Toward the Automated Localization of Buggy Mobile App UIs from Bug Descriptions

Antu Saha
William & Mary
Williamsburg, Virginia, USA
asaha02@wm.edu

Yang Song
William & Mary
Williamsburg, Virginia, USA
ysong10@wm.edu

Junayed Mahmud
University of Central Florida
Orlando, Florida, USA
junayed.mahmud@ucf.edu

Ying Zhou
George Mason University
Fairfax, Virginia, USA
yzhou29@gmu.edu

Kevin Moran
University of Central Florida
Orlando, Florida, USA
kpmoran@ucf.edu

Oscar Chaparro
William & Mary
Williamsburg, Virginia, USA
oscarch@wm.edu

Abstract

Bug report management is a costly software maintenance process comprised of several challenging tasks. Given the UI-driven nature of mobile apps, bugs typically manifest through the UI, hence the identification of buggy UI screens and UI components (*Buggy UI Localization*) is important to localizing the buggy behavior and eventually fixing it. However, this task is challenging as developers must reason about bug descriptions (which are often low-quality), and the visual or code-based representations of UI screens.

This paper is the first to investigate the feasibility of automating the task of Buggy UI Localization through a comprehensive study that evaluates the capabilities of one textual and two multi-modal deep learning (DL) techniques and one textual unsupervised technique. We evaluate such techniques at two levels of granularity, Buggy *UI Screen* and *UI Component* localization. Our results illustrate the individual strengths of models that make use of different representations, wherein models that incorporate visual information perform better for UI screen localization, and models that operate on textual screen information perform better for UI component localization – highlighting the need for a localization approach that blends the benefits of both types of techniques. Furthermore, we study whether Buggy UI Localization can improve traditional buggy code localization, and find that incorporating localized buggy UIs leads to improvements of 9%-12% in Hits@10.

CCS Concepts

• **Software and its engineering** → **Maintaining software**.

Keywords

Bug Reports, Mobile Applications, UI Data, Information Retrieval

ACM Reference Format:

Antu Saha, Yang Song, Junayed Mahmud, Ying Zhou, Kevin Moran, and Oscar Chaparro. 2024. Toward the Automated Localization of Buggy Mobile App UIs from Bug Descriptions. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680357

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680357

1 Introduction

Bug report management is an essential, yet costly process for software projects, in particular for mobile apps [88]. It demands high developer effort [15, 22, 35, 72, 87, 88] due in part to the potential for large volumes of reported bugs and the varying quality of submitted bug reports. These reports are the central artifact in bug management [28, 35, 87, 88], as they directly impact downstream tasks such as bug triaging, reproduction, localization, program repair, and even regression testing. Bug reports typically describe defects found during software development and usage, and are expected to include, at minimum, the app's observed (incorrect) behavior (**OB**, the expected behavior (**EB**), and the steps to reproduce the bug (**S2Rs**) [21, 28, 50, 68, 69, 87].

Given the UI-centric nature of mobile apps, a large majority of reported bugs for these apps manifest through the UI [45]. Therefore, an important first step toward understanding, diagnosing, and resolving underlying bugs in the code is localizing the buggy behavior to both a UI screen and UI components [58]. As such, a critical bug report management task for mobile apps is the identification of UI screens and UI components (*e.g.*, buttons or text fields) that cause or display the reported incorrect behavior of the app (*i.e.*, the OB), a task that we term ***Buggy UI Localization***. This task is essential but can be difficult for developers, especially when many incoming bug reports need to be addressed and fail to include important details or graphical information (*e.g.*, buggy app screenshots [34]). Despite the growing body of work on automating bug report management tasks [88], prior work has not yet explored how to assist developers in Buggy UI Localization.

In this paper, we present the first empirical study that investigates *the feasibility of automatically localizing bug descriptions to UI screens and UI components of mobile apps.* Similar to traditional buggy code localization [14, 25–27, 40, 46, 76], we formulate Buggy UI Localization as a retrieval task, in which a bug description (*i.e.*, the **OB**) is used as query input to a retrieval engine that searches the space of UI screens and UI components of an app and recommends a ranked list of candidates that most likely correspond to the bug description. Specifically, the study focuses on two retrieval tasks for a given bug description: ***screen localization (SL)***, which

involves retrieving potentially buggy UI screens from the app, and ***component localization (CL)***, which aims to retrieve the relevant buggy UI components from a given buggy UI screen.

The study investigates how the textual and visual information from UI screens and UI components can be leveraged for Buggy UI Localization, and hence, explores the effectiveness of unsupervised textual techniques and pre-trained textual and multi-modal deep learning (DL) techniques. Specifically, we examine one unsupervised text-based model (Lucene [42]) and three DL models: a supervised text-based model (SentenceBert or SBert [67]), and two supervised vision-language learning models (Clip [65] and Blip [52]), under a zero-shot setting to explore their capabilities for Buggy UI Localization. To evaluate the effectiveness of the models in real-world scenarios, we created a manually curated dataset of 228 OB descriptions from 87 real bug reports. The dataset also includes associated buggy UI screens and UI components that we manually labeled from a UI corpus created by employing GUI app exploration techniques [59], for 39 Android mobile apps.

The results of our study indicate that no single technique universally performs best for the two localization tasks (screen and component localization). The best-performing approaches suggest the correct buggy UI screens (Blip) and UI components (SBert) in the top-3 recommendations for 52% and 60% of the bug descriptions, respectively. We also found the models tend to perform better for higher-quality bug descriptions and easier-to-retrieve cases as judged by humans. The results show the feasibility and effectiveness of using existing DL models for Buggy UI Localization.

To illustrate the practical usefulness of automated Buggy UI Localization, we conducted a second empirical study that investigated how identified buggy UI screens from the best-performing screen localization model can improve traditional buggy code localization approaches. We adapted the approach proposed by Mahmud *et al.* [58] to filter or boost code files retrieved via existing buggy code localization approaches using retrieved UI screen information by screen localization. We designed an end-to-end, automated approach comprising two major steps: (1) ***Buggy UI Localization***, which receives a bug description and the app UI screens and automatically identifies buggy UI screens, and (2) ***Buggy Code Localization***, which (i) computes the textual similarity between the bug report and the app code files to retrieve potentially buggy files, and (ii) boosts the rankings of retrieved code files related to identified buggy UI screens in step (1). Using two buggy code localization tools applied to 79 bug reports we found that incorporating information from the automatically identified Buggy UI screens can lead to a 9% to 12% improvement in Hits@10, compared to baseline techniques that do not use UI data.

In summary, we make the following contributions:

(1) We are the first to address the problem of **Buggy UI Localization** by investigating how different information sources (textual UI metadata and app screenshots) and four existing textual and multimodal models can be leveraged for Buggy UI Localization. Our results suggest that the models perform differently depending on the retrieval task. These findings can inform the design of future domain-specific models.
(2) We illustrate a practical application of buggy UI screen localization on **Buggy Code Localization**. The application of our screen localization approach to Mahmud *et al.*'s approach [58] illustrates that it can both automate and improve upon existing buggy code localization techniques.
(3) We provide a novel, publicly available benchmark (data, infrastructure, results, and documentation) for Buggy UI Localization, which facilitates replication and experimentation [12]. The benchmark provides a new, manually-curated dataset with buggy UI screens and UI components, textual and visual retrieval corpora, and bug descriptions for each bug report.

Title: Can no longer enter text in SSID Filter TextView

Description: Cannot enter any text in the SSID Filter field.
Steps:
1. Click on Filter icon.
2. Click/tap on SSID Filter text field.
3. Keyboard does not pop up.

Expected Behaviour:
Should display keyboard and allow you to enter SSID filter text.

Figure 1: Bug report #191 from the WiFi Analyzer app [10]

2 Background, Problem, & Motivation

2.1 Bug Descriptions & App UI Elements

In this paper, a ***bug description*** is the observed or incorrect app behavior (**OB**) textually described in a sentence of a bug report. We focus on descriptions of bugs that manifest visually on the device screen. Figure 1 shows a real bug report for WiFi Analyzer [10], an app for monitoring the strength and channels of surrounding WiFi networks [9]. The bug/OB descriptions in the bug report are underlined in Figure 1 and describe a bug in which the app fails to show the keyboard to enter the WiFi's SSID.

App ***UI screens*** implement one or more app features and represent the canvas upon which UI components (*a.k.a.* widgets) are drawn. ***UI components*** are elements rendered on a UI screen (*e.g.*, buttons, text fields, or checkboxes) that allow end-users to interact with the application. A screen is composed of a hierarchy of UI components and containers (*a.k.a.* layouts) that group UI components together [7]. Figure 2 shows examples of UI screens (2b) and their components (2c) for the WiFi Analyzer app. In this paper, a UI screen is represented as a *screenshot* and its corresponding *UI hierarchy* of components/containers described in metadata. Each UI component is represented by a set of *attributes*, including the component type (*e.g.*, TextView or Button [7]), its label or text shown on the screen, an ID, a description, and various visual properties such as the component's visibility and size. **Buggy UI Screens and UI Components** display unexpected, incorrect behavior of an app.

