Generating Failure-Based Oracles to Support Testing of Reported Bugs in Android Apps

Jack Johnson*, Junayed Mahmud†, Oscar Chaparro‡, Kevin Moran†, Mattia Fazzini*
*University of Minnesota, MN, USA; †University of Central Florida, FL, USA; ‡William & Mary, VA, USA
joh19267@umn.edu, junayed.mahmud@ucf.edu, oscarch@wm.edu, kpmoran@ucf.edu, mfazzini@umn.edu

***Abstract*—In the context of mobile apps, bug report management tasks have been shown to be among the most time-consuming and intellectually intensive software maintenance activities. As such, researchers have developed tools to automate the reproduction, validation, and localization of reported bugs. However, one complex, time-consuming, and important task that lacks automated support is the creation of test oracles for reported functional failures that manifest through the GUI. This is challenging task – requiring nuanced, multi-modal reasoning about bug descriptions, affected GUI components, and the characteristics of the related erroneous program state(s).**

To explore the feasibility of automating this task, we conduct a empirical investigation into how the multi-modal (*i.e.*, text and GUI-related code) reasoning capabilities of Large Language Models (LLMs) can be used to automatically generate assertion-based test oracles for non-crashing, functional failures described in Android app bug reports. Building upon the findings of this study, we construct and evaluate ANDROB2O, an automated, LLM-based approach that, given a bug report and the GUI screen associated with the reported failure as inputs, generates failure-based oracles (FBOs) in the form of test assertions. The approach first identifies the GUI elements related to the failure and then defines assertions that aim to confirm the *absence* of the failure based on the elements' properties. To evaluate ANDROB2O, we create the first dataset of Android bug reports containing test cases with GUI interactions and test oracles that reveal reported failures. The results of our evaluation on 152 failures show that ANDROB2O is able to generate FBOs that successfully identify the failure (and hence can confirm its absence) in 61.2% of the cases. We integrated ANDROB2O with REBL, a failure reproduction tool, to evaluate its effectiveness in automated generation of test cases complete with oracles for reported failures, and obtained promising results.

I. INTRODUCTION

Due to the inherent limitations of software verification and validation techniques, it is impossible to reveal and eliminate every fault before a system's release. Consequently, users inevitably experience software failures. In mobile applications (or apps in short), most failures manifest as GUI-based functional issues that appear visually on the app screen without an explicit oracle such as a crash [1, 2]. Since these failures lack automated detection and reporting mechanisms, users typically report them manually via bug reports [1–6].

Managing these bug reports – including reproducing, confirming, localizing, fixing, and preventing the reoccurrence of the bugs – has been illustrated to be a challenging and time-consuming set of tasks. To help alleviate the burden of bug report management in the context of mobile apps, researchers have proposed novel bug reporting systems [1, 3, 7, 8], fault localization techniques [9–13], and automated techniques that translate the reproduction steps contained in bug reports into executable GUI interactions, which can be used to validate the presence of a bug, and build reproduction scripts [14–24]. While there has been great attention given to developing techniques that support reporting and reproducing bugs, less research has been conducted on the downstream tasks of fixing and creating regression tests for reported bugs.

More specifically, one critical task related to testing and bug fixing that no prior technique is capable of carrying out is *generating oracles for reported failures*. While some recent automated GUI testing techniques can generate oracles via differential state analysis [25], they *cannot* create oracles from bug reports. The lack of automated oracle generation for reported bugs is a key gap in the current literature, as it plays a important role in automated generation of GUI-level regression tests [26], and automated program repair techniques that often rely on regression tests as a feedback signal [27].

However, while the task of oracle generation is important, it also carries with it notable challenges. First, it requires understanding the bug description from the report, which can be written using diverse discourse and lexicon [28]. Second, it requires identifying the GUI elements affected by the failure (*e.g.*, buttons or text fields), and verifying if the properties of such elements exhibit an erroneous state (*e.g.*, incorrect textual labels or element dimensions). This requires nuanced reasoning that reconciles bug descriptions and a wide variety of mobile app bug manifestation patterns through the GUI [2].

