (BB) of the target bug. Specifically, to recognize the BB, current reproduction techniques search for a predefined log exception message related to the target bug in the device console. This requires a human tester to specify the exception message in advance, which limits their capability to fully automate the reproduction process. In addition, due to this limitation, current approaches can only reproduce bugs with exception messages specified in bug reports, which are usually crash bugs. However, as our study on mobile bug reports in Section 3 reveals, not all crash bug reports include the error message, and many reports describe non-crash bugs that do not produce an exception.

Automating the task of recognizing the BB requires addressing several challenges. The first of these challenges is to identify where in the provided bug report the user-provided description of the BB exists. This is challenging because, although bug reporting systems encourage good structure, bug reports are generally unstructured, and the description of the BB is commonly embedded alongside other details, such as reproduction steps and user commentary [34, 35, 46]. Assuming the description of a BB could be identified, an automated approach must be able to account for the broad diversity in the ways BB can manifest and be described by bug reporters. As found in related work [71] and our own study in Section 3, BB can include crashes or non-crashes, involve different subsystems of the OS (e.g., volume or keyboard), be based on the behaviors of the widgets in the UI, involve comparisons across various app states, or comprise any combination of these behaviors.

In this work, we introduce a novel approach for automatically recognizing the buggy behavior during the automated reproduction process. As a first step, to understand how BBs manifest in mobile devices and how they are described in bug reports, we conducted an empirical study of 380 bug reports. From these findings, we designed a standardized representation, BugIR, to abstract and model the manifestations of BBs. At a high level, given a bug report, our approach first identifies and converts the BB description to a BugIR expression. Then, during the automated reproduction process, our approach matches the BugIR expression with the real-time UI and device information to determine if the BB is exhibited by the app, thus automating the recognition of BB. To evaluate our approach, we implemented it into a research prototype, BugSpot, and conducted experiments on 87 real-world bug reports. Shown by the evaluation results, BugSpot was able to effectively capture the BB information using BugIR by achieving a 91% precision and 90% recall, and BugSpot correctly recognized the bugs in 83% of the bug reports. By combining BugSpot with existing reproduction techniques, we showed that BugSpot could improve their reproduction performance by enabling them to reproduce more types of bug reports.

In summary, this paper has made the following contributions:

(1) We conducted an empirical study on the buggy behavior descriptions in mobile app bug reports. Our study identified useful characteristics of buggy behavior descriptions and provided guidance on how to recognize the bugs.
(2) Based on the empirical findings, we defined a standardized representation, BugIR, to abstract and normalize buggy behavior information.
(3) We designed and implemented an automated approach to analyze bug reports and recognize the buggy behaviors during the reproduction process. We made our implementation publicly available to facilitate future research [1].

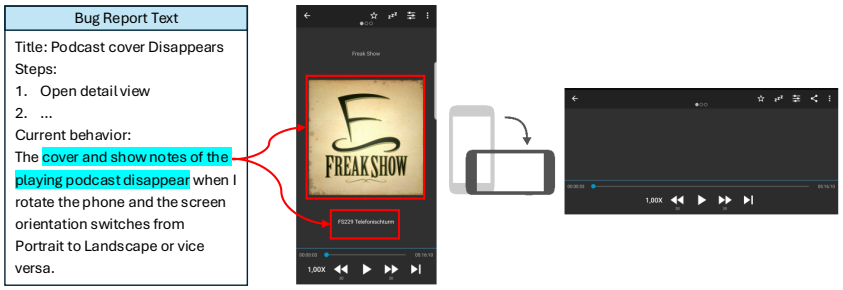


Fig. 1. An example of a bug report and the corresponding buggy behavior in the app.

(4) We conducted an empirical evaluation of our approach. The evaluation results showed its effectiveness in recognizing bugs and usefulness during the automated reproduction process.

2 Motivating Example

In this section, we introduce an example that we will use throughout the paper to illustrate the limitation of existing reproduction techniques in recognizing BBs during the automated reproduction process. Figure 1 presents a real-world bug report [8] detailing an issue with a music player app. This bug occurs when the user rotates the screen while playing music, resulting in the disappearance of two widgets—the "cover" and "show notes," which are highlighted in the red box. To reproduce the bug report, existing methods [39, 76, 78] must explore the application's UI to identify UI events that may trigger the bug. These methods conclude their exploration when the target bug is detected on the device. However, existing approaches are only able to identify crash bugs, as indicated by the content of their papers and by their evaluations, which only consider crash bugs. Consequently, they are unable to verify the bug presented in the example, resulting in an unsuccessful bug reproduction. Our study in Section 3 shows that over 80% of bug reports for mobile apps are non-crashes. Consequently, existing techniques cannot reproduce these reports automatically, making this an important issue to address. This motivated us to develop a technique that can automatically recognize various mobile app bugs, including both crashes and non-crashes, based on the bug report.

3 Empirical Study on Buggy Behaviors Descriptions

The goal of this study is to understand how the buggy behaviors (BBs) are described in mobile bug reports and manifest on the mobile device. The study results directly motivate the design of our detection approach in Section 4. To achieve this goal, we answer the following Research Questions (RQs). **RQ 1**: *What types of mobile app bugs are included in our dataset?* This RQ categorizes the bugs in our dataset and helps us understand the bugs better. **RQ 2**: *What are the commonly-used information modalities for describing BB?* This RQ investigates the ways that BB information is presented in the bug reports (e.g., text or video) and motivates the design decision of which information modality the detection tool should support. **RQ 3**: *Where in the bug report do reporters describe the BB?* This RQ explores the location in the bug report where users describe BB. The answer to this question helps determine which part of the bug report should be analyzed when recognizing BB. **RQ 4**: *What are the common manifestations of the bugs on the mobile device?* This RQ explores how the reports describe the display of the bug on the mobile app or the mobile device (i.e., the manifestation). It motivates the design of the mechanisms for recognizing bugs based on the BB description.

3.1 Methodology

3.1.1 Bug Reports Preparation. Our empirical study was conducted on 380 real-world mobile app bug reports. To ensure the representativeness of this dataset, we followed the data collection practice used by existing research works on mobile app bug report studies [34, 46, 69] and standard statistical sampling methods [18]. Specifically, we chose GitHub [23] as the source of mobile bug reports due to its popularity in hosting mobile apps and tracking issues. Since our approach focuses on analyzing bug reports for mobile apps, we selected repositories hosting apps available on FDroid [22], a popular open-source mobile app store, resulting in 3,342 repositories. We then mined issues from these GitHub repositories between November 8, 2018, and November 8, 2023. To ensure our dataset contained only bug reports, we only retained issues labeled as "bug", excluding feature requests and other non-bug-related issues. This resulted in an initial set of 43,390 reports from 755 repositories. The minimum, median, mean, and maximum number of issues from a single repository was 1, 6, 57, and 3,646. Next, to specifically focus on mobile apps, we filtered out reports containing the keywords "windows," "linux," or "tv" (in a case-insensitive manner) since some repositories host apps for multiple platforms. This filtering reduced our dataset to 39,819 issues. To make the dataset manageable for human review while ensuring statistical validity, we sampled 380 bug reports from the dataset, achieving a 95% confidence level with a 5% margin of error [18]. During the sampling, to prevent overrepresentation of any single app, we limited each repository to at most ten sampled reports following an existing practice [46, 69]. This resulted in a final dataset of 380 bug reports from 157 different mobile apps.

3.1.2 Coding Process. To answer the RQs, two authors conducted a qualitative coding process on the bug reports in the dataset. The authors followed the practice of both deductive and inductive coding [37] to code the BB descriptions systematically. Deductive coding describes an analysis with predefined codes based on past studies or experiences, while inductive coding derives the codes from the data. In our study, combining these two approaches helped accelerate the coding process by reusing codes from prior studies and discovering new findings by defining new codes. For RQ1, we collected common types of mobile app bugs studied by researchers. Specifically, they include: (1) UI Display Bugs: Issues that affect UI display [56]; (2) Function bugs: Problems that prevent certain functions of the app from working properly [71]; (3) Performance bugs: Bugs that slow down the app's responsiveness or lead to increased energy or memory consumption [53]; (4) Security bugs: Vulnerabilities that threaten the security of user information [52]; and (5) Crash bugs: Bugs that cause the app to crash unexpectedly [45]. For RQ2, we adopted the information modalities concluded by a study on mobile bug report reproductions [46] as the initial codes: Natural Language, Image, and Video. For RQ 3 and 4, we did not have initial codes since, to the best of our knowledge, no existing work answered these questions. Therefore, we identified and defined their codes during the coding process as introduced below.