2.2 Problem and Motivating Example

We envision a system that suggests to the developer a ranked list of UI screens (*i.e.*, app screenshots) that display or is related to the buggy app behavior reported by a bug description in a bug report (see Figure 2b). The developer would then inspect the suggested UI screens in the ranked list (in a top-down fashion) and select one or more screens that s/he deems display the reported bug. The system would then identify (and highlight) the UI components in

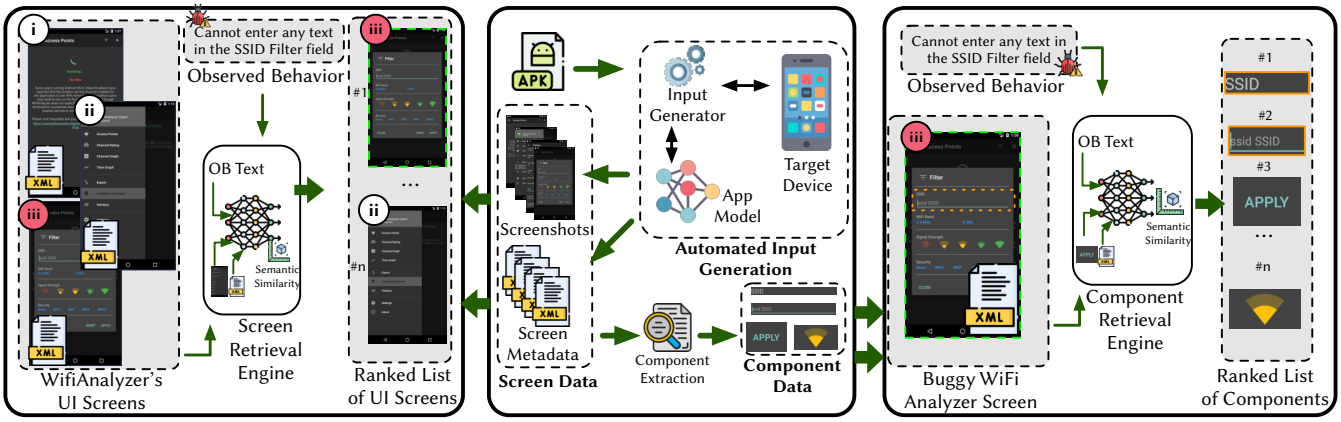

Figure 2b : Screen Localization (SL) **Figure 2a : GUI Info Extraction** **Figure 2c : Component Localization (SL)**

Figure 2: Example of the UI screen/component localization process for an OB/bug description of the WifiAnalyzer app [10].

the selected buggy UI screens that are most related to the reported bug (see Figure 2c). The suggestions of this system can help developers not only automatically localize buggy UI screens and UI components [13, 23, 62, 80, 87], but also understand the reported bug, and assist them in other bug management activities (*e.g.*, bug reproduction). Additionally, this system can be useful for various bug report management tasks, as it can provide information to existing automated techniques that aim to reproduce bugs [83, 84], generate test cases [37], assess the quality of bug reports [24, 70, 71], and perform buggy code localization [58].

While bug reports provide the steps to reproduce the bug (S2Rs) and the expected app behavior (EB), which can be used to identify the buggy UI screens and UI components, we focus on OB descriptions for at least two reasons [28]: (1) they convey the faults observed by the user, and (2) they are often written using different wordings (even for a single bug type—see fig. 1). The S2Rs and EB in bug reports do not necessarily describe a bug and they are often described using a more limited language compared to that of OBs [28].

We formulate automated buggy UI localization as two retrieval tasks (see Figure 2): *screen* and *component* localization. In ***screen localization (SL)***, a bug/OB description (*i.e.*, the ***query***) is the input to a retrieval engine that searches the space of (automatically identified) UI screens (see Figure 2a) of a given app and retrieves a list of UI screens ranked by their similarity to the bug description, which indicates the likelihood of a UI screen to show or be affected by the bug described by the query. Figure 2b illustrates the screen localization process for one OB description from the bug report shown in fig. 1. The highlighted UI screen with the green border (see Screen (iii) of Figure 2b) is the buggy screen (initially unknown to the developer). The two best approaches we studied (Blip & SBert) are able to retrieve the buggy screen as their first suggestion. In ***component localization (CL)***, the retrieval engine searches the space of (automatically identified) UI components (see Figure 2a) of a given buggy UI screen and retrieves a list of UI components ranked by a similarity score that indicates the likelihood of the components to show or be affected by the bug. Figure 2c illustrates the component localization process for the buggy UI screen of the bug description. The UI components in orange are the ones that the bug description refers to, hence they are expected to be ranked higher by the component retrieval engine. The two best-performing approaches we studied (SBert & Blip), rank the buggy components in the first position(s).

Screen and component localization are impacted by the amount of information that a OB description contains (*i.e.*, ***query quality***) and the difficulty in retrieving buggy UI screens/UI components (*i.e.*, ***retrieval difficulty***). If the bug description is poorly written or does not provide enough information about the bug (which is not uncommon in bug reports [24, 71]), then a retrieval engine (or even a human) would have a hard time identifying the buggy UI screens and UI components (if not familiar enough with the app). This problem is exacerbated by the fact that the same bug can be described in a variety of ways [28] – *e.g.*, see the underlined sentences in fig. 1. Even if the OB is clear and informative, identifying the buggy UI screens/UI components can be challenging when numerous similar UI screens/UI components exist in the app. As an example, consider the last OB/bug description from the Wifi Analyzer app shown in fig. 1: "Keyboard does not pop up". The best approach for screen localization, Blip, retrieved the true buggy screen at the 21st position and true buggy component at the 16th position. Component localization's best approach, SBert, retrieved the true buggy screen at the 6th position and the true buggy components at the 12th position. This illustrates the difficulty of buggy UI localization, hence in our study, we assess the performance of various approaches considering bug descriptions of different quality and retrieval difficulty levels.

3 Automating Buggy UI Localization

This study aims to investigate different methods for automatically locating buggy UI screens and UI components based on bug/OB descriptions and measure their effectiveness for this problem. To that end, we investigate existing retrieval approaches that leverage textual and/or visual information from the bug descriptions and UI screens and components, to perform screen and component localization. With this in mind, we address three research questions (RQs):

RQ$_1$: *How effective are retrieval approaches at locating buggy UI screens (SL) from bug descriptions?*

RQ$_2$: *How effective are retrieval approaches at locating buggy UI components (CL) from bug descriptions?*

RQ$_3$: *How effective are retrieval approaches for different query quality and retrieval difficulty levels?*

To answer the RQs, we selected four (un)supervised approaches of various kinds (section 3.1). Then, we constructed a real-world dataset for evaluating the effectiveness of the approaches (section 3.2). We executed the approaches (section 3.3) and measured their performance with standard retrieval metrics (section 3.4). This section provides details about these steps, while sections 3.5 and 3.6 present and discuss the obtained results.

3.1 Retrieval Approaches

We investigated three deep learning (DL)-based approaches and one baseline unsupervised approach, which support text-to-text or text-to-image retrieval for the two Buggy UI Localization tasks.

SentenceBert (or **SBert**) [67] is a neural text-based language model, which augments the traditional Bert model [33] with siamese and triplet networks. It can establish semantic similarity for a pair of textual descriptions by generating embeddings. SBert can be utilized for both screen and component localization using the textual bug descriptions, UI screens, and UI components.

Clip [65] is a neural multi-modal vision/language model that can learn semantic embeddings from text and images via a contrastive architecture. Given a text-image pair, it can determine the similarity between them. Hence, Clip can be utilized for both screen and component localization using textual information from the bug description and visual information from UI screen/components.

Blip [52] excels in vision-language understanding and generation tasks, and uses a multi-modal encoder-decoder component (MED) and a dataset bootstrapping method, *i.e.*, captioning-filtering (CapFilt). We used a Blip version optimized for text-image retrieval tasks, which only implements contrastive and matching losses. Blip can be utilized the same way Clip is used for buggy UI localization.

Finally, we selected **Lucene** [42] as a baseline technique for text retrieval. Lucene is a classical unsupervised approach that combines the vector space model (VSM), based on the TF-IDF representation, and the boolean text retrieval model, to compute the (cosine) similarity between a query and a document. Lucene can be utilized for both screen and component localization using the textual information in bug descriptions, UI screens, and UI components.