In this work, we are the first to investigate the feasibility of automatically generating GUI-based test oracles for non-crashing, functional failures described in bug reports. We refer to the generated oracles as *failure-based oracles* (FBOs), as the oracles are centered around the notion of understanding the failure characteristics of a reported bug as it manifests through the GUI. The key intuition of our technique is that by first inferring the characteristics of GUI elements that exhibit a failure, a reliable oracle can be created that is able to detect the *absence* of such characteristics or properties, and hence the absence of the related bug.

Given the complex reasoning required for the creation of such oracles, we investigate the capabilities of a Large Language Model (LLM) in generating FBOs for Android app failures. We focus on an LLM-based approach as prior work highlighted the diverse manifestations of mobile app bugs [29], suggesting that a generalized learning approach is

likely necessary for practical FBO generation. Additionally, LLMs are able to understand and generate both natural language and (GUI-related) code, and have been shown to be effective in multiple software engineering tasks, such as code summarization, testing, and repair [4, 30–32].

We first conduct a preliminary study using 17 bug reports from 17 Android apps to evaluate the ability of widely used LLM to generate FBOs. We systematically analyze its effectiveness under various prompt configurations, measuring its success in generating assertions that detect whether erroneous element properties are present. Informed by the results of this study, we design ANDROB2O, the first fully automated approach for generating FBOs from Android bug reports. Given a bug report and an XML representation of a failing GUI screen (provided manually by a developer or automatically by a reproduction tool), ANDROB2O identifies GUI elements related to the failure and generates assertions to confirm the absence of the failure. FBOs are implemented using the `UIAutomator` API [33] and can be integrated with test interactions produced by existing approaches [14–24] to form robust/complete, automated test cases.

To evaluate ANDROB2O, we create a novel benchmark called FBO4A (Failure-Based Oracles for Android) that contains 152 bug reports from 24 Android apps, and allows for the assessment of effectiveness in creating oracles. When the failing GUI screen is manually provided, ANDROB2O successfully generates FBOs for 61.2% of the reports, which is twice the success rate of a baseline that directly asks an LLM to generate FBOs from textual bug descriptions and screen metadata. Additionally, ANDROB2O achieves a comparable success rate (62.5%) on 16 additional bug reports likely outside of the LLM's training data, providing evidence of the approach's generalizability.

To further evaluate ANDROB2O's performance when the failing screen is automatically provided, we integrate our approach with REBL [24], a state-of-the-art Android failure reproduction tool. REBL generates GUI interactions from a bug report's reproduction steps and provides a mechanism to validate proper reproduction. However, while the tool is capable of reproducing complex bugs, *it is not able to produce a test oracle for the reported failure*. In combination with REBL, ANDROB2O successfully generates FBOs for 19 of 32 failures (59.4%) reproduced by REBL. When REBL is supplied with additional app configuration steps, ANDROB2O generates FBOs for 75 of 136 failures (55.1%). These results show that ANDROB2O effectively generates FBOs to reveal reported failures across both manually and automatically identified failing screens. Finally, to provide further evidence of the effectiveness of the approach under different settings, we also evaluated ANDROB2O with three additional LLMs on a stratified sample of 35 bug reports from FBO4A and observed consistent performance across the three models.

Based on these results, we believe that our work represents a key step toward automating test oracle creation for non-crashing mobile app failures, providing a foundation for more effective and efficient bug resolution. By automatically generating FBOs, our approach has the potential to reduce the manual effort required for supporting mobile app debugging, regression testing, and repair at scale.