The coding process for each RQ was conducted in an iterative manner. Specifically, at each round, each coder independently analyzed 50 bug reports and labeled each BB description in the report using either one of the existing codes or a new code if none of the existing codes fit. After each round of independent coding, the two authors met and discussed their coding results. This meeting aimed to resolve the conflicts in their coding and discuss the new codes. When the two authors could not agree, a third author was involved to resolve the disagreement. After the discussion, the codebook was updated with the new codes and used in the following coding round. The iterative coding and discussion process continued for four rounds. After the fourth round, no new codes were generated during the independent coding process. Then, the two coders used the codebook to code the rest of the bug reports. In the end, two coders merged their codings and resolved any conflicts by discussion.

We measured the Inter-Coder Agreement (ICA) on the independent codings to evaluate the reliability of the coding results. The Krippendorff's alpha (α) [48] coefficient was used for this evaluation. Our analysis revealed high ICA on the overall codings (α = 81%, i.e., Substantial Agreement [64]). Specifically, the coders agreed on codings for RQ1 to RQ4 with corresponding α to be 91%, 94%, 92%, and 78%.

3.2 Empirical Study Findings

3.2.1 RQ1: Types of Mobile App Bugs. Among the 380 reports, we identified 71 Crash bug reports (18.7%); 159 Function bug reports (41.8%); 11 Performance bug reports (3%); 65 UI Display bug reports (17%); and 1 Security bug report (0.3%). The remaining 73 (19%) reports were found not to be bug reports for mobile apps. This finding showed that the bug reports selected in our dataset include various types of bugs for mobile apps. Please note that reports identified not as a bug report for a mobile app were excluded from the coding process in RQ2-4. However, they were included in the total count of bug reports for the statistics presented in these RQs.

3.2.2 RQ2: Modality of BB Description. Our study identified three information modalities for describing BBs: natural language, videos, and screenshots, with natural language being the most commonly used. Specifically, our study found that 298 reports (78%) used natural language when describing the BB, 72 reports (20%) used images, and 18 reports (5%) used videos. Note that the sum of these percentages is larger than 100% since a bug report may have several BB descriptions. Among the reports with natural language BB descriptions, 70% of bug reports described BB only in natural language, and 30% of reports provided videos or screenshots as additional information. Only one report provided only screenshots or videos when describing the BB.

3.2.3 RQ3: Location of BB Description. Our study showed that BB was described in various locations in a bug report. Specifically, we found that 290 bug reports (76%) provided the BB description in the report body, 119 reports (31%) described BB in the title, and 21 reports (5%) described the BB in the comments of the report. Again, the sum of these percentages is larger than 100% since a bug report may have several BB descriptions. For the 290 reports documenting the BB in the report body, we found that 183 (63%) report bodies followed a structured template, while 107 (37%) were free-form text. Generally, the report template specified a section for documenting the BB. However, we observed that in many cases, reporters did not strictly follow the template and mixed the BB description with other information. For example, we found that 20% of the reports described the BB in the "Reproduction Step" section instead of the "Buggy Behavior" section.

3.2.4 RQ4: Manifestation of BB. Our study found that although the bug reports described different types of bugs (as categorized in RQ1), their manifestation on the mobile device shared common characteristics. In total, we found four common manifestations (Ms), and each of them was further divided into sub-categories.

M1. Cause App Crash (16%≈62/380) Bug reports in this category described the BB as causing a crash on the mobile app. Generally, such a bug triggered the default crash handler in the OS system, which displayed a dialog window on the UI saying "xx has stopped" [21].

M2. Affect UI Widget Status (40%≈153/380) Bug reports in this category described the BB as affecting the display of UI widgets. We further divided this category into Display Abnormal Widget and Conceal Widget.

(1) Display Abnormal Widget (114/153): This manifestation describes a case where the bug caused a UI widget with certain abnormal characteristics to display on the UI. We found reporters described the displayed widget in the bug reports from three aspects:

- Text Label (82/114): The most common way reporters described the widget was by using the text label that appeared on the UI widget. Under this category, we observed that 63% (52/82) of the reports directly mentioned the exact text shown on the UI widget. For example, in report nextcloud-android-7602 [14], the reporter described an error message widget by saying, "I always retrieve the same error message *The built-in media player can not play the file*." For the other 37% (30/82) reports, we observed that reporters provided a natural language description of the label, which did not exactly match the label text. An example of such cases is report AnySoftKeyboard-2825 [13]. The reporter described the abnormal display of characters on a widget by saying, "All letters are lowercase."
- Icon (28/114): Reporters also described the affected widget by its icon. Among bug reports containing such descriptions, we found that most icon descriptions (11/28) were related to the widget type. For example, in report msoultanidis-quillnote-46, the reporter mentioned "the preview was empty, except for a single checkbox." The reporter directly used the widget type, "checkbox" [20], to describe it. Besides the widget type, we found six reports describing the icon based on its function, such as the "uninstall button"; eight reports describing the color of the icon; and three reports describing the size of the icon.
- Status (8/114): In eight bug reports, reporters described the status of the displayed widget in the BB description. Specifically, the status refers to whether a button is clickable or whether a checkbox or radio button is checked. For example, in report unstoppable-wallet-android-3763 [29], the developer described a bug that caused a widget to become unclickable by saying, "The enabled passphrase is changed to disabled.".

(2) Conceal Widget (32/153): In addition to displaying a UI widget, reporters also described the effect on a widget as the UI widget disappearing. For example, in report nextcloud-android-8905 [15], reporters described the buggy behavior that caused a disappeared widget by saying, "No media controls to be found".

M3. Cause Abnormal Relation between Two UI States (13%≈50/380) Bug reports in this category described the BB as causing unusual relationships between two UI states. Among such bug reports, we found two common relationships: *Identical Screens* (20/50) and *In-screen Widget Changes* (18/50). The first relation, *Identical Screens*, describes two given UI states that are the same. For instance, in report streetcomplete-2410, the reporter mentioned "Nothing happens". This implies that the UI remained the same before and after the last reproduction step. The second relationship, *In-screen Widget Changes*, describes the changes or persistence of widgets on two given UI states. In 12 reports, reporters mentioned certain widgets stay on the UI after a UI action. An example of such a case is deltachat-android-725[7], where the reporter mentioned, "The notification stays". In six reports, reporters noted that the widget disappeared from the UI after a UI action. For example, in report jellyfin-android-459 [12], the reporter mentioned that "The controls disappear". Note that, this manifestation describes a change involving two UI screens, and is different from the "Conceal Widget" in M2, which describes the absence of a widget in a single UI screen. We also observed other less common UI states relationships. For example, in two bug reports, the reporter described a brightness change between two UI states.

M4. Affect Mobile Device Status (17%≈65/380) The reports in this category described the BBs as effects on the mobile device status. In most of the reports in this category (90%), the effect caused certain messages to be generated in the device log. Such reports were mostly crash bug reports, as classified in RQ1. However, we also observed some non-crash bugs with such BB descriptions. For example, in report openhab-android-2583 [26], the reporter provided the device log information for a UI display bug. In addition to the device log, we observed bug reports describing effects on the mobile device's audio (6%), volume (2%), and keyboard status (2%). For example, in report

flex3r-DankChat-66 [25], the bug caused the keyboard to keep displaying even after another button was clicked.

Unclear Manifestation Logic (10%≈38/380) We were not able to classify the BB manifestation of bug reports in this category due to insufficient BB descriptions. Specifically, bug reports in this category usually described the bugs by directly saying "xxx doesn't work" or "unable to do xxx" (e.g., report Shabinder-SpotiFlyer-764 [27]). Without knowing further details of the app logic and the bug information, we could not analyze and classify the BB for these reports.

Manifestation Co-occurrence Analysis Based on the manifestations concluded above, we analyzed the co-occurrence between different manifestations within the same bug report. We found that 16% (59 out of 380) of bug reports provided more than one manifestation of the bug. Among them, 58 bug reports described two manifestations, and one bug report described three manifestations. The three most common combinations of the manifestations that we found in the dataset were (1) M1 and M4, (2) M2 and M4, and (3) M2 and M3.

3.3 Discussion of Empirical Findings

In this section, we present the findings (Fs) on BB information in mobile bug reports derived from the empirical study results. We also discuss the requirements for automated recognition techniques based on these findings, which motivate the design of our approach in Section 4.

Based on the results of RQ2 (Section 3.2.2) and RQ3 (Section 3.2.3), the first finding (**F1**) is as follows: *the BB information is often conveyed in unstructured natural language, with no consistent location within bug reports.* Based on the results of RQ4, we identified the second finding (**F2**): *The BB manifests in various aspects of the mobile device, from the UI to device internal status.* Specifically, we categorized four common manifestations of mobile app BBs in RQ4. Among these, we observed that *the objects affected by the bugs can be classified as UI widgets, UI screens, and Mobile Device State* (**F2.1**). By studying how reports described these affected objects (e.g., the impacted UI widget) in RQ4, we found that *the descriptions often do not precisely match the UI information* (**F2.2**). Furthermore, we noted that *the manifestations involve different relationships among the objects*, such as a UI widget appearing or disappearing from UI screens, and *one bug report may contain multiple manifestations* (**F2.3**).