While Clip and Blip have been pre-trained with general-purpose data, they have performed well under zero-shot settings [52, 65, 67] for tasks such as semantic similarity computation, object detection, image captioning, and text-image retrieval, under distinct domains. These models have also been fine-tuned for downstream tasks [18, 31, 66, 81], enhancing their capabilities. This study investigates the capabilities of these models for Buggy UI Localization.

The lack of a large, high-quality dataset specifically created for Buggy UI Localization prevents us from fine-tuning Clip and Blip. While the RICO dataset [8] would be a good dataset candidate for our task, it does not provide bug descriptions, and the mobile app screenshots included in RICO do not show any buggy behavior. As such, creating this dataset with real-life bug descriptions and buggy UI screens of mobile apps would demand an enormous effort that is beyond the scope of our work. A potential solution to create this dataset, which we leave for our future work, is creating synthetic bug descriptions using templates for different bug types and wordings, based on automatically modified RICO screenshots that show various buggy behaviors (*e.g.*, incorrect app output, crashes, non-crashing errors, cosmetic issues, and navigation misbehavior).

Besides the three DL models, we considered models specifically designed for mobile app UI understanding tasks, including UIBert [17], VuT [57], and Screen2Vec [53]. However, UIBert and VuT's source code and pre-trained models are not available and Screen2Vec would require a significant adaptation effort for our task as the model is only designed for generating UI screen embeddings from screen text and UI hierarchies; extra modules would be required to adapt this model for Buggy UI Localization. We should note, though, that we experimented with it as a zero-shot encoder to represent UIs and with an SBert model for representing bug descriptions, computing the cosine similarity on both embeddings to establish similarity. Unfortunately, this led to poor performance for both screen and component localization tasks, hence we decided to not report their performance in this paper.

Large language models (LLMs), *e.g.*, ChatGPT [63] and Llama [73], can be used for text-based Buggy UI Localization. However, studying their capabilities for this task likely warrants a separate study because it requires careful control of several factors to make for a fair comparison with non-LLMs, including addressing non-deterministic responses, possible data leakages, selection of bug reports, token limits, and prompting strategies. Studying the capabilities of LLMs for the task of Buggy UI Localization is part of our future work.

3.2 Dataset Construction

We built a dataset of real-life bug descriptions and relevant buggy UI screens and UI components to assess the effectiveness of the models in a realistic setting. The dataset construction process included:

(1) identifying bug/OB descriptions in a set of bug reports – these descriptions represent the **queries** used for retrieval.
(2) building the **UI screen corpus** and **the UI component corpus** used for retrieval. These corpora are constructed for the app corresponding to each bug report.
(3) identifying the **(ground truth) buggy UI screens and UI components** in the corresponding corpus, and assigning a quality and retrieval difficulty level to each query.

3.2.1 Bug Report Selection. Since one of our goals, later detailed in this paper (see section 4), is to assess the usefulness of Buggy UI Localization models for Buggy Code Localization, we followed a pragmatic approach to select the bug reports for this study. We started with the 80 bug reports of the buggy code localization dataset provided by Mahmud *et al.* [58] so that we could reuse the data for this study and the buggy code localization study reported in section 4. Mahmud *et al.*'s dataset was created based on the AndroR2 dataset [44, 75], which consists of 180 manually reproduced bug reports for popular open-source Android applications hosted on GitHub. Of the 180 bug reports, Mahmud *et al.* discarded 100 reports because they: (1) described bugs that were no longer reproducible, (2) included bug fixes in non-Java code files only, (3) were no longer publicly available, or (4) included ambiguous code changes or commit IDs. This resulted in 80 bug reports.

When collecting ground truth data for Buggy UI Localization (see section 3.2.4), we discarded one bug report (from the GnuCash app [1]) from the 80 bug reports of Mahmud *et al.*'s dataset because

we were unable to reproduce the reported bug, thus leaving 79 reports. To expand the set of bugs usable for this study, we selected 14 extra bug reports from the 100 discarded ones whose bug fixes were in XML resource files as opposed to Java code and discarded six reports because we obtained errors trying to collect the retrieval corpus for those reports (see section 3.2.3). This resulted in eight extra bug reports, for a total of 87 reports. Although Mahmoud *et al.* [58] could not use these 8 bug reports, they are usable for the buggy UI localization task since they are reproducible and we collected (ground truth) UI data for the corresponding applications.

From the **87 bug reports** (1 to 8 per app), 32 describe an output problem, 23 report an app crash, 23 describe a UI cosmetic issue, and 9 report a navigation problem. The bug reports correspond to 39 Android apps (*e.g.*, GnuCash [4], Mozilla Focus [2], K-9 Mail [6], WiFi Analyzer [10], Images to PDF [5]) of different domains (*e.g.*, finance tracking, web browsing, emailing, WiFi network diagnosis, and image conversion) and UI layouts.

3.2.2 Bug Description Identification. To identify bug/OB descriptions in the 87 bug reports, two authors inspected and annotated the 1807 sentences in the reports' title and description. Based on the definition of OB and the criteria to annotate OB sentences defined by Chaparro *et al.* [28], one author labeled each bug report sentence as either an OB or non-OB. Here, the OB sentences describe the buggy app behavior (*e.g.*, the underlined sentences in Figure 1). The second author then verified the annotations made by the first author, indicating agreement or disagreement. Out of the 1807 sentences across all bug reports, the authors reached agreement on the labels for 1774 sentences (≈98% agreement, 0.91 Cohen's kappa [3]). The authors solved disagreements via discussion and consensus. Reasons for disagreement included mostly mistakes and misinterpretations (*e.g.*, when sentences described root causes in the code, rather than UI faults). Finally, 228 sentences were identified as bug/OB descriptions for the 87 bug reports (2 or 3 OBs per bug report on average), which serve as queries for the UI screen and component localization tasks.

3.2.3 Retrieval Corpus Collection. To build the retrieval corpus for each bug description, we require the set of UI screens and UI components of the apps, including the buggy UI screens and UI components. To collect these data, we employed a semi-automatic app execution approach that consisted of: (1) a record-and-replay methodology (used in prior studies [32, 71]), and (2) an automated app exploration methodology (used by Chaparro *et al.* [24]). Both methodologies reduce the manual effort of collecting UI (meta)data.

The goal of the **record-and-replay methodology** was to collect the buggy UI screens for each bug report and the screens navigated while reproducing the bugs, including screenshots and related metadata. Two authors manually reproduced the reported bugs by executing the reproduction steps found in the bug report on a Pixel 2 Android emulator. While reproducing the bugs, the authors used the AVT tool [32, 71] to collect UI-event traces and a video showing the user interactions with the app and the bug itself [59, 60]. These traces were replayed on the emulator via the TraceReplayer tool [58] to automatically collect app screenshots, UI hierarchies, and metadata for the exercised app UI screens.

The goal of the **automated app exploration** was to collect as many UI screens as possible for building the corpus. We executed a version of the CrashScope tool [60] that implements multiple exploration strategies to interact with the UI components of app screens comprehensively, trying to exercise as many app screens as possible. In the process, CrashScope collects app screenshots and XML-based UI hierarchies/metadata for the exercised app UI screens, in the same manner as TraceReplayer.

Since these two methodologies can generate duplicate UI screens, we employed the approach by Chaparro *et al.* [24] to produce a unique set of UI screens for each of the 87 bug reports. This approach parses the UI hierarchies of the collected UI screens for an app and establishes the uniqueness between two screens: if they have the same hierarchical structure, based on component types, sizes, and parent-children relationships, they are considered the same screen and one of the two is used. This implies that two UI screens with the same structure but different textual information are considered the same UI screen. To create the UI component corpus for a given (buggy) UI screen, we parsed the UI hierarchy of the screen and identified the visible leaf components, which are typically the ones shown to the user on the mobile device. However, we discarded layouts and other containers, thus focusing on labels, buttons, text fields, and other UI components that users typically interact with.

This procedure resulted in UI screen corpora containing ≈26 UI screens per bug report on average, which are used for screen localization. The UI component corpora contain ≈17 UI components per (buggy) UI screen on average, which is used for component localization. A potential limitation of our corpus collection process, based on dynamic app exploration, is the possibility of missing UI screens for an app, which may affect models' performance. However, we evaluate every approach using the same collected retrieval corpus for each bug report, ensuring a fair evaluation. We discuss the (dis)advantages of a static analysis-based approach for corpus collection over dynamic analysis approaches in Section 6.

3.2.4 Ground Truth Construction. We used a rigorous data annotation procedure to identify the buggy UI screens for each bug report, and the buggy UI components for each buggy screen.