In summary, this paper makes the following contributions:

- A systematic preliminary study that investigates an LLM's ability to generate failure-based oracles from bug reports describing issues appearing in Android apps.
- ANDROB2O, the first automated approach capable of generating failure-based oracles by identifying the GUI elements related to the failures and generating assertions that focus on the relevant element's properties.
- FBO4A, the first dataset and benchmark for the creation of FBOs in Android apps, including test cases and assertions spanning 152 bug reports from 24 Android apps.
- A comprehensive evaluation of ANDROB2O, including its integration with a reproduction tool, showing its effectiveness across diverse usage scenarios, failures, and apps.
- A publicly available artifact [34] that contains ANDROB2O's source code, the FBO4A benchamrk, and results to support validation of our work and future research.

II. BACKGROUND & TERMINOLOGY

We focus on the most prevalent type of mobile app *bug reports*, those describing *non-crashing functional failures*: issues that affect an app's functionality without causing crashes and manifesting through the GUI [2]. A bug report typically consists of four key parts. The *title* is a brief summary of the failure. The *steps to reproduce the failure* (S2Rs) describe the events or actions needed to trigger the failure. The *observed behavior* (OB) describes the failure. The *expected behavior* (EB) describes how the app should behave if the failure did not occur. We focus on bug reports where these components are documented in text, which is the most common modality for reporting mobile app failures [2].

In this paper, we investigate the feasibility of automatically creating assertion-based test oracles that validate whether the failure described in a bug report manifests on the GUI screen resulting from executing the S2Rs. We refer to such oracles as *failure-based oracles* (FBOs). By definition, FBOs *fail* on the buggy app (*i.e.*, the assertions indicate the occurrence of the failure) and *pass* on its fixed version. An FBO is composed of one or more assertions. Each assertion is a conditional expression that checks the properties of a subset of *elements* in a GUI screen. For example, an FBO could assert that an erroneous text label (relevant to the failure described in a bug report) does not appear on screen. Because assertions operate on GUI elements, an FBO needs to contain the code necessary to retrieve the elements from the screen. Given this, an FBO contains two parts: (i) the element(s) associated with the assertion(s) and (ii) the assertion(s).

III. DATASET

To assess the capabilities of an LLM to generate mobile app FBOs, we need a dataset of bug reports and associated data, including: (i) failing screens, (ii) XML hierarchies [35], and (iii) ground-truth FBOs. To the best of our knowledge,

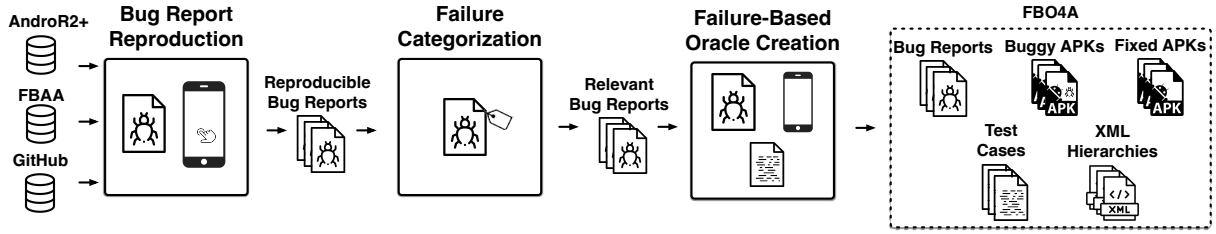

Fig. 1: Overview of the methodology for creating the FBO4A dataset.

no such dataset currently exists. As such, we construct a new benchmark for validating FBO generation. Figure 1 provides an overview of the methodology we used to create the benchmark, which we call FBO4A (Failure-Based Oracles for Android). The dataset is composed of three subsets: FBO4A$_{PS}$, FBO4A$_{AE}$, and FBO4A$_{UD}$. We use FBO4A$_{PS}$ to assess the performance of different LLM prompting configurations in our preliminary study (see Section IV), FBO4A$_{AE}$ to evaluate the effectiveness of our finalized automated approach for generating FBOs (see Section VI), and FBO4A$_{UD}$ to assess the our approach's effectiveness on unseen data (*i.e.*, data that the LLM likely did not see during training). Two authors worked on the creation of FBO4A.