Based on these findings, we identified two key requirements (Rs) for an automated BB recognition technique. First, from both F1 and F2, the technique needs a *strong natural language understanding and processing capability* (**R1**). This ability is essential to thoroughly analyze bug report text, accurately identify BB information, and recognize the nuances between the natural language description and the actual UI details. Second, from F2, the technique must be able to *identify various types of BB manifestations within the bug report text* and *flexibly manage the relationships between these different manifestations* (**R2**).

4 Automatically Recognize Mobile App Buggy Behaviors

The goal of our approach is to automatically extract BB information from a bug report and use that information to recognize whether the reported bug is successfully reproduced by a reproduction tool. Figure 2 shows the workflow of a typical reproduction tool (in the dashed box) and how our approach integrates into this workflow. As shown in the figure, the input to the bug report reproduction process is a bug report that needs to be reproduced. A typical reproduction tool analyzes the bug report and then begins triggering UI events on the app with the goal of finding a sequence of events that enables the bug to be reproduced. To automatically recognize the BB during this process, the first step of our approach, Buggy Behavior Extraction, parses the bug report to identify the text that describes the BB and then analyzes this text to convert the BB into a standardized representation, BugIR. The second step of our approach, Buggy Behavior Recognition,

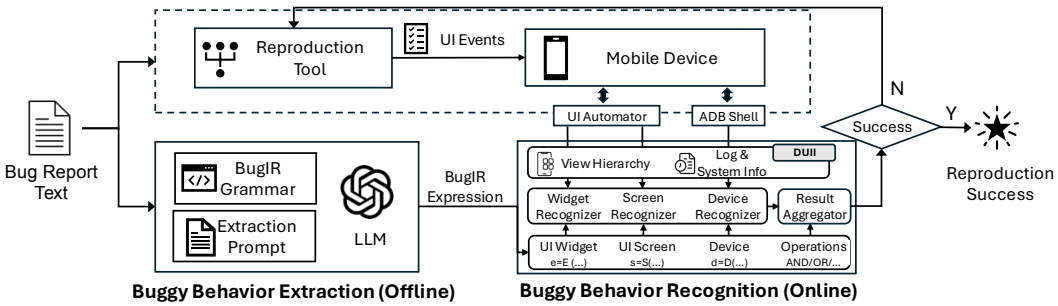


Fig. 2. Workflow of our approach with a typical bug report reproduction process shown in the dotted box.

interacts with the mobile device to determine if the bug was triggered after the reproduction tool executed a UI event. As input, this step takes the BugIR expression generated by the first step and the corresponding real-time Device and UI information (DUII) retrieved from the mobile device. To determine if the bug is reproduced, our approach then evaluates whether the BB expressed by the BugIR expression matches the DUII. If a match is found, the reproduction process is deemed as successful, and the reproduction ends, as indicated by the "Y" branch of the "Success" diamond in the diagram. Otherwise, the reproduction continues, as indicated by the "N" branch.

In the rest of this section, we first present the structure and semantics of BugIR in Section 4.1. Then, we describe our approach for identifying the BB in the bug report and converting it to a BugIR expression in Section 4.2. Finally, in Section 4.3 we detail the design and processes for the mechanisms that evaluate whether the BugIR expression matches the DUII.

4.1 BugIR Definition

BugIR (**Bug**gy Behavior **I**ntermediate **R**epresentation) is our standardized representation for expressing different manifestations of the BB described in a bug report. The design of BugIR is driven by the findings in Section 3.3, specifically findings **F2.1** and **F2.3**. At a high level, the syntax of BugIR consists of four parts: $BugIR = (T, A, O, X)$. The first part, *Types (T)*, defines a set of objects that are impacted by the buggy behavior, such as UI widgets and UI screens, as indicated by **F2.1**. The second part, *Attributes (A)*, is a set of attributes, associated with each Type. The third part, *Operations (O)*, expresses the relations among the objects, as indicated by **F2.3**. The last part, *Expressions (X)* is the set of BugIR expressions that can be constructed using the previous three parts. BugIR is designed to represent all types of manifestations of mobile app bugs we identified in Section 3.2.4. In the following, we introduce each part of BugIR in detail.

4.1.1 BugIR Types (T) and Attributes (A). We define BugIR *Types* with their *Attributes* to represent the various kinds of objects described in bug reports, which can be formally written as $T(a)$ where T is a Type and a is the attributes' value. Specifically, BugIR includes three types, UI Widget, UI Screen, and Device, which are summarized, along with their attributes, in Table 1. Together, the three types represent all objects described in bug reports we found in our study (Section 3.2.4). The Attributes of each Type are defined according to the ways in which bug reports describe these objects as identified in our study. The first type, ***UI Widget***, represents a View (e.g., a button) in the device's UI. It was described in bugs with manifestation M2 (Section 3.2.4). This type has two attributes, *"desc"* and *"status"*. The *"desc"* attribute captures any natural language description in the bug report of the View's label or icon. The *"status"* attribute is a set of values that represent the status of the View in the UI. For example, if the bug report states that the View cannot be

Table 1. Types in BugIR

Type	Attribute
UI Widget (E)	**desc**: The textual description of the affected UI widget's label or icon. **status**: The status of the affected UI widget, can be one or more of the following: *clickable*, *unclickable*, *checked*, *unchecked*, *enabled* and *disabled*.
UI Screen (S)	**crash**: A boolean value indicating if the screen displays a crash dialog. **keyboard**: A boolean value describing if the keyboard is displayed.
Device (D)	**volume**: An integer value indicating the mobile device's volume. **audio**: A boolean value representing if the mobile device's audio is on or off. **log**: The text generated in the mobile device's log.

clicked, the value of *"status"* will include "unclickable". The full list of possible status values is shown in Table 1. The second type, ***UI Screen***, represents a UI of the mobile app, and has two attributes: *"crash"* and *"keyboard"*. The *"crash"* attribute captures whether the screen was described as exhibiting a crash in the bug report, and the *"keyboard"* attribute captures whether the keyboard is reported to be displayed. This type and its attributes were described in bugs with manifestations M1, M3, and M4 (Section 3.2.4). The third type, ***Device***, represents the status of the mobile device. The Device type and its attributes are mentioned in bugs with manifestation M4 (Section 3.2.4). The Device type has three attributes. The first attribute, *"volume"*, is an integer value that captures the volume of the device when it is mentioned in the bug report. The second attribute, *"audio"*, is a boolean value and is set to true when the bug report mentions that audio is playing. The final and third attribute, *"log"*, captures any exception message printed to the device console mentioned in the bug report.

In manifestations M2, M3, and M4 (Section 3.2.4), we found that BBs could be characterized as a change between two object types. For example, the bug could cause the device audio to be turned off after an action. Therefore, in BugIR we introduce the notion of an initial state and an after state. We represent the after state by adding the $'$ symbol to the symbol representing the initial declaration of the state. For example, if $d = D(audio = true)$ denotes a Device state with audio on, then $d' = D(audio = false)$ in the same BugIR expression denotes the after state of d and indicates that the audio turned off.

4.1.2 BugIR Operations. The *Operations* are defined to describe relations among Types, which can be formally written as $O(t_1, t_2)$ where O is an operation and t_1 and t_2 are the involved Types. BugIR defines two kinds of operations, which together allow it to describe all relationships among Types that we saw in our study (Section 3.2.4). The first operation is **comparison operators**, which express comparisons between two instances of a Type or its attribute. For example, to describe that there is a volume difference between two device states. BugIR supports four comparison operations between attributes: *equal* (==), *not equal* ($\neq$), *less than* (<), *larger than* (>). The *equal* or *not equal* operators can also be used between two instances of a Type. For example, if a bug report states that there is no change in a screen's appearance after a UI event executed by a reproduction tool (as identified in manifestation M3 in Section 3.2.4), this can be written as $s == s'$. The second kind of operation is the **in_screen(e, s)** function, which captures the information from a bug report that the Widget e is displayed in the Screen s. For example, it can be used to represent text in a bug report that indicates that a menu button is present in the UI.