During multiple annotation sessions, four paper authors (*a.k.a.* annotators) first read and understood the reported bugs, watching (if needed) the bug reproduction video collected during the corpus collection step. Then, the annotators inspected the app screens from the corpus to identify the buggy screens shown in the video and marked them as such in a spreadsheet. The annotators identified and marked the buggy UI components in the same spreadsheet. Each bug report was assigned to two annotators, making sure the annotators had an even number of bug reports to annotate. For each bug report, the first annotator identified the buggy UI screen and UI components and then the second annotator validated whether the identified screens and components were indeed buggy. Both annotators followed the procedure described above, marking potential disagreements in a shared spreadsheet. At the end of each annotation session, the annotators discussed disagreements (mostly due to misinterpretation of the bugs), and reached a consensus to produce the final ground truth set of buggy UI screens and UI components.

Besides identifying the buggy UI screens and UI components, the annotators rated the quality of the bug/OB descriptions based on the amount of information they provided to understand the bug. Since Buggy UI Localization is performed using individual bug/OB

Table 1: Screen and component localization statistics

Statistic	Screen loc.	Component loc.
# of retrieval tasks/queries	228	254
# of hard-to-retrieve tasks	111	130
# of easy-to-retrieve tasks	117	124
Avg. # of buggy UI screens/comp.	2.06 (2)	1.86 (1)
Avg. of corpus size	25.97 (22)	17.11 (14)

Average (median) values per query/retrieval task

descriptions, the annotators judged the quality of each OB in a bug report independently. The detailed understanding of each bug report and the identified buggy UI screens and components assisted the annotators in assessing OB quality. The annotators agreed on a quality rating based on a 1-5 discrete scale. A rating of 1 means the bug description does not contain useful information to understand the problem. Conversely, a rating of 5 means the description contains complete information to understand the bug. A rating between 1 and 5 indicates that there is missing information in the OB that hinders bug comprehension. Additionally, the annotators marked each bug description as *easy* or *hard* to localize, based on the difficulty they encountered in identifying the buggy UI screens and UI components. A common reason why bug descriptions were judged as *hard to retrieve* was that multiple UI screens (and UI components) were similar, yet only one or a few were displaying the reported bug. During the reconciliation sessions, disagreements were discussed and solved to produce the final query quality and retrieval difficulty category for each bug description.

3.2.5 Summary of the Collected Retrieval Data. For screen localization (SL), each OB description (*i.e.*, the query) represents a unique UI screen retrieval task. Hence, our dataset contains 228 queries in total, with 2.1 buggy UI screens per query as ground truth and 26 UI screens in the corpus on average (see table 1).

For component localization (CL), each OB description can have multiple ground truth buggy UI screens, hence each combination of OB description and UI screen represents a single retrieval task. Based on this, we created 254 queries (or retrieval tasks), with 1.9 buggy UI components per buggy UI screen as ground truth, and 17 components in the corpus on average.

In summary, we collected: OB descriptions (*i.e.*, the queries), the retrieval corpora of UI screens and UI components for each query (including app UI screenshots and cropped component images, and their UI hierarchy with associated metadata: component text, ID, *etc.*), and ground truth buggy UI screens and UI components.

3.3 Execution of the Retrieval Approaches

Each retrieval approach processes the query and retrieval corpus differently. Some approaches rely solely on textual information, while others utilize both textual and visual information.

Clip and Blip leverage textual and visual information from the query and UI screens and components. The query for screen and component localization (SL & CL) is the text of a OB description. For SL, the corpus is all the screenshots of the application for which the bug is reported, while for CL, the corpus is all the cropped UI component images of a buggy UI screen. The models receive a text-image pair and produce a score indicating how similar the bug/OB description and each UI screen and UI component are.

Lucene and SBert leverage only the textual information from the query and UI screens and UI components. The query is the OB descriptions. As for the corpus, we extracted and concatenated the text found in UI component metadata (*i.e.*, the component ID, label, and type) to create textual documents for retrieval. For CL, each document is represented by the extracted document for a component. For SL, we concatenated the textual documents of the components in a given screen to form the textual document of a screen. Lucene and SBert compute a score that represents how similar the bug description and each textual document are. Only for Lucene, we applied standard textual pre-processing on the queries and documents (lemmatization, stop word removal, *etc.*).

The computed similarity scores yield a ranked list of UI screens and components. Higher-ranked screens and components in these lists are more likely to manifest or be associated with the bug.

3.4 Evaluation Metrics

We used standard retrieval metrics, widely used in prior studies [14, 32, 40, 51], to measure the effectiveness of the studied models:

- **Mean Reciprocal Rank (MRR)**: it gives a measure of the average ranking of the *first* buggy UI screen/component in the candidate list given by a model. It is calculated as: $MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$, for N queries ($rank_i$ is the rank of the first buggy UI screen/component for query i).
- **Mean Average Precision (MAP)**: it gives a measure of the average ranking of *all* the buggy UI screens/components for a query. It is computed as: $MAP = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{BU}\sum_{b=1}^{BU}P_i(rank_b)$, where BU is the set buggy UI screens/components for query i, $rank_b$ is the rank of the buggy UI screen/component b, and $P_i(k) = \frac{buggy\_elements}{k}$ is the number of buggy UI screens/components in the top-k candidates.
- **Hits@K (H@K)**: it is the percentage of queries for which a buggy UI screen/component is retrieved in the top-K candidates.

All metrics give a normalized score in [0, 1]—the higher the score, the higher the retrieval performance of the models. We executed the models and the baseline approach on the constructed query sets for screen and component localization and computed/compared the metrics between these approaches.

3.5 Results

We present and discuss the effectiveness of the results of the approaches for both screen (SL) and component localization (CL). We focus our discussion on MRR since the other metrics show similar trends to the MRR results for all the models. Our replication package contains the results of all the experiments we conducted [12].

3.5.1 ***RQ$_1$****: Screen Localization (SL) Results.* Table 2 shows the screen localization performance of the approaches for 228 queries. The results reveal that Blip performs the highest (0.457 MRR), outperforming the second best SBert (0.415 MRR) and the third best Lucene (0.411 MRR) with a relative improvement of 10.1% and 11.31% respectively. While Lucene outperforms Clip (0.381 MRR) by 7.87%, it fails to retrieve buggy screens in 18 cases (*i.e.*, 7.89% - not shown in the table). However, Lucene achieves a competitive H@1 to Blip (0.285). In terms of H@K, Blip outperforms the remaining models by a considerable margin. For example, it outperforms

Table 2: Screen localization (SL) results

Approach	MRR	MAP	H@1	H@2	H@3	H@4	H@5
Blip	0.457	0.443	0.285	0.447	0.518	0.592	0.671
SBert	0.415	0.385	0.259	0.390	0.456	0.526	0.557
Lucene	0.411	0.384	0.285	0.386	0.465	0.522	0.575
Clip	0.381	0.348	0.206	0.338	0.465	0.526	0.592

Table 3: Component localization (CL) results

Approach	MRR	MAP	H@1	H@2	H@3	H@4	H@5
SBert	0.517	0.504	0.339	0.512	0.598	0.701	0.744
Blip	0.424	0.405	0.244	0.417	0.500	0.567	0.614
Clip	0.413	0.399	0.244	0.386	0.472	0.567	0.618
Lucene	0.398	0.355	0.311	0.441	0.480	0.504	0.512

the models with a maximum relative improvement of 20.47% H@5 (compared to SBert). The models other than Blip achieve a similar H@5 (Clip: 0.592, Lucene: 0.575, and SBert: 0.557).

The results show that Blip is the most effective model for screen localization by a significant margin. This indicates that its rich representations, learned from images and text from other domains, can be transferred to the Buggy UI Localization problem. Also, the results imply that both sources of information (UI pixels and text) are beneficial for screen localization. Interestingly, SBert performs second and outperforms Clip with a relative improvement of 8.9% MRR. These results stem from the higher H@1-2 results achieved by SBert. This is interesting and somewhat unexpected as SBert only utilizes textual information (making it potentially less expensive to execute), while Clip uses both visual and textual information. Lucene is possibly the least expensive approach, and performs comparably to SBert, yet it fails to retrieve buggy UI screens in 7.8% of the cases. In line with the original Blip evaluation [52], Blip outperforms Clip, which can be explained by the models' architecture. Blip is a model particularly designed for textual-image matching that includes a matching loss for aligning text phrases and images, learning joint representations of both sources via a contrastive loss. In contrast, Clip aims to learn representations of both textual phrases and images in the same embedding space without performing any matching.

The results also indicate that there is still room for improvement, as the best model (Blip) can suggest the buggy UI screens in the top-1 to top-3 recommendations in about 29% to 52% of the cases (see the H@1-3 results in table 2). A more sophisticated model is required to perform screen retrieval more effectively. Such a model should leverage both the visual information of a screen image and the textual information from the UI metadata of that screen.