To create FBO4A$_{PS}$ and FBO4A$_{AE}$, we leveraged two existing datasets of reproducible Android app bug reports: AndroR2+[2] (180 reports) and FBAA[36] (399 reports). These are the largest publicly available datasets containing GUI-based crashing and non-crashing failures, constructed following systematic procedures [2, 36].

We applied a systematic filtering process to select relevant bug reports. Starting from AndroR2+ and FBAA, we excluded reports describing crashes (49), duplicates (5), missing APKs [37] (4), as well as those that required complex setup (72). Then, each author independently reproduced all remaining reports on a clean Android emulator, using the app version specified in the source dataset or associated report. The authors then met to and reached full agreement on whether or not each bug was reproduced. 179 bug reports were non-reproducible, and many such cases stemmed from two apps (100 reports) whose server-side changes prevented reproduction. After filtering and confirming bug reproduction, 270 reports remained (86 from AndroR2+ and 184 from FBAA).

The two authors then categorized these 270 reports using the failure taxonomy by introduced by Baral *et al.* [29], which classifies bug reports based failure detection strategies. The two authors independently labeled all 270 bug reports. The Krippendorff's Alpha for the labeling task was 0.938, indicating high reliability [38, 39]. All disagreements were resolved through discussion until consensus was reached [40]. During this labeling process, we filtered out 62 additional reports whose failures could not be detected using the `UIAutomator` API, which is the target API for implementing FBOs in our work. This step resulted in 208 bug reports remaining.

Next, the two authors, each with more than four years of Android programming experience, worked together to write test cases with both interactions that reproduced the failures and FBOs for the failures. The authors worked on one test case at a time. For each test case, the authors started by writing the test case interactions and then executed the test to collect the XML hierarchy and the screenshot of the failure screen (using Android OS utilities [33]). The XML hierarchy is an XML file that contains the hierarchical structure of GUI elements on the screen (buttons, text fields, layouts, *etc.*) and their metadata (text labels, color, dimensions, *etc.*). Then, the authors analyzed the element metadata and identified how to detect the failure via assertions on such metadata through discussion and consensus. (In six cases, the two authors had an initial diverging opinion on how to detect the failure but then reached consensus through discussion.) At this point, the authors wrote the FBOs, consisting of the code to retrieve the elements(s) and the assertion(s) that checked for the presence of the failure. The authors used the `UIAutomator` API to write the FBOs and then added it to the test. Finally, the authors verified that the FBOs failed on the buggy version of the app and passed on the fixed app version by running the test cases, making adjustments if needed. The FBOs generated through this process represent our ground-truth dataset, which includes **169 test cases with FBOs** from 27 apps, and corresponding bug reports, buggy APKs, fixed APKs, and XML hierarchies. We do not report inter-rater agreement on the creation of FBOs as this is not a labeling task.

During the process of writing FBOs, we filtered out an additional 39 reports. In 25 cases, we could not obtain the XML hierarchy of the buggy screen (due to dynamic screen content loaded at runtime and limitations of Android OS' tooling), and in 14 cases, the FBOs could not be derived based on XML hierarchy information (required by `UIAutomator`). We consider that the resulting dataset is representative of Baral *et al.*'s [29] failure categories detectable through the XML hierarchy (which are most prevalent categories). Furthermore, the dataset size is on the same order of magnitude as the source datasets' (AndroR2+ and FBAA).

The authors also categorized the FBOs by analyzing their characteristics to create the dataset for the preliminary study (FBO4A$_{PS}$). The authors used inductive coding [41, 42] for the categorization. This process led to 11 categories, reported in Table I. The table describes the categories and the number of FBOs in each category (*FBO#*). We built **FBO4A$_{PS}$** by selecting **17 reports** (10%) from the 169 available. To create this dataset, we first randomly selected 11 bug reports (making sure each report belonged to a different app) so that each category from Table I was represented in the dataset. We then selected

TABLE I: Failure-based oracle categories and frequency.