4.1.3 BugIR Expressions (X). The *Expressions* define how to express a BB using the elements defined previously in BugIR. Specifically, an expression could be either a declaration of a Type, an Operation, or any logic formula of Expressions, which is formally represented as $X ::= T(a) \mid O(t_1, t_2) \mid X_1 \wedge X_2 \mid X_1 \vee X_2 \mid \neg X_1$. In this way, BugIR expression supports complex BBs that manifest on multiple objects or bug reports involving multiple manifestations. In the following, we provide two examples to illustrate the usage of BugIR and showcase its flexibility in expressing BBs in bug reports. Each example provides the textual description of the BB and the corresponding BugIR expression.

```
1 Buggy Behavior Description:
2 I cannot click the miniplayer. Also, after I rotated the phone, it disappeared.
3 BugIR Expression:
4     e=E(desc="miniplayer", status="unclickable") ∧
5     s=S() ∧ in_screen(e,s) ∧
6     s'=S() ∧ ¬ in_screen(e,s')
```

Listing 1. BugIR Expression Example 1

The first example in Listing 1 describes a BB that occurs after the phone is rotated. This bug causes a UI View associated with the miniplayer to disappear. The Widget object e at line 4 with a “desc” attribute set to “miniplayer” and the “status” attribute set to "unclickable", represents the miniplayer View described by the user. The first Screen object s at line 5 represents that the miniplayer widget was first present in the screen. The second screen object s' at line 6 represents that the miniplayer widget was no longer present in the screen. Note that, the rotation itself is not included in the BugIR expression since it describes a reproduction step (UI Event) and not the buggy behavior.

```
1 Buggy Behavior Description:
2 The volume increased after I pressed the up button. However, the music stopped
3 playing, and an AudioError was shown on the device console.
4 BugIR Expression:
5     d=D(audio=True) ∧
6     d'=D(audio=False, log="AudioError") ∧
7     d'.volume>d.volume
```

Listing 2. BugIR Expression Example 2

The second example in Listing 2 describes a BB that occurs after the user presses the up button. This bug causes the music to stop and an error message "AudioError" to be shown in the console. The Device object d at line 5 with an "audio" attribute set to true describes the initial device state where the audio is on. The second Device state object d' at line 6 with an "audio" attribute set to false and the “log” attribute set to "AudioError", represents the changed state where the audio is off and an error message has been printed to the console. The comparison operation at line 7, shows that the volume of the second Device state d' is greater than the volume of the first Device state d, representing the volume increase noted in the bug report text. Note that the action of pressing the up button is not included in the BugIR expression since it describes a reproduction step (UI Event) and not the buggy behavior.

4.2 Transforming BB Descriptions into BugIR Expressions

Given the bug report, the first step of our approach is to transform the BB described in the report into a BugIR expression. As noted in finding $F1$ in Section 3.3, the BB description is often mixed in with other information and written in diverse ways using natural language, which complicates this transformation. Our insight to address this challenge is that we can leverage an Large Language Model (LLM), which has strong natural language processing and generation abilities, to help us

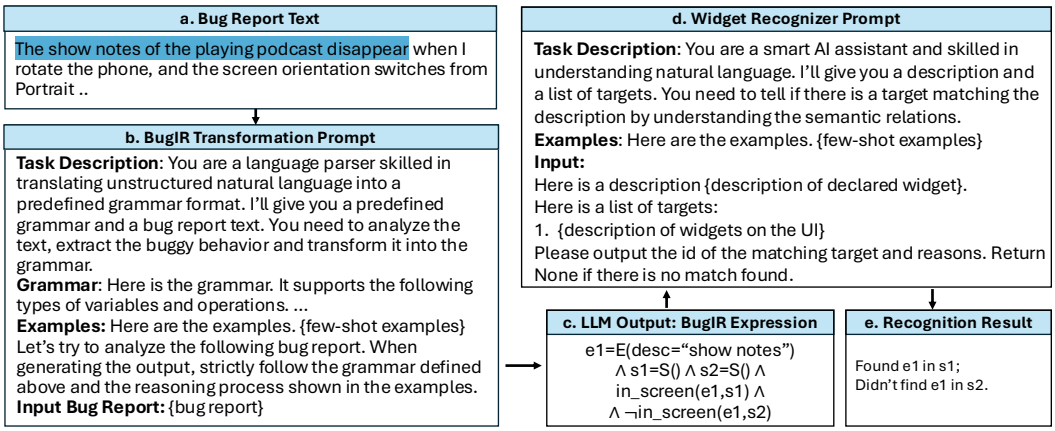


Fig. 3. An example of our approach's workflow

Table 2. An example of our approach's CoT prompt for generating the BugIR expression

Step 1: The BB description is "the show notes of the playing podcast disappear". **Step 2**: The BB manifestation is to cause a widget to disappear. **Step 3**: In the BugIR expression, I first declare a UI Widget variable and two UI Screen variables. I set the "desc" attribute of the widget as "show notes" based on the description in bug report. To describe "disappear", I use the function in_screen to express the widgets on a screen. Then, I use the NOT and in_screen operations to express that the widget does not exist on the other screen. The final expression is ...

analyze the bug reports and perform the transformation. To do this, we designed prompts that can teach the LLM to identify the text in a bug report that describes the BB and transform this text into a BugIR expression. These prompts are then given to the LLM along with each bug report to get the BugIR expression corresponding to the bug report. Figure 3 illustrates the processing steps on the bug report in Figure 1.

At a high level, this step of the approach takes the bug report as input. It then generates a teaching prompt for the LLM. Box b in Figure 3 shows an example of the prompt. The format of the teaching prompt is: ⟨Task Description⟩+⟨BugIR Introduction⟩+⟨Examples⟩+⟨Input Bug Report⟩. ⟨Task Description⟩ explains the goal of the task, and the format of input and output to the LLM. ⟨BugIR Introduction⟩ describes the syntax of BugIR (as defined in Section 4.1) using natural language. ⟨Examples⟩ provides teaching examples, as described in the next paragraph, and ⟨Input Bug Report⟩ is the whole text of the bug report. The output is the BugIR expression that represents the BB described in the bug report. Box c shows the corresponding output.

Our process for generating the Examples enables our approach to address challenges that arise since the off-the-shelf LLMs have not been trained to identify BBs and generate BugIR expressions. The first challenge is that the generation of BugIR expression involves multiple steps. Our approach first needs to identify where the BB is described, extract its manifestation, and then generates the BugIR expression. However, LLMs exhibit less effective performance on tasks with such multi-step reasoning. To address this challenge, we employed Chain-of-Thought (CoT) [68] prompting to guide the LLM to perform a step-by-step reasoning. The next challenge arises from the variety of BB that could be described in bug reports as identified in Section 3.2.4. To address this issue, we

employed few-shot prompting [60], which allowed us to provide the LLM with several representative transformation examples, from which it could learn to handle different variations of the BBs in the bug reports. Based on these two insights, we selected 13 bug reports from our empirical study dataset and designed a CoT prompt for each of them to demonstrate how to generate the final BugIR expression step by step. The examples cover different manifestations identified in section 3.2.4 and the usages of BugIR. Table 2 shows the CoT prompting example of the bug report in Box a of Figure 3. It includes the three steps in our task: (1) locating the BB description in the bug reports, (2) extracting its manifestation, and (3) converting the parts of the recognized text to a BugIR expression. Note that, to avoid over-fitting, we excluded all bug reports selected in this section from our evaluation dataset in Section 5.

4.3 Recognizing BB Based on BugIR Expressions

The second step of our approach identifies whether the BB described in the bug report has been triggered by the reproduction tool. As shown in Figure 2, the inputs for this step include the BugIR expression generated by the analysis discussed in Section 4.2 and the DUII retrieved from the device, which specifically includes the view hierarchy (VH) of the UI collected using UI Automator [28], the device log, and system information (i.e., audio status and volume) collected via ADB Shell [19].

Our approach detects the BB by checking if the BugIR expression matches the DUII. Due to the different semantic meanings of each part in the BugIR, matching them requires different mechanisms. Therefore, based on the current definition of BugIR, we designed three recognizers for matching the three BugIR Types (defined in Table 1) with the DUII, and their related Operations. At a high level, given the inputs, this step of our approach begins by extracting the Types declared in the BugIR expression along with the related operations. Then, it matches them with the corresponding DUII using the recognizers. The individual match results (true for a successful match and false for an unsuccessful match) are then combined with the logic operators in the BugIR expression together to evaluate whether the overall expression is true or false. This ultimately determines whether the BB is triggered or not.

4.3.1 Recognizing UI Widgets. This mechanism identifies, in an app's UI, the view described by a BugIR Widget type. The key challenge in this identification is that even though the BugIR type expression contains a description of the widget from the bug report, the description generally does not exactly match the corresponding UI view's information (finding *F*2.2 of Section 3.3). Instead it may be similar in topic or semantics. This led us to explore mechanisms based on an LLM, as they are very good at considering semantic relationships, which helps our approach to identify the connection between the language used in the BugIR description and any textual information associated with a view in the UI. Our approach for recognizing UI widgets has two steps. Given the target widget declared in a BugIR expression and views shown in the UI screen, the first step is to generate natural language descriptions for the widget and views. Then, in the second step, these descriptions are composed into a prompt and provided to the LLM, which identifies the closest match based on the semantic relations among the descriptions.