3.5.2 ***RQ$_2$****: Component Localization (CL) Results.* Table 3 shows the component localization results of all the approaches for 254 retrieval tasks. Note that although the number of OBs is 228, some OBs may have a different component corpus for retrieval, one for each buggy screen in the ground truth (*i.e.*, one OB/bug description may correspond to multiple buggy UI screens).

The results reveal that all supervised approaches perform higher (0.413+ MRR) than the baseline (Lucene), which achieves 0.398 MRR. SBert is the most effective of all approaches (0.517 MRR), significantly outperforming Blip, Clip, and Lucene by 21.86%, 25.24%, and 29.7%, respectively. The superiority of SBert is consistently observed across all the metrics and it can suggest the buggy UI components in the top-1 to top-3 results in about 34% to 60% of the cases.

As in screen localization, Blip slightly outperforms Clip, yet both models achieve similar performance for H@1, 4, and 5. Although Lucene (0.398 MRR) achieves similar performance to Clip's (0.413 MRR) and Blip's (0.424 MRR) and higher H@1, 2, it fails to retrieve buggy UI components in 66 of 254 tasks (25.98%).

Several observations can be derived from these results. First, the superiority of the supervised models compared to Lucene suggests that DL models are better for component localization. Second, there is still room for improving component localization: while the performance of the best model is not low, the performance is not very high either, which means that specialized models for component localization are needed. Third, the textual information present in the UI components of the screens seems to be highly effective in performing localization, as indicated by SBert results. Fourth, Blip's superiority over Clip stems from their architectural differences (as discussed in Section 3.5.1). Fifth, while it may be counter-intuitive that SBert outperforms the multi-modal approaches, we generally observed that OBs tend to describe the buggy components using a language that is more similar to the component text observed by the user, which a language model like SBert is specifically designed for. While Blip also leverages textual information from components, it does so based on the pixel data rather than the actual component text extracted from the UI metadata.

3.5.3 ***RQ$_3$****: Results by Query Quality and Retrieval Difficulty.*
Query Quality. Figure 3 shows the screen localization results (based on MRR) across different query quality ratings (from 1 to 5, 5 meaning most informative). The figure shows that while different approaches perform differently across the quality ratings, all models achieve the best performance for the most informative queries (*i.e.*, rating 5). Moreover, the performance trend is similar for all the models on the queries with quality ratings 4 and 5. Interestingly, of 18 queries for which Lucene fails to retrieve a buggy screen, eight of them have a rating of 1, and the MRR achieved for the remaining successful cases is relatively high.

Figure 4 shows the component localization results (based on MRR) across different query quality ratings. The figure shows a clear trend: the models tend to perform better for higher-quality queries (rating 4 & 5) than lower-quality queries (rating 1 & 2). As in screen localization, of 66 queries for which Lucene fails to retrieve the buggy components, most of them (43) have a rating of 1 or 2. Of the 17 queries with a rating of 1, Lucene fails to retrieve the UI components for 14 queries. For the remaining 3 queries, it cannot retrieve any relevant component resulting in a 0 MRR.

For screen localization, we found a medium-to-high positive correlation between the OB quality and the MRR results: a Spearman's correlation of 0.41 to 0.8 across all models except CLIP. For component localization, we found a high correlation: Spearman's correlation of 0.72 to 0.99 across all models. The results show that the models tend to perform better for higher-quality queries than lower-quality queries for both screen and component localization. Our replication package contains the # of queries per quality ratings [12].
Retrieval Difficulty. Figures 5 and 6 show the results (based on MRR) for easy- and hard-to-retrieve retrieval tasks, for screen and

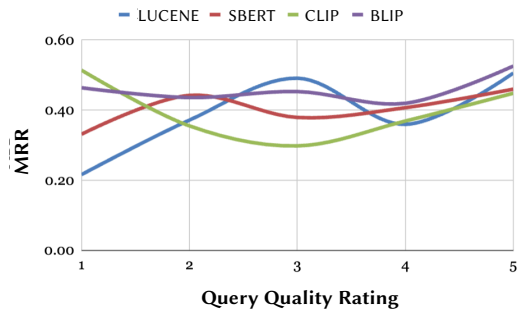

Figure 3: SL results for different query quality levels

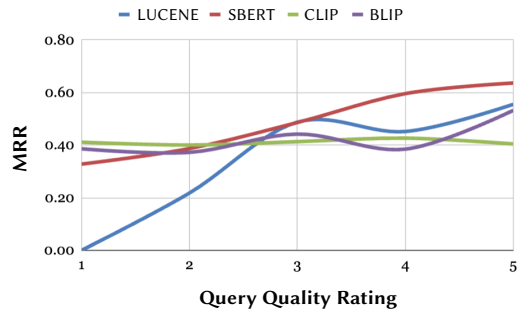

Figure 4: CL results for different query quality levels

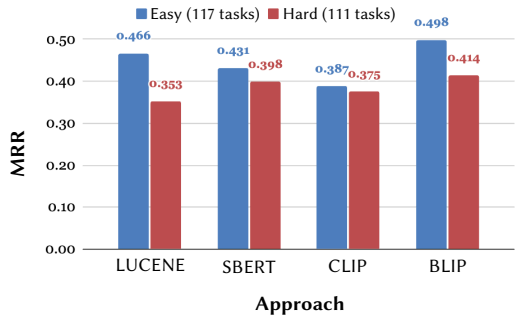

Figure 5: SL results for easy- and hard-to-retrieve tasks

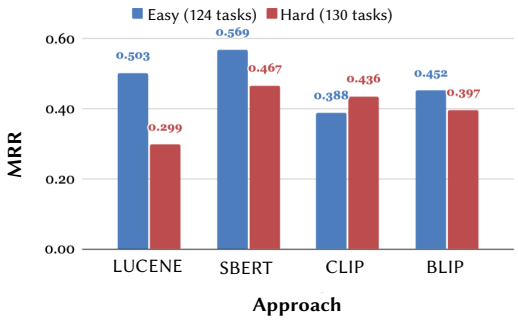

Figure 6: CL results for easy- and hard-to-retrieve tasks

component localization respectively. For SL, all models perform higher on easy-to-retrieve tasks. The same results are found for component localization, except for Clip. The biggest performance gap is observed for Lucene (31% for SL and 68.6% for CL). Of 18 failed screen localization cases for Lucene, 3 tasks are easy and 15 tasks are hard to retrieve, and of 66 failed component localization cases, 17 tasks are easy and 49 tasks are hard to retrieve. Regardless of the difficulty of the tasks, Blip performs highest for screen localization, and SBert performs highest for component localization. The results suggest a correlation between the difficulty of retrieval by humans and the retrieval performance of the models: they tend to perform higher/lower for easier/harder cases.

The results of the different models stem from their distinct architectures, training datasets, and types. Being heavily dependent on word overlap, Lucene may fail to retrieve any UI screen/components resulting in an MRR of 0. As among the failed cases of Lucene, the majority of the queries have a lower quality level and are hard to retrieve, it exhibits the worst performance in Figures 3 and 4 and the largest gaps between easy- and hard-to-retrieve cases in Figures 5 and 6. For the other three models, it is always guaranteed that no matter whether there is textual/visual similarity or not, the models will retrieve the desired UI screen/UI component in some position.

3.6 Discussion

Screen Localization *vs.* Component Localization. We found performance differences between screen localization (SL) (0.381 - 0.457 MRR) and component localization (CL) (0.398 - 0.517 MRR). Several factors make SL more challenging than CL. First, the corpus size is larger for SL than for CL (25.97 screens per app vs. 17.11 components per screen on avg.). Second, SL is more abstract or general than CL as the scope of SL is broader (all screens of the application vs. all components of a screen). Additionally, OBs are generally written focusing on the component level as the user interacts with the component while reproducing the bug. Third, the quality of the OBs has an important impact on the results. For instance, *"The color is unset."* is an OB with a quality rating of 2. The best SL model, Blip, identified the relevant screen for this OB in the 21st place. However, the best CL model, SBert, identified the relevant component in the 1st place.

Textual *vs.* Multi-modal Models. For screen localization, Blip performs best on all metrics, while, for component localization, SBert is the best performing. This distinction stems from the fundamental differences in the tasks: the retrieval corpus for component localization consists of components from the buggy screen, requiring a model that can distinguish subtle differences among components without necessarily understanding the entire screen. Additionally, when reporting bugs related to specific components, users often use the text displayed in UI components to precisely describe the issues. Therefore, text-based models like SBert are effective at capturing the semantic meanings of these textual descriptions, making them more suitable for component localization.

However, for screen localization, multimodal models like Blip which integrates both textual and visual data, excel in understanding the full context of a screen. They offer detailed insight into spatial and functional interactions of various UI elements, which is crucial for localizing buggy UI screens. Moreover, multimodal models excel not only in capturing the overall screen content but also in demonstrating strong grounding capabilities by locating specific UI components within the screen based on textual bug descriptions. This capability is particularly crucial for screen localization, as it allows the model to identify and differentiate screens

by accurately pinpointing the relevant components when bugs are related to specific UI components.