ID	*Summary Description*	*FBO#*
C01	Check that the text of an element is not the wrong one.	60
C02	Check that the wrong element is not on screen.	53
C03	Check that the a correct element is on screen.	32
C04	Check that the checked property of an element is not the wrong one.	7
C05	Check that the number of elements on screen is not the wrong one.	6
C06	Check that the position of an element on screen is not the wrong one.	4
C07	Check that the keyboard does not have the wrong property.	3
C08	Check that the position between two elements is not the wrong one.	2
C09	Check that the bounds of an element are not the wrong ones.	2
C10	Check that the enabled property of an element is not the wrong one.	1
C11	Check that the focused property of an element is not the wrong one.	1

the remaining six bug reports to best mirror the distribution of the bug reports across the derived FBO categories in Table I. Excluding these 17 reports, this left us with **152 bug reports** (across 24 apps) that compose the **FBO4A**$_{AE}$ dataset, used for the final ANDROB2O approach evaluation.

To create FBO4A$_{UD}$, we first identified bug reports for GitHub-hosted apps using the methodology defined by ANDROR2+ [2]. We ensured that the collected reports were created after the cutoff date (December 2023) of the LLM that we considered in our preliminary study (Section IV) and the evaluation of our approach (Section VI). This resulted in a set of 2,023 bug reports. Following the methodology described earlier in this section, two authors started reproducing the bug reports describing non-crashing functional failures (processing bug reports randomly) until they obtained 16 bug reports (from 6 additional apps) and corresponding FBOs. The **16 FBO4A**$_{UD}$ **bug reports** correspond to 10% of FBO4A$_{AE}$ (159 reports) and represent an unseen "test set".

It required four person months to build FBO4A—further details can be found in our artifact [34].

IV. Preliminary Study

Our preliminary study investigates the effectiveness of different LLM prompt configurations to automatically generate FBOs. The goal is to understand how these configurations perform and gather insights for building an approach for FBO creation. We investigated the following research question (RQ):

RQ$_1$: *How effective are different LLM prompt configurations at automatically generating FBOs?*

A. Methodology

We systematically investigated the effectiveness of different text-based prompt configurations for creating FBOs as related work found that most bug reports for Android apps are text-based [2]. We chose GPT-4 Turbo (version `gpt-4-0125-preview` [43]) as the LLM for the study as it was the most recent model that had the most recent knowledge cutoff in the GPT-4 family when we started the study in February 2024 [44].

The prompt configurations use: (i) the XML hierarchy of the app screen that shows the failure, (ii) different content elements of a bug report (title, S2Rs, OB, and EB), and (iii) different prompting strategies. We consider configurations based on the full bug report (full BR) and based *only* on the OB. We experimented only with the OB because it is the main element that describes the failure. Given that we aim for our approach to be automated and end-to-end, to identify the sentences related to the OB, we used BEE [45], a state-of-the-art automated tool for the task. We do not experiment with only the EB as prior work has found that this information is often missing in bug reports [28].

We start from two *base* prompt configurations (the first based on the full BR and the second based on the OB) and investigate configurations that extend these two with prompt strategies that are: 1) applicable to our problem domain, 2) popular for context learning [46–48, 48–50], and 3) effective as shown in prior work on bug reporting [20, 22]. The strategies we considered are: zero-shot chain of thought (ZS-CoT) [48], one-shot [51], chain-of-thought (CoT) [49], and chaining [50]. ZS-CoT prompting, which asks an LLM to perform step-by-step reasoning, has been shown effective in several domains [48]. One-shot prompting can lead to better overall performance by providing examples of questions and expected results in the prompt [51]. CoT can be combined with few-shot prompting to illustrate the thought process behind a certain example so that the LLM can learn to solve a given problem [49]. Chaining can lead to a better overall performance by decomposing a prompt for a task into multiple prompts [50]. We also investigated the combination of multiple strategies where appropriate. Table II reports all the *Prompt Configurations (PCs)* considered in our study. Each configuration uses the XML hierarchy associated with the failure screen (manually provided in this preliminary study) and the relevant content from the bug report (either the full BR or OB). We now detail the configurations by discussing the instantiations of the different prompting strategies in our context. Due to space limitations, we include a summarized template only for the base strategy. Detailed information on all prompts are available in our artifact [34].