Generating Descriptions for Widgets By itself, the textual information contained in the widget's and views' attributes lacks enough context to enable the LLM to adequately7 evaluate similarity. Therefore, the first step in the Widget recognition is to generate a natural language description for each widget that provides contextual information about the text so that the LLM can more accurately identify the closest match. Our approach uses predefined templates to generate such natural language descriptions for both the widget described in the BugIR expression and the view objects in the app's UI. For the widget variable declared in a BugIR expression, we defined the template as "*A widget with description: {desc} and status: {status}*" where *desc* and *status* are the

values of the attributes for the *UI widget* variable. For a view in the app's UI, the template was defined as "*The widget is a {class} with label {label}. It has a description as {content-desc}. The name of its icon is {resource-id}. It's {clickable}, {enabled} and {checked}.*" The placeholders represent the values of the view's attributes. Specifically, *class* is the class name of the widget, such as `Textview` and `Button`; *label* is the text displayed on the widget; *content-desc* is the description of the widget that defined by mobile app developers; *resource-id* is the filename of the widget icon image, which is generally used as a meaningful representation of the icon; *clickable*, *enabled* and *checked* correspond to the fields that show the status of the widget. We selected these fields based on the aspects that mobile app users described widgets in their bug reports (Section 3.2.4). When the values of a view's attributes are missing, we substitute the placeholders in the template with empty quotation marks to indicate to the LLM that the value is absent.

Prompt Generation The goal of the second step of the recognition is to generate a prompt to guide the LLM to identify the most closely matched widget, if it exists. To help the LLM handle different inputs, our approach generates the prompt following the same Few-shot prompting practice introduced in Section 4.2. We chose not to use CoT prompting for this task, as the widget matching primarily requires natural language understanding rather than complex reasoning. For this task, we selected four examples from our empirical study dataset to cover different aspects from which reporters may describe a UI widget. Again, these reports were also excluded from our evaluation dataset in Section 5. As shown in Box d of Figure 3, the prompt includes three parts: ⟨Task Description⟩+⟨Examples⟩+⟨Input⟩. The ⟨Task Description⟩ introduces the role of LLM for this task and instructs it to return a boolean value indicating if there is a match and corresponding reasons. The ⟨Examples⟩ shows the few-shot examples to LLM. The ⟨Input⟩ includes the descriptions of both the declared widget in BugIR expression, the widgets in the UI, and the final query.

4.3.2 Recognizing UI Screens. This mechanism aims to verify if a given UI matches the description of the UI Screen Type in the BugIR expression. This matching involves checking situations indicated by the two attributes of a UI Screen Type (as defined in Table 1) : (1) "crash", indicating a crash happened on the screen when its value is true; and (2) "keyboard", indicating that the keyboard is displayed when its value is true.

The situations described by both attributes of the UI Screen Type are related to some specific views being displayed in the UI. Therefore, our approach analyzes the VH file of the UI to perform this matching. To detect a crash on a screen, our approach searches for a crash-related view in the VH. However, this could be challenging since the view contained in crash dialogs can vary due to different mobile system versions and developer customization. Despite these variations, we found that the labels on crash widgets consistently convey a semantically similar message: the app has terminated due to a crash. Therefore, our approach identifies a crash by searching for a view in the device's UI that includes a description semantically similar to "app has stopped" utilizing the LLM prompting method introduced in Section 4.3.1. Leveraging the natural language understanding capabilities of LLM, our approach can recognize the crash dialog regardless of its visual appearance on the screen. To check if the keyboard is displayed, our approach looks for views associated with the virtual keyboard in the app's UI. These views are organized under specific packages defined in the Android system framework, such as "com.google.android.inputmethod.xx." Our approach detects the existence of views belonging to these packages in the VH to determine if the keyboard is displayed.

4.3.3 Recognizing Device Status. The goal of this mechanism is to retrieve device status information, namely, the three attributes in the Device type (as shown in Table 1): "volume", "audio" and "log", and then determine if it matches the values or satisfies comparisons described in the BugIR expression. For the "log" attribute, our approach validates the specified value by searching for an exact string

match in the device log. For the "volume" and "audio" attributes, our approach directly retrieves their corresponding values from the device using the ADB commands. With this information, our approach further checks the corresponding values (if specified) or evaluates the comparison operations to see if they match the BugIR expression.

5 Evaluation

We evaluated our approach by answering the following Evaluation Research Questions (EQs):
EQ 1 How accurately can our approach generate the BugIR expression from bug report text?
EQ 2 How effectively can our approach recognize the described buggy behavior at runtime?
EQ 3 How impactful is our approach when integrated with existing bug report reproduction techniques?

5.1 Approach Implementation

To facilitate the evaluation, we implemented our approach into a research prototype, BugSpot. The core algorithms of BugSpot were implemented in Python. BugSpot uses GPT-4 [2] for processing the bug report text and recognizing UI widgets. We set GPT-4's parameters "seed" to 10 and "temperature" to 0. These values were selected to ensure that the responses from the language model were as deterministic as possible. Our implementation is available in a replication package [1]. We obtained the implementations of ReCDroid [78], ReproBot [76] and Roam [75], three state of the art reproduction techniques for mobile bug reports, from their public websites [9–11]. Our experiments were performed on Android emulators running on a physical x86 Ubuntu 20.04 machine with eight 3.6GHz CPUs and 32G of memory.

5.2 Evaluation Dataset

We used a dataset of 87 real-world Android bug reports for our experiments. The subjects came from the artifacts of three data sources: (1) the evaluation dataset of automated reproduction techniques ReCDroid [77, 78], Yakusu [39], ReproBot [76] and Roam [75]; (2) an empirical study on Android bug report reproduction [46]; and (3) an open-source Android bug report dataset [69]. Altogether, the datasets provided an initial set of 399 bug reports. For each of the bug reports, we attempted to manually reproduce them to make sure the buggy behavior was reproducible and to identify duplicates. During this process, we found 134 bug reports that were not usable for our evaluation due to the following reasons: (1) The bug report or the corresponding APK file of the app was no longer available; (2) The steps were unreproducible due to a lack of necessary setup information, such as an account or a specific environmental configuration; and (3) The described buggy behavior was not displayed after reproducing the steps. After removing the duplicates from the bug reports in the dataset, we obtained the final 87 subjects for the evaluation. Among them, 43 are crash bug reports and 44 are non-crash bug reports. Note that none of the bug reports in this dataset were used in the empirical study in Section 3 or in the prompts used by our approach.

5.3 EQ 1: Performance on Generating BugIR Expression

5.3.1 Protocol. The goal of this EQ is to evaluate how accurately the mechanism defined in Section 4.2 can transform the buggy behavior described in bug report text to a BugIR expression. In the experiment, we ran BugSpot on each of the raw bug report texts (collected in Section 5.2), generated the BugIR expression, and then evaluated the accuracy aginst ground truth. We manually constructed the ground truth of the BugIR expression for each bug report in the dataset. To avoid biases in the manual construction, the first two authors first wrote down the BugIR expressions for each bug report separately, then compared and discussed the results to get the final ground truth. To measure accuracy, we computed the precision and recall of the generated BugIR expression

aginst the ground truth. For a given BugIR expression, our approach's precision was calculated as the percentage of declared *Type*, *Operations*, and logic operators that matched the ground truth, and recall was calculated as the percentage of *Type* variables, *Operations*, and logic operators from the ground truth that were included in the expression. As a BugIR expression could be partially correct, employing both precision and recall provided a more nuanced understanding of the overall correctness and incorrectness. We repeated the experiment ten times to mitigate the impact of the non-determinism of the LLM. In addition, we measured the *Coefficient of Variation (CV)* of the results from ten rounds of experiment to assess their consistency. According to existing literature [33, 50], a CV exceeding 30% suggests that experiments are excessively interfered with randomness.

5.3.2 Discussion of Results. The average precision and recall of the BugIR expressions generated by BugSpot was 91% and 90%. Additionally, BugSpot achieved perfect precision on an average of 73 (84%) reports and perfect recall on 71 (81%) reports for the ten runs. As shown by the results, BugSpot could accurately generate the BugIR expression for most bug reports. This showcased its effectiveness in identifying the natural language description of BB and translating it into a BugIR expression. The CV was 1.7% for precision and 1.9% for recall, both of which were far below the threshold 30, indicating that the results are consistent across different runs of experiments.