While we found that both textual and textual-visual models achieve a reasonable performance for Buggy UI Localization, no single type of model seems to stand out. Blip and SBert were the best-performing approaches, yet no single model was the best for both tasks. The results indicate that both types of information, textual and visual, can be leveraged for Buggy UI Localization, yet textual data seems to be more useful for component localization, while visual data seems to be useful for screen localization.

Design Requirements for Buggy UI Localization Methods. The results suggest that both textual and visual information alone are helpful for Buggy UI Localization. However, there is still room for improving the localization performance and specialized models may be required for this. We believe that both visual and textual information of the UI screens should be blended to build a more sophisticated model to increase localization performance. Other sources that can be explored are UI hierarchy information, which has shown promising results for command/instruction UI grounding [55, 57]. Moreover, for a successful localization approach, we may require potentially distinct models for screen and component localization.

Finally, while our study showed that it is feasible to leverage the pre-trained models for Buggy UI Localization, fine-tuning may be required to increase the performance of these models. However, creating or obtaining a comprehensive dataset for model fine-tuning is challenging because it should include OB descriptions of different types of bugs and wordings found in real bug reports, with corresponding ground truth data. At the same time, such a dataset should include a variety of mobile apps and should be sufficiently large for the models to effectively learn patterns from the data. Creating a global model that applies to any mobile app and bug description is challenging. Future work should explore the possibility of comparing global *vs.* local models that work for specific apps, which brings an additional challenge: collecting sufficiently large ground truth data for individual apps.

UI Metadata Quality Assessment. To examine the impact of potential noisy UI metadata on the results, we assessed the quality of the three UI metadata attributes used by the textual models (Lucene and SBert): component ID, label, and type. We focused on two factors: attribute value presence and informativeness level.

For value presence, 95.2% of all the 36,572 UI components in our dataset have at least one attribute value. For 92.9% of the components, there is a value for the component ID and/or label, which are potentially more informative than the component type. These results mean that, in 95.2% of the cases, Lucene and SBert leverage at least one piece of information from the UI components for retrieval.

For the informativeness level, we qualitatively analyzed the attributes of 380 UI components (a statistically significant sample with 95% confidence level and 5% error margin). One author assessed and assigned a category (informative or non-informative) to the value of the three attributes. Another author reviewed the first author's categorization, agreeing in 91.4% of the cases. We found that 95.8% of components have a least one informative attribute (among the three attributes), which means that Lucene and SBert leverage at least one informative value from the UI components for retrieval. Aggregating across the three attributes, we assessed the informativeness of 927 attribute values and found that 92% of them are informative: 91.2% of the component IDs, 89.8% of the labels, and 94.7% of the component types are informative.

We conclude that the UI metadata of our dataset is of high quality, thus giving high confidence in the study results and conclusions.

4 Improving Buggy Code Localization

To illustrate the usefulness of automated Buggy UI Localization, we conducted an additional study that investigated how the identified buggy UI screens by Blip, our best-performing screen localization model, can improve traditional buggy code localization approaches. We aim to answer the following research question:

RQ$_4$: *Can the identified buggy UI screens by Blip lead to improved buggy code localization?*

To answer this RQ, we adapted Mahmud *et al.*'s approach [58] (section 4.1) as an end-to-end automated buggy code localization technique (section 4.2), which retrieves potentially buggy files based on UI information from Blip's suggested buggy UI screens. We defined different pipelines that combine Buggy UI Localization and buggy code localization (section 4.2) and compared their performance with baseline techniques that do not use UI data (section 4.3).

4.1 UI-Based Buggy Code Localization

Mahmud *et al.* [58] showed that mobile app UI interaction data improves the performance of four IR-based buggy code localizers that rely on bug reports (*e.g.*, BugLocator [86]). Their approach modifies the initial ranking of potentially buggy code files produced by a buggy code localizer for a given bug report, by **boosting** relevant files and/or **filtering** out irrelevant files, or by performing **query reformulation**. These operations (*a.k.a.* augmentations) use information extracted from the UI screen that shows the reported bug and the preceding 1-3 screens in a bug reproduction trace.

The information extracted from UI screens is *UI terms* (*e.g.*, activity and window names) which are matched against code file names to produce a set of *UI-related files*. The UI terms and UI-related files are used by two augmentation methods: (1) **Reformulating queries** via query expansion, which appends UI terms to bug reports, or via query replacement, which uses UI terms as the query; and (2) **File re-ranking** by filtering, boosting, or combining filtering and boosting. Filtering removes files that do not match UI-related files (*e.g.*, classes that directly interact with the UI) from the file corpus. Boosting elevates the ranking of files in the corpus that match UI-related files during the search.

Mahmud *et al.* employed four main configuration parameters to integrate UI information into the IR bug localizers: (1) the number of UI screens in a reproduction trace (*i.e.*, the buggy screen and the preceding 1-3 screens); (2) five types of UI information sources (*e.g.*, UI screen, UI components, and exercised UI components); (3) query reformulation strategies; and (4) re-ranking strategies. In total, 657 configurations were defined and evaluated for each bug localizer.

4.2 Integrating Buggy UI and Code Localization

Mahmud *et al.*'s approach [58] requires as input a trace of the UI screens and UI components that the user interacted with to reproduce a reported bug. The trace and the buggy UI screen in the trace are meant to be manually collected/identified by the developer. Mahmud *et al.*'s approach then uses the metadata information

from the buggy screen and the 1-3 prior screens/components in the reproduction scenario as input to their augmentation approaches.

Our goal is to eliminate the manual effort of Mahmud *et al.*'s approach and define an effective and fully automated end-to-end pipeline of buggy code localization using the buggy UI screens recommended by a UI screen localizer. To that end, we adapted their approach by using our best screen localization approach (*i.e.*, Blip) to automatically suggest the top 3-4 buggy UI screens as the only information needed by the buggy code localization pipeline.

As such, we defined an approach that integrates both the screen localization and buggy code localization pipelines since the ultimate goal is to produce a ranked list of potentially buggy code files for a given bug report. The challenge in defining this combined approach is that a bug report can contain multiple OB descriptions. If we execute Blip on each OB description, it would produce multiple lists of potentially buggy UI screens. Therefore, this challenge is rooted in deciding which buggy screens should be given as input to the localization pipeline, to produce a single ranking of buggy files.

To address this problem, we considered two options: (1) produce and provide a single ranking of UI screens for the bug report, or (2) provide each ranking of UI screens (for each OB description in the bug report) to the buggy code localization pipeline, to produce multiple code file rankings, and then combine these rankings into a final code file ranking. For option #1, we explored two strategies, namely: (i) **Concat OBs**, which concatenates the OB descriptions in a bug report and uses the resulting query as input to Blip, and (ii) **First OB**, which selects only the first OB description found in the bug report as a query to Blip. When executed, these two strategies produce a single UI screen ranking, which can be used by the buggy code localization pipeline to suggest a single code file ranking for the bug report. As for option #2, to produce a single code file ranking, we defined a strategy called **Individual OBs**, which first averages the similarity scores of each code file found in all the buggy file rankings to produce a single similarity score for the file. Then, these combined similarities, for all the files in the rankings, are used to produce a final code file ranking (*i.e.*, sorting by these similarities).

4.3 Approach Execution, Dataset, and Metrics

We selected the two best IR-based buggy code localization techniques from Mahmud *et al.*'s study [58], namely Lucene [42] (adapted for buggy code localization) and BugLocator [86], and executed them in our combined pipeline for buggy code localization. We tested all 70 feasible configurations of different augmentation methods and UI information kinds defined in the prior work. We also experimented with providing the top 3 and 4 buggy UI screens suggested by Blip, based on the best number of screens found in the prior study for Lucene (4 screens) and BugLocator (3 screens).

We executed the three combined pipelines defined above, namely **Concat OBs**, **First OB**, and **Individual OBs**, using both buggy code localizers. However, we could not execute Individual OBs with BugLocator because the tool provided by the original authors [86] does not provide the code file rankings needed by Individual OBs. The pipelines were executed on 79 of the 80 bug reports from the buggy code localization benchmark provided by Mahmud *et al.* [58]. As mentioned in section 3.2.1, we excluded one bug report because we could not reproduce the bug. The full set of 87 bug reports was not used in the Buggy UI Localization due to the absence of Java files as ground truth for buggy code localization.

Table 4: Buggy Code Loc. Performance via Buggy UI Loc.

Bug Localizer	Approach	# Scrns	H@5	H@10	RI of H@10	#Bug Top10
Lucene	Baseline	4	0.74	0.79	-	61
	Concat OBs	4	0.75	0.88	11.49%	68
	First OB	4	0.74	0.88	11.49%	68
	Individual OBs	4	0.77	0.87	9.85%	67
BugLocator	Baseline	3	0.59	0.71	-	56
	Concat OBs	3	0.72	0.79	10.72%	62
	First OB	3	0.61	0.80	12.41%	63

The performance of the combined pipelines, using all possible configurations and IR bug localizers, was measured and compared using Hits@k and its relative improvement (RI), in line with the methodology followed by the prior work [58]. We used as baselines the original IR bug localizers, without using any UI information. Note that 4 screen experiments used 77 bug reports as 2 bug reports have only 3 screens in the screen localization (SL) corpus.

4.4 Results

Table 4 shows the buggy code localization results for both IR bug localizers and the best configurations we obtained among all configurations. These results are obtained when Blip suggests the top 3 and 4 buggy screens. Complete results of all configurations and experiments are provided in our replication package [12].

For each pipeline, IR bug localizer, and number of buggy screens recommended by Blip, we consistently found that the best configuration (*i.e.*, the highest H@10 improvement compared to the baselines) includes filtering with UI Components (SC) and boosting with UI Screens (GS). Additionally, the best configuration includes query expansion with GS when Concat OBs is used with Lucene and query expansion with SC when First OB is used with BugLocator. As in the prior work [58], we obtained the best results with 4 screens for Lucene and 3 screens for BugLocator.

Table 4 reveals that all the combined pipelines for buggy code localization lead to a performance improvement compared to the baselines, by 9.85% to 12.41% H@10. When using Lucene, the Concat OBs and First OB pipelines achieve the best improvement: 11.49% for H@10, which translates into retrieving the buggy code files in the top-10 results for 7 more bug reports, compared to the baseline. When using BugLocator, First OB pipeline results in improving the baseline by 12.41% H@10 (*i.e.*, 7 more successful retrieval tasks).

We compare our results from table 4 with the results achieved by the best configurations obtained for Lucene and BugLocator by Mahmud *et al.* [58], since those results represent a perfect identification of the buggy UI screen, along with the reproduction scenario. However, we must cautiously compare these results since the bug reports used in both studies are not exactly the same. Not surprisingly, the performance of the manual buggy code localization approach by Mahmud *et al.* [58] is slightly higher than the performance of our best configurations (0.9 vs 0.88 H@10 for Lucene, and 0.84 vs 0.80 H@10 for BugLocator). This difference is acceptable, considering that we propose a fully automated way of localizing buggy code files via Buggy UI Localization, which still outperforms baseline localizers, while the prior work requires manual effort in collecting reproduction traces and the buggy screen.

Given the results, we conclude that Buggy UI Localization can be useful to improve the performance of UI-based buggy code localization in a fully automated end-to-end way.

5 Threats to Validity

Construct Validity. There may be subjectivity introduced in the dataset construction when identifying the OB descriptions in the bug reports, their quality rating, retrieval difficulty levels, and the ground truth buggy UI screens and components. We mitigated this threat by adopting a rigorous methodology to label and curate the data during joint sessions of bug understanding, replication, and analysis among four authors, reaching consensus in all cases.

Internal Validity. The selection of models affects the internal validity of our results/conclusions. To mitigate this, we covered both uni-modal (SBert) and multi-modal (Clip & Blip) DL models, and a unsupervised textual technique (Lucene) as baseline for Buggy UI Localization. For buggy code localization study, we conducted various experiments with all feasible configurations on two localizers (Lucene and BugLocator) to obtain the best-performing configuration. Another threat concerns the methodology we used to collect UI corpus data, which may have led to incomplete coverage of UI screens for an app. While this can have an impact on the study results, we used the same screen corpus to evaluate all models, thus ensuring a fair evaluation.

External Validity. The conclusions of our study may not generalize to other retrieval models, bug descriptions, and apps. To improve the generalization, we selected different types of models and built a real dataset containing a variety of bug types, and apps that implement different UIs for multiple domains.

6 Related Work

Mobile App Bug Report Management. Recent research [36, 38, 39, 71, 82] has explored the use of mobile app bug reports to automate various bug report management tasks. Researchers [82, 84] have proposed approaches to reproduce Android bugs or generate test cases based on bug reports. Our Buggy UI Localization approach that identifies the buggy screens/components can help these approaches to generate assertions that validate the reported bugs. Song *et al.* [71] proposed a chatbot to help users report Android bugs via visual guidance and quality verification. This chatbot can benefit from a Buggy UI Localization approach by accurately assessing how bug descriptions corresponds to UI screens/components. Despite the growing body of research on automating bug report management tasks such as bug reporting [36, 71], reproduction [19, 20, 38, 82–84, 84], localization [14, 25–27, 40, 46, 76], and others [32, 39, 78, 85], prior work has not explored how to automatically localize buggy UIs as we do.

Static Analysis of Mobile App UIs. Researchers have proposed techniques/tools to statically analysis mobile app UIs (*e.g.*, FrontMatter [47, 48], Gator [79], Backstage [16], GoalExplorer [49], and others [41, 54]). One main advantage of these approaches, over dynamic analysis, is their ability to cover a large number of UI screens for an application. Our future work will explore the use of these techniques to assess how UI screen coverage can impact the performance of the studied models. Static analyzers also provide features that can assist buggy code localization. For example, FrontMatter [47, 48] identifies which Android APIs can be triggered by an interaction with a UI component, which may help identify buggy code elements. While the main problem we address in this paper is buggy UI localization, our future work will explore how static analyzers and UI localizers can be integrated to better localize buggy code based on a bug description. The main limitations of static analyzers, which prevented us from using them for data collection, include: (1) they may fail to capture server-side content loaded only at runtime, resulting in potentially unrealistic UI screens (2) they do not provide UI screenshots, needed to evaluate the studied multi-modal models, and (3) potential imprecision of app behavior captured by these tools as they may not provide UI screens displaying certain bugs (*e.g.*, incorrect output and navigation issues).

UI Representation Learning and Applications. UI representation learning aims to represent UI elements or text via embeddings [17, 43, 53, 57] for downstream tasks such as image captioning [30, 61, 74] and UI component labeling [29, 30, 56]. One application of UI representation learning is mapping (*a.k.a.* grounding) textual instructions to UI action/elements [55, 64, 77]. Pasupat *et al.* [64] evaluated three models to ground natural language commands to web elements. Li *et al.* [55] utilized transformers models for this task, based on three synthetic training datasets. There are key differences that make it hard to adapt those models to our problem. For example, Li *et al.*'s approach [55] requires a sequence of screens where the instructions are performed, and then locating the corresponding UI component for each instruction. In contrast, our work identifies the buggy UI screens and components without any prior information about relevant screens. Furthermore, our study deals with bug descriptions, whose language is more complex than that of UI instructions [28].

7 Conclusions

This paper reports the results of the first empirical study that investigates the effectiveness of textual/visual neural models for automatically localizing buggy UI screens and UI components based on the bug descriptions of mobile apps. We evaluated approaches for screen and component localization, using a real-world dataset of manually-curated bug descriptions and ground truth UI data.

The study reveals that the best-performing approaches can localize correct buggy UI screens and components in the top-3 recommendations for 52% and 60% of the bug descriptions. We found that the models tend to perform better for the bug descriptions which are easier to retrieve even for humans and which have a higher quality. We also illustrate that Buggy UI Localization can be useful to automate and improve traditional buggy code localizers.

8 Data Availability

For verifiability and reproducibility, we have made all the study artifacts (*e.g.*, dataset, source code, and documentation) publicly available including the latest version [12] and the first version [11].

Acknowledgments

This work is supported by the U.S. NSF CCF grants 1955853, 2343057, and 2007246. Any opinions, findings, and conclusions expressed herein are of the authors and do not necessarily reflect the sponsors'.