We further analyzed the cases where BugSpot inaccurately generated the BugIR expression and identified two predominant reasons. One reason was that BugSpot misinterpreted the buggy behaviors and consequently generated inaccurate BugIR expressions. This happened when understanding the BB description involved a complex reasoning process. An example is Omni-Notes#634 [6]. In the bug report, the reproduction steps are to enter two tags, "testA" and "test", then delete the tag "test". The report describes the BB as "when removing one tag, its text was removed from all tags." This implies that on the buggy UI screen, the text shown in the remaining tag is changed to "A". However, BugSpot was not able to understand the logic and generated an incorrect BugIR expression based on another part of the bug report text. The other reason was that BugSpot misused the BugIR's grammar when generating the BugIR expression. For example, in the report, ATimeTracker#121 [3], the BB is displaying a widget called "activities menu". The correct way of describing this widget in BugIR is *e=E(desc="activities menu")*. However, in the expression generated by BugSpot, BugSpot mistakenly added another attribute "status" in the widget variable declaration. In addition to these two major reasons, we also observed one special failure case in report MaterialFiles#184 [4]. The failure occurred because the described BB cannot be expressed by BugIR's grammar. Specifically, the reporter only provided a very high-level BB description "Server restarts by its own." Due to the lack of detailed information of what happened on the device, this BB could not be captured by any of the *Types* or *Operations* defined in Section 4.1. In this case, instead of a BugIR expression, BugSpot generated a natural language sentence explaining this unsupported situation.

5.4 EQ 2: Accuracy in Recognizing Buggy Behavior at Runtime

5.4.1 Protocol. The goal of this EQ is to evaluate the accuracy of the mechanism introduced in Section 4.3 for recognizing the buggy behavior at runtime. During the experiment, for each bug report, we executed the correct reproduction steps on the app, and after each step, we ran BugSpot to recognize the BB. In this experiment, we compared our approach, BugSpot, with two additional methods: BugSpot+GT, LogOnly. The BugSpot+GT method is used to evaluate the impact of the quality of BugIR expression on the recognition performance. It utilizes the ground truth (i.e., perfect) BugIR expression to recognize a bug. The LogOnly method is the mechanism used by existing reproduction approaches [38, 76, 78] to recognize BBs. It recognizes bugs through exception logs extracted from bug reports. Similar to EQ 1, the experiments were repeated ten times to mitigate the impact of the non-determinism of the LLM.

Table 3. Results of EQ2 - Performance on Recognizing Buggy Behavior (%)

Metrics		BugSpot	BugSpot+GT	LogOnly
Avg. Rec-Rate	Overall	83	88	18
	Crash	98	98	35
	Non-crash	70	79	2
Avg. FP-Rate	Overall	17	17	0
	Crash	0	0	0
	Non-crash	33	33	0

We measured two metrics in this experiment: *recognition rate (Rec-Rate)* and *false positive rate (FP-Rate)*. The *recognition rate* is the percentage of bug reports for which the tool correctly recognized the BB on the final buggy state during reproduction. This metric reflects how effectively an approach can identify a bug when it occurs, with higher values being more desirable. The *False Positive Rate* is the percentage of bug reports for which BugSpot incorrectly reported the BB on a non-buggy state during the reproduction. This metric reflects how often an approach may falsely report BBs, with lower values being more desirable. The Coefficient of Variation (CV) was also used to evaluate the consistency of the results.

5.4.2 Presentation of Results. Table 3 displays the recognition results of the three approaches. Row "Avg. Rec-Rate" and "Avg. FP-Rate" show the average value of the two metrics used in the experiment across the ten runs. For each metric, the table lists the results on the whole dataset (row "Overall"), and the breakdown of the results on only crash bug reports (row "Crash") versus only non-crash bug reports (row "Non-crash").

5.4.3 Discussion of Results. As shown in Table 3, BugSpot correctly identified BBs in 83% of the bug reports across the entire dataset, with accuracy rising to 88% when using perfect BugIR expressions. In contrast, LogOnly could identify only 18% of the bugs in the dataset. These results highlighted BugSpot's capability in recognizing BBs, effectively handling the majority of cases in the dataset, and surpassing the existing recognition mechanism. For BugSpot, the CV was 1% for Rec-Rate, and 4% for FP-Rate across ten rounds. For BugSpot+GT, the CV was 1.8% and 4.5% for the two metrics. This showed that BugSpot's performance remained consistent across different runs.

As shown by column "Avg. Rec-Rate" in Table 3, BugSpot failed to recognize bugs in 17% (=100%-83%) of the bug reports (false negative recognitions). We investigated the results to understand the reasons. For report collect#2191 [5], the only crash bug that BugSpot failed to recognize, we found the reason was that the bug report provided an incorrect stack trace. The reasons for the other failure cases were mainly twofold. One reason was that the tool incorrectly generated the BugIR expression due to the issues discussed in Section 5.3.2. Consequently, BugSpot could not recognize the bug. For these cases, after being provided with the ground truth BugIR expression, BugSpot could correctly recognize the bugs, achieving a higher successful recognition rate as shown in Table 3. The second reason was that the app's UI did not offer enough textual information about the UI widget. Since BugSpot relies on UI details to locate and identify the affected UI widget, without that information, BugSpot was unable to find a match and identify the BB.

As shown by the column "FP-Rate" in Table 3, BugSpot falsely recognized some non-buggy states during the reproduction process as buggy. Upon further investigation, we identified two common cases of false recognition. One was when the BB caused the display of an abnormal widget

Table 4. Results of EQ 3 - Performance on Reproducing Bug Reports

	Tool	#TPs	#FPs	#TNs	#FNs
Roam	with BugSpot	48	8	26	5
	without BugSpot	14	0	30	43
ReproBot	with BugSpot	24	13	45	5
	without BugSpot	5	1	57	24
ReCDroid	with BugSpot	14	14	58	1
	without BugSpot	2	2	69	14

and the app happened to have similar abnormal UI widgets on non-buggy UI screens. In this case, BugSpot falsely reported the BB after finding these widgets. Another case in which BugSpot made mistakes was when the buggy behavior caused a UI widget to not be displayed. For example, in report GPSTest#330, the buggy behavior was "No E Explanation" (E is a certain app element). In this case, the affected UI widget, "E Explanation", was not only not shown in the buggy UI but also not shown in other UIs.

5.5 EQ 3: Usefulness for Automated Reproduction Techniques

5.5.1 Protocol. Although the results of EQs 1 and 2 showed that our approach can extract BugIR expressions and recognize BBs in bug reports with high accuracy, the results were not perfect, and this could impact the overall usefulness of our approach. Therefore, the goal of this EQ is to evaluate the impact of our approach on the automated bug reproduction process. To do this, we integrated BugSpot with three state-of-the-art automated reproduction approaches: Roam [75], ReproBot [79] and ReCDroid [78]. In the experiment, we ran the three reproduction techniques with BugSpot to reproduce the bug reports in our dataset with a 15-minute time limit per report. As a comparison, we also ran the reproduction techniques without BugSpot to reproduce the reports.

For each reproduction, we manually reviewed the UI event traces generated by the reproduction tools to verify the reproduction results. During the manual checking, we tracked the number of true positives (TPs), false positives (FPs), true negatives (TNs), and false negativess (FNs) of the reproduction results. Specifically, TPs and TNs represent the bug reports for which the reproduction tool correctly reported the reproduction results (success or failure); FPs occurred when the reproduction tool reported a successful reproduction, but it actually failed to reproduce the bug; and FNs occurred when the reproduction tool reported a failure, but we found that it actually successfully reproduced the bug in one of the UI events traces. These four metrics together covered different scenarios of the reproduction results. We also estimated how much developer effort could be saved by using BugSpot to recognize the bug instead of manual checking. To estimate this effort, we counted the number of UI event traces that had to be manually reviewed for each bug report. We used this metric because it represents the workload that a developer would have to filter in some way to recognize the manifestation of the BB. We also measured BugSpot's running time during the reproduction to assess the additional overhead caused by BugSpot.

5.5.2 Presentation of Results. Table 4 displays the results of each reproduction technique when using BugSpot ("with BugSpot") and not using BugSpot ("without BugSpot"). The columns in the table list the number of TPs, FPs, TNs, and FNs of the reproduction results. Table 5 shows the number of UI event traces we manually reviewed for the reports where BugSpot accurately determined the reproduction result. It lists the median, max and mean number of UI Event traces

Table 5. Results of EQ 3 - Number of Manually Reviewed UI Event Traces

Tool	Median	Max	Mean	Total
Roam	1	25	5	338
ReproBot	12	28	11	704
ReCDroid	14	50	17	1,221

for each bug report, and also the total traces for all bug reports. The average running time of BugSpot on each reproduction was 32, 56, and 80 seconds when integrated with Roam, ReproBot, and ReCDroid, respectively.

5.5.3 Discussion of Results. As shown by the results in Table 4, by using BugSpot, the reproduction techniques correctly reported reproduction results (i.e., TPs and TNs) for an average of 71 (82%) bug reports. BugSpot's ability to recognize different types of bugs helped the three reproduction techniques reproduce more bug reports when using BugSpot (shown by column "#TPs") than without. Specifically, Roam reproduced 34 more bugs, ReproBot reproduced 19 more bugs, and ReCDroid reproduced 12 more bugs. In contrast, when not using BugSpot, reproduction techniques could not recognize the described bug in the reports and categorized more reproduction results as failures, resulting in a higher number of TNs and FNs. It is worth noting that BugSpot achieved this improvement in a fully automated way. As shown in Table 5, if developers were to manually check these results, they would need to review an average of 11 execution traces per bug report and 754 execution traces for the entire dataset. These results showcased that BugSpot could help existing reproduction techniques recognize more bugs and, therefore, automatically reproduce more bug reports.