References

[1] 2017. GnuCash's bug report #723. http://tinyurl.com/4zhjdcua.
[2] 2021. BugZilla Issue Tracker - https://bugzilla.mozilla.org.
[3] 2021. Cohen's kappa. https://en.wikipedia.org/wiki/Cohen%27s_kappa.
[4] 2021. GnuCash. https://play.google.com/store/apps/details?id=org.gnucash.android&hl=en_US&gl=US.
[5] 2021. Images2PDF. https://play.google.com/store/apps/details?id=imagetopdf.pdfconverter.jpgtopdf.pdfeditor&hl=en_US&gl=US.
[6] 2021. K-9. https://play.google.com/store/apps/details?id=com.fsck.k9&hl=en_US&gl=US.
[7] 2023. Android Layouts and UI Components/Widgets. https://developer.android.com/develop/ui/views/layout/declaring-layout.
[8] 2023. Rico dataset. https://interactionmining.org/rico.
[9] 2023. WiFi Analyzer. https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer.
[10] 2023. WiFi Analyzer's issue #191. https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer/issues/191.
[11] 2024. First version of the replication package. https://doi.org/10.5281/zenodo.12669081.
[12] 2024. Latest version of the replication package. https://doi.org/10.5281/zenodo.12669080.
[13] Vishakha Agrawal, Yong-Han Lin, and Jinghui Cheng. 2022. Understanding the characteristics of visual contents in open source issue discussions: a case study of jupyter notebook. In *EASE'22*. 249–254.
[14] Shayan A Akbar and Avinash C Kak. 2020. A large-scale comparative evaluation of IR-based tools for bug localization. In *MSR'20*. 21–31.
[15] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *ICSE'06*.
[16] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. 2017. Detecting behavior anomalies in graphical user interfaces. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 201–203.
[17] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, et al. 2021. Uibert: Learning generic multimodal representations for ui understanding. *arXiv preprint arXiv:2107.13731* (2021).
[18] Alberto Baldrati, Marco Bertini, Tiberio Uricchio, and Alberto Del Bimbo. 2022. Conditioned and composed image retrieval combining and partially fine-tuning CLIP-based features. In *CVPR'22*.
[19] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. 2022. Translating Video Recordings of Complex Mobile App UI Gestures Into Replayable Scenarios. *IEEE Transactions on Software Engineering* (2022), to appear.
[20] 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 (ICSE'20)*. 309–321.
[21] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *FSE'08*.
[22] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon. 2013. Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. In *ISSRE'13*.
[23] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *CSCW'10*. 301–310.
[24] 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 *FSE'19*. 86–96.
[25] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2017. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In *ICSME'17*.
[26] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2019. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *ESE'19* 24 (2019), 2947–3007.
[27] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. 2019. Reformulating queries for duplicate bug report detection. In *SANER'19*. IEEE, 218–229.
[28] 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 *FSE'17*.
[29] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *ICSE'20*.
[30] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Towards Complete Icon Labeling in Mobile Applications. In *CHI'22*.
[31] Marcos V Conde and Kerem Turgutlu. 2021. CLIP-Art: Contrastive pre-training for fine-grained art classification. In *CVPR'21*.
[32] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. In *ICSE'21*.
[33] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[34] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *MSR'14*. 62–71.
[35] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. 2018. Chaff from the wheat: Characterizing and determining valid bug reports. *TSE'18* (2018).
[36] Mattia Fazzini, Kevin Patrick Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2022. Enhancing Mobile App Bug Reporting via Real-time Understanding of Reproduction Steps. *TSE22* (2022).
[37] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *ISSTA'18*. 141–152.
[38] Sidong Feng and Chunyang Chen. 2022. GIFdroid: an automated light-weight tool for replaying visual bug reports. In *ICSE'22*.
[39] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. 2023. Read It, Don't Watch It: Captioning Bug Recordings Automatically. *arXiv preprint arXiv:2302.00886* (2023).
[40] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. 2021. Combining query reduction and expansion for text-retrieval-based bug localization. In *SANER'21*. IEEE, 166–176.
[41] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. 2020. Improving automated GUI exploration of android apps via static dependency analysis. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 557–568.
[42] Erik Hatcher and Otis Gospodnetic. 2004. *Lucene in Action*. Manning Publications.
[43] Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichers, Gabriel Schubiner, Ruby Lee, and Jindong Chen. 2021. Actionbert: Leveraging user actions for semantic understanding of user interfaces. In *AAAI'21*.
[44] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. [n. d.]. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *SANER'22*.
[45] 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, 321–322.
[46] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential Biases in Bug Localization: Do They Matter?. In *ASE'14*. 803–814.
[47] Konstantin Kuznetsov, Chen Fu, Song Gao, David N Jansen, Lijun Zhang, and Andreas Zeller. 2021. Frontmatter: mining android user interfaces at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1580–1584.
[48] Konstantin Kuznetsov, Chen Fu, Song Gao, David N Jansen, Lijun Zhang, and Andreas Zeller. 2021. What do all these buttons do? statically mining android user interfaces at scale. *arXiv preprint arXiv:2105.03144* (2021).
[49] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–127.
[50] Eero I. Laukkanen and Mika V. Mäntylä. 2011. Survey Reproduction of Defect Reporting in Industrial Software Development. In *ESEM'11*.
[51] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *ISSTA'18*. 61–72.
[52] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. 2022. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *ICML'22*.
[53] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2vec: Semantic embedding of gui screens and gui components. In *CHI'21*.
[54] Yuying Li, Yang Feng, Chao Guo, Zhenyu Chen, and Baowen Xu. 2023. Crowdsourced test case generation for android applications via static program analysis. *Automated Software Engineering* 30, 2 (2023), 26.
[55] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).
[56] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020. Widget captioning: Generating natural language description for mobile user interface elements. *arXiv preprint arXiv:2010.04295* (2020).
[57] Yang Li, Gang Li, Xin Zhou, Mostafa Dehghani, and Alexey Gritsenko. 2021. Vut: Versatile ui transformer for multi-modal multi-task user interface modeling. *arXiv preprint arXiv:2112.05692* (2021).
[58] Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, SM Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. 2024. On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization. *ICSE'24* (2024).

[59] Kevin Moran, Mario Linares-Váquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *ICST'16*.
[60] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Cristopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: A Practical Tool for Automated Testing of Android Applications. In *ICSE'17*.
[61] Kevin Moran, Ali Yachnes, George Purnell, Junayed Mahmud, Michele Tufano, Carlos Bernal Cardenas, Denys Poshyvanyk, and Zach H'Doubler. 2022. An Empirical Investigation into the Use of Image Captioning for Automated Software Documentation. In *SANER'22*.
[62] Maleknaz Nayebi. 2020. Eye of the mind: Image processing for social coding. In *ICSE'20*. 49–52.
[63] OpenAI. 2023. ChatGPT. https://www.openai.com. Generative Pre-trained Transformer for conversational AI.
[64] Panupong Pasupat, Tian-Shun Jiang, Evan Zheran Liu, Kelvin Guu, and Percy Liang. 2018. Mapping natural language commands to web elements. *arXiv preprint arXiv:1808.09132* (2018).
[65] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *ICML'21*.
[66] Hanoona Rasheed, Muhammad Uzair Khattak, Muhammad Maaz, Salman Khan, and Fahad Shahbaz Khan. 2022. Fine-tuned CLIP Models are Efficient Video Learners. *arXiv preprint arXiv:2212.03640* (2022).
[67] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).
[68] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. 2020. The significance of bug report elements. *ESE'20* (2020).
[69] Yang Song and Oscar Chaparro. 2020. BEE: a tool for structuring and analyzing bug reports. In *FSE'20*.
[70] Yang Song, Junayed Mahmud, Nadeeshan De Silva, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2023. BURT: A Chatbot for Interactive Bug Reporting. In *ICSE'23*.
[71] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (Android app) end-users. In *FSE'22*.
[72] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. 2015. Automated prediction of bug report priority using multi-factor analysis. *ESE'15* (2015).
[73] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. https://ai.meta.com/blog/large-language-model-llama-meta-ai. Technical report, Meta AI.
[74] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2words: Automatic mobile UI summarization with multimodal learning. In *UIST'21*.
[75] Tyler Wendland, Jingyang Sun, Junayed Mahmud, SM Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A dataset of manually-reproduced bug reports for android apps. In *MSR'21*.
[76] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *ICSME'14*. 181–190.
[77] Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica S Lam. 2021. Grounding open-domain instructions to automate web support tasks. *arXiv preprint arXiv:2103.16057* (2021).
[78] Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2024. Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports. In *ICSE'24*. 1–13.
[79] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25 (2018), 833–873.
[80] Nor Shahida Mohamad Yusop, John Grundy, and Rajesh Vasa. 2016. Reporting usability defects: do reporters report what software developers need?. In *EASE'16*. 1–10.
[81] Renrui Zhang, Ziyu Guo, Wei Zhang, Kunchang Li, Xupeng Miao, Bin Cui, Yu Qiao, Peng Gao, and Hongsheng Li. 2022. Pointclip: Point cloud understanding by clip. In *CVPR'22*.
[82] Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William GJ Halfond. 2023. Automatically Reproducing Android Bug Reports Using Natural Language Processing and Reinforcement Learning. *arXiv preprint arXiv:2301.07775* (2023).
[83] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *TOSEM'22* 31, 3 (2022), 1–33.
[84] 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 *ICSE'19*. 128–139.
[85] Jian Zhou and Hongyu Zhang. 2012. Learning to rank duplicate bug reports. In *CIKM'12*. 852–861.
[86] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 14–24.
[87] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *TSE'10* (2010).
[88] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2018. How practitioners perceive automated bug report management techniques. *TSE'18* (2018).

Received 2024-04-12; accepted 2024-07-03