Despite the improvements brought about by BugSpot, the reproduction tools also reported a few incorrect reproduction results when using BugSpot (shown by columns "#FPs" and "#FNs" in Table 4). Through manual analysis of the results, we determined that the reasons behind these mistakes were the same as the ones we found in Section 5.4.3.

BugSpot increased the running time per reproduction by 32 to 80 seconds. This represented an 8% increase in the total running time of the reproduction tool on a bug report. Nearly all of this overhead was spent querying the LLM. We found that the wide range of the overhead was caused by differences in the reproduction tools' approaches. Reproduction tools that explored more possible UI event sequences invoked BugSpot more often to check the reproduction results, consequently causing more overhead. Overall, our results showed that although BugSpot imposed additional overhead on the reproduction process, the overhead was relatively small compared to the overall running time of the reproduction tool and was generally proportional to the number of times BugSpot was invoked.

5.6 Discussion

5.6.1 Limitations. Our tool recognizes bugs by analyzing the descriptions of their physical manifestations provided in bug reports. While this capability allows it to identify various types of bugs, as shown in our evaluation, our tool is limited in detecting bugs that either have no physical manifestations or are insufficiently described in bug reports. Additionally, the definition of BugIR encompasses all the common manifestations identified in our study (Section 3.2.4), allowing it to recognize various types of bugs. BugIR is also designed to be easily extensible by allowing more Types, Attributes, and Operations to be added. However, the current design of BugIR may not

accommodate all the varied bug descriptions in real-world reports. Future work can be done to extend the definition of BugIR to support specific bug manifestations.

5.6.2 Threats to External Validity. A potential threat to the external validity of our results is the representativeness of the selected bug reports. We attempted to overcome this threat by collecting bug reports from previous related research, and all of the subjects are real-world Android bug reports from either GitHub Issues [16] or Google Code [17]. Another potential threat is the representativeness of the reproduction techniques used in EQ3's experiments. To mitigate this threat, we used three state-of-the-art reproduction techniques. These techniques are publicly available and commonly selected for comparison in other related works [43, 51]. In addition, the choice of a single LLM in our experiments poses a threat to the generalizability of the results across other LLMs. To address this concern, we repeated the experiments again using a widely recognized open-source LLM, Llama3.1-70B [24], with its parameter "temperature" set to zero. The new experiments revealed that our approach achieved 90% precision and 88% recall in generating BugIR expressions, along with an 83% recognition rate and a 16% false positive rate in identifying buggy behaviors. The results indicated that the performance of our approach remains similar between both proprietary and open-source LLMs.

5.6.3 Threats to Internal Validity. A threat to the internal validity is the potential biases that influence the manual creation of the BugIR expression ground truth in EQ 1 and the manual validation of the reproduction results in EQ 3. To mitigate the influence of this threat, both tasks involved two authors. Each of them conducted the task separately and then compared the results to reduce the potential biases. Another threat comes from the inherent randomness in LLM-generated responses. To mitigate this effect, we conducted the experiments ten times, and the results of our experiment exhibited a CV ranging from 1% - 5% (much lower than the threshold 30% [33]), which showed the high consistency of the results.

6 Related Work

Mobile App Bug Report Reproduction There have been many research efforts devoted to automatically reproducing bug reports for mobile applications using the textual reproduction steps [39, 41, 44, 51, 66, 76, 78], video recordings [31, 40], or the crash stacktrace [36, 43, 57]. Our work complements these works by addressing existing work's limitation on bug recognition in the reproduction process. Specifically, existing works, such as ReCDroid [78], ReproBot [76] and Roam [75], can only reproduce crash bug reports due to the lack of an effective mechanism for recognizing other types of bugs during reproduction. Our work addresses this problem by introducing an automated technique to recognize various types of buggy behavior based on mobile app bug report text. As shown in the evaluation section, our approach was able to effectively recognize the buggy behavior and improve the performance of existing reproduction techniques.

Empirical Study on Mobile App Bug Reports There are several studies on mobile applications bug reports. Johnson et al. [46] studied the quality of information in bug reports and their relation to bug reproduction. Xiong et al. [71] conducted a comprehensive study on functional bugs in mobile apps and concluded their root causes and common oracles. Wang et al. [65] studied how images can be used in bug report reproduction. Compared to these studies, to the best of our knowledge, our paper is the first to explore the location of BB descriptions in mobile app bug reports, how the BBs can manifest in mobile apps, and how the manifestations are described in bug reports. Chaparro et al. [35] studied mobile app bug reports, and concluded syntactical patterns to identify sentences describing the buggy behaviors. In contrast, our study focuses on how the sentences semantically describe the manifestation of the bug. The patterns identified in Section 3.2.4 aim to characterize

and abstract bug manifestations, allowing systematic evaluation during reproduction to determine if the bug is triggered.

LLM for Mobile App Analysis LLMs have demonstrated a significant impact and utility across different research topics in mobile app analysis. Liu et al. leveraged LLMs in generating input values [54], simulating human-like moves for mobile app testing [55, 73]. Feng et al. [41] and Wang et al. [66] leveraged LLMs to execute reproduction steps in mobile app bug reports. Different from existing works, our approach leverages the LLM to recognize the buggy behavior description of a bug, thereby improving the automated reproduction of mobile app bug reports.

Non-Crash Bug Detection for Mobile Apps Many research works have focused on defining common test oracles for detecting non-crash bugs in mobile apps [30, 72, 74]. Different testing techniques have also been applied to detect non-crash bugs automatically. For example, both Xiong et al. [71] and Wang et al. [67] employed differential testing to detect non-crash functional bugs. Additionally, property-based testing has been applied to this problem [62, 70]. Researchers have also explored the use of LLMs for detecting non-crash functional bugs [47]. Furthermore, numerous studies proposed techniques for detecting specific types of functional bugs, including accessibility bugs [49, 56, 58], data issues [42, 62], UI scaling issues [61], and system setting issue [63]. The main distinction between our work and previous studies is that our approach identifies a specific bug based on the bug report. In contrast, the previous works do not consider this aspect, which limits their applicability in the bug report reproduction process.

7 Conclusion

In this paper, we proposed a novel approach to automatically recognize the buggy behavior of the described bug during the automated reproduction of the bug report. To design our approach, we conducted an empirical study on the buggy behavior descriptions in real-world bug reports. The empirical evaluation showed that our approach is effective in recognizing bugs and improves the performance of existing automated reproduction techniques.

8 Data Availability

In the replication package [1], we provide the coding results for our empirical study, source code and documentation of BugSpot, the dataset we used for the evaluation, and the detailed evaluation results.

Acknowledgments

This work was supported by the U.S. National Science Foundation (NSF) under grants CCF-2211454 and CCF-2403747.

References

[1] 2014. Replication Package. https://github.com/USC-SQL/BugSpot-Artifact.
[2] 2019. GPT-4 website. https://openai.com/index/gpt-4/.
[3] 2019. Issue 121 for ATimeTracker Github Repository. https://github.com/netmackan/ATimeTracker/issues/121.
[4] 2019. Issue 184 for MaterialFiles Github Repository. https://github.com/zhanghai/MaterialFiles/issues/184.
[5] 2019. Issue 2191 for Collect Github Repository. https://github.com/getodk/collect/issues/2191.
[6] 2019. Issue 634 for Omni-Notes Github Repository. https://github.com/federicoiosue/Omni-Notes/issues/634.
[7] 2019. Notifications are not cleared on display of chat. https://github.com/deltachat/deltachat-android/issues/725.
[8] 2019. Podcast Cover Disappears on Device Rotation #2992. https://github.com/AntennaPod/AntennaPod/issues/2992.
[9] 2019. ReCDroid Github repository. https://github.com/AndroidTestBugReport/ReCDroid.
[10] 2019. ReproBot's Github repository. https://github.com/USC-SQL/ReproBot-Artifact.
[11] 2019. Roam's Replication Package. https://zenodo.org/records/11068809.
[12] 2020. Chromecast controls disappear immediately. https://github.com/jellyfin/jellyfin-android/issues/459.

[13] 2020. I can't gesture type words with the first letter upper-case and others lower-case. https://github.com/AnySoftKeyboard/AnySoftKeyboard/issues/2825.
[14] 2020. Impossible to reproduce any video in Android App. https://github.com/nextcloud/android/issues/7602.
[15] 2021. Audio file playing can't be stopped unless app is closed. No media controls to be found. https://github.com/nextcloud/android/issues/8905.
[16] 2023. Github Issue Tracker. https://github.com/issues.
[17] 2023. Google Code Issue Tracker. https://code.google.com/archive/.
[18] 2023. Sample size determination. https://en.wikipedia.org/wiki/Sample_size_determination.
[19] 2024. Android ADB Shell. https://developer.android.com/tools/adb.
[20] 2024. Android Checkbox. https://developer.android.com/reference/android/widget/CheckBox
[21] 2024. Android Crash Handler. https://developer.android.com/games/optimize/crash.
[22] 2024. FDroid. https://f-droid.org/en/.
[23] 2024. GitHub. https://github.com.
[24] 2024. Hugging Face: Llama-3.1-70B. https://huggingface.co/meta-llama/Llama-3.1-70B.
[25] 2024. Keyboard keeps showing after opening settings menu. https://github.com/flex3r/DankChat/issues/66.
[26] 2024. Openhab Android Issue #2523. https://github.com/openhab/openhab-android/issues/2583.
[27] 2024. SpotiFlyer Issue #2523. https://github.com/Shabinder/SpotiFlyer/issues/764.
[28] 2024. UI Automator. https://developer.android.com/training/testing/other-components/ui-automator.
[29] 2024. Unstoppable Wallet Issue #3763. https://github.com/horizontalsystems/unstoppable-wallet-android/issues/3763.
[30] Kesina Baral, Jack Johnson, Mattia Fazzini, Julia Rubin, Junayed Mahmud, Sabiha Salma, Jeff Offutt, and Kevin Moran. 2024. Automating GUI-based Test Oracles for Mobile Apps. In *Proceedings of the 21st International Conference on Mining Software Repositories*. Lisbon Portugal.
[31] 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*. ACM, Seoul South Korea, 309–321. https://doi.org/10.1145/3377811.3380328
[32] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. In *2013 17th European Conference on Software Maintenance and Reengineering*. 133–143. https://doi.org/10.1109/CSMR.2013.23 ISSN: 1534-5351.
[33] Charles E. Brown. 1998. *Coefficient of Variation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 155–157. https://doi.org/10.1007/978-3-642-80328-4_13
[34] 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
[35] 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
[36] Ning Chen and Sunghun Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering* 41, 2 (Feb. 2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469 Conference Name: IEEE Transactions on Software Engineering.
[37] Juliet M. Corbin and Anselm L. Strauss. 2015. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. SAGE.
[38] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 308–318. https://doi.org/10.1109/ASE.2017.8115644
[39] 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*. ACM, Amsterdam Netherlands, 141–152. https://doi.org/10.1145/3213846.3213869
[40] 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*. ACM, Pittsburgh Pennsylvania, 1045–1057. https://doi.org/10.1145/3510003.3510048
[41] Sidong Feng and Chunyang Chen. 2024. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, Portugal.
[42] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 605–616. https://doi.org/10.1145/3533767.3534402

[43] 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*. ACM. https://dl.acm.org/doi/abs/10.1145/3597503.3623298

[44] Yuchao Huang, Junjie Wang, Liu Zhe, 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, Melbourne, Australia.

[45] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2019. Characterizing Android-Specific Crash Bugs. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 111–122. https://doi.org/10.1109/MOBILESoft.2019.00024

[46] 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

[47] Bangyan Ju, Jin Yang, Tingting Yu, Tamerlan Abdullayev, Yuanyuan Wu, Dingbang Wang, and Yu Zhao. 2024. A Study of Using Multimodal LLMs for Non-Crash Functional Bug Detection in Android Apps. In *Proceedings of the 31st Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Chongqing, China. https://doi.org/10.48550/arXiv.2407.19053 arXiv:2407.19053 [cs].

[48] Klaus Krippendorff. 2004. *Content Analysis: An Introduction to Its Methodology (second edition)*. Sage Publications.

[49] Arun Krishna Vajjala, S M Hasan Mansur, Justin Jose, and Kevin Moran. 2024. MotorEase: Automated Detection of Motor Impairment Accessibility Issues in Mobile App UIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639167

[50] Yuanhong Lan, Yifei Lu, Minxue Pan, and Xuandong Li. 2024. Navigating Mobile Testing Evaluation: A Comprehensive Statistical Analysis of Android GUI Testing Metrics. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 944–956. https://doi.org/10.1145/3691620.3695476

[51] Gaoyi Lin, Zhihua Zhang, and Zhanqi Cui. 2023. Widget Hierarchy Graph Guided Crash Reproduction Method for Android Applications (S). In *The 35th International Conference on Software Engineering and Knowledge Engineering, {SEKE} 2023*. 584–587. https://doi.org/10.18293/SEKE2023-066

[52] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. 2017. An Empirical Study on Android-Related Vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2–13. https://doi.org/10.1109/MSR.2017.60

[53] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[54] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-Aware Automated Text Input Generation for Mobile GUI Testing. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 1355–1367. https://doi.org/10.1109/ICSE48619.2023.00119

[55] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639180

[56] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: spotting UI display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 398–409. https://doi.org/10.1145/3324884.3416547

[57] Maryam Masoudian, Heqing Huang, Morteza Amini, and Charles Zhang. 2024. Mole: Efficient Crash Reproduction in Android Applications With Enforcing Necessary UI Events. *IEEE Transactions on Software Engineering* 50, 8 (Aug. 2024), 2200–2218. https://doi.org/10.1109/TSE.2024.3428543 Conference Name: IEEE Transactions on Software Engineering.

[58] Forough Mehralian, Navid Salehnamadi, Syed Fatiul Huq, and Sam Malek. 2023. Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3551349.3560424

[59] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. 2020. Why are Some Bugs Non-Reproducible? : –An Empirical Investigation using Data Fusion–. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Adelaide, Australia, 605–616. https://doi.org/10.1109/ICSME46990.2020.00063

[60] Jake Snell, Kevin Swersky, and Richard Zemel. 2017. Prototypical networks for few-shot learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4080–4090.

[61] Yuhui Su, Chunyang Chen, Junjie Wang, Zhe Liu, Dandan Wang, Shoubin Li, and Qing Wang. 2023. The Metamorphosis: Automatic Detection of Scaling Issues for Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3551349.3556935

[62] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1088–1100. https://doi.org/10.1145/3611643.3616286

[63] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3460319.3464806

[64] Anthony J. Viera and Joanne Mills Garrett. 2005. Understanding interobserver agreement: the kappa statistic. *Family medicine* 37 5 (2005), 360–3. https://api.semanticscholar.org/CorpusID:38150955

[65] Dingbang Wang, Zhaoxu Zhang, Sidong Feng, William G.J. Halfond, and Tingting Yu. 2025. An Empirical Study on Leveraging Images in Automated Bug Report Reproduction. In *Proceedings of the 22rd International Conference on Mining Software Repositories*.

[66] 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 the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1048–1060. https://doi.org/10.1145/3650212.3680341

[67] 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 (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 434–446. https://doi.org/10.1145/3540250.3549170

[68] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, 24824–24837.

[69] Tyler Wendland, Jingyang Sun, Junayed Mahmud, S. M. 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, Madrid, Spain, 600–604. https://doi.org/10.1109/MSR52588.2021.00082

[70] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/3691620.3694986

[71] 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*. ACM, Seattle WA USA, 1319–1331. https://doi.org/10.1145/3597926.3598138

[72] Ali Asghar Yarifard, Saeed Araban, Samad Paydar, Vahid Garousi, Maurizio Morisio, and Riccardo Coppola. 2024. Extraction and empirical evaluation of GUI-level invariants as GUI Oracles in mobile app testing. *Information and Software Technology* (July 2024), 107531. https://doi.org/10.1016/j.infsof.2024.107531

[73] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. http://arxiv.org/abs/2311.08649 arXiv:2311.08649 [cs].

[74] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, Cleveland, OH, USA, 183–192. https://doi.org/10.1109/ICST.2014.31

[75] Zhaoxu Zhang, Fazle Mohammed Tawsif, Komei Ryu, Tingting Yu, and William G. J. Halfond. 2024. Mobile Bug Report Reproduction via Global Search on the App UI Model. *Reproduction Package for "Mobile Bug Report Reproduction via Global Search on the App UI Model"* 1, FSE (July 2024), 117:2656–117:2676. https://doi.org/10.1145/3660824

[76] Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William G.J. 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. ACM, Seattle WA USA, 411–422. https://doi.org/10.1145/3597926.3598066

[77] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William G. J. Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Transactions on Software Engineering and Methodology* 31, 3 (July 2022), 1–33. https://doi.org/10.1145/3488244

[78] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 128–139. https://doi.org/10.1109/ICSE.2019.00030

[79] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 423–435. https://doi.org/10.1109/ICSE43902.2021.00048

[80] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering* 36, 5 (Sept. 2010), 618–643. https://doi.org/10.1109/TSE.2010.63

Received 2024-09-13; accepted 2025-04-01