1) Base: This configuration uses (i) the XML hierarchy of the failure screen and (ii) the relevant bug report content. *PC01-Template* reports the template when the base prompt is used with the full bug report (PC01 in Table II).

***PC01-Template*: (Full Bug Report, Base)**

[XML Hierarchy]
Given the previous XML hierarchy containing a failure in a mobile app screen and the following bug report:
[Full Bug Report]
Write one or more assertions using `UIAutomator` that identify the failure in the XML hierarchy.

2) ZS-CoT: This configuration extends the base by asking the LLM to perform step-by-step reasoning through the sentence: "Explain each step of your reasoning".

3) One-shot: We systematically investigate the effectiveness of one-shot prompting. For each one-shot configuration, we evaluate all prompt instances with our dataset of 17 bug reports. In each instance, we use one bug report from FBO4A$_{PS}$ and its ground truth FBO as the *example* for the prompts that generate FBOs for the remaining

16 bug reports. All instances use a different bug report from FBO4A$_{PS}$ as the example. Every one-shot configuration leads to the investigation of at least $17 \times 16 = 272$ prompts. We did not consider a systematic analysis of a two-shot-based configuration as the time cost to manually analyze all the results is high (see Section IV-B). In summary, a one-shot prompt configuration uses (i) the XML hierarchy of the example bug report's failure screen, (ii) the relevant content of the example report, (iii) the manually created FBO for the example, (iv) the XML hierarchy of the failure screen under analysis, and (v) the relevant content of the bug report under analysis.

4) CoT: We combine CoT with one-shot prompting by extending a one-shot prompt with the thought process of how we derived the code of the FBO associated with the example. Two authors wrote the thought process for each example together.

5) Chaining: We use chaining to decompose the task of creating an FBO into two steps. First, the LLM identifies the GUI element(s) related to the failure. Then, it generates assertions for those elements. This strategy involves two prompts. The first prompt asks the LLM to find the XML code for the affected elements. The second prompt asks the LLM to generate assertions based on `UIAutomator` code that can locate the LLM-identified elements on the XML hierarchy. Since chaining requires two prompts per bug report, it doubles the total number of prompts. For example, PC05 uses 17$\times$ 2 = 34 prompts.

We executed the prompts associated with the configurations and the bug reports in FBO4A$_{PS}$ on the LLM (GPT-4 Turbo). Although we set GPT-4's temperature to zero, to limit output randomness, it is still possible that the LLM provides a different answer when the same prompt is executed multiple times. For this reason, we executed each prompt *three* times and considered an FBO correctly generated only if it matched our ground truth FBO all three times (see section IV-B for details on the evaluation metrics we used). This setting led to executing 10,404 prompts and the analysis of 6,936 FBOs. We decided to limit the number of repetitions to three due to the high number of FBOs that need to be analyzed. Due to the cost of the analysis, we also focus the preliminary study on suitable/expected inputs (*i.e.*, provided failing screens). In Section VI we assess the quality of our derived approach on non-failing screens when integrated with a reproduction tool.

B. Metrics

To measure the effectiveness of the prompt configurations, we compared the generated FBOs with our ground-truth FBOs and computed four metrics for each configuration: *accuracy* (A), *precision* (P), *recall* (R), and F_1 *score* (F_1).

To compute the metrics, we executed the tests with the generated FBOs and performed manual inspections. We executed the tests with generated oracles on both the buggy and the fixed versions of the app. Manual checks were performed to identify spurious FBOs that do not semantically match the ground truth but fail on the buggy version and incidentally pass on the fixed version due to concurrent changes that are unrelated to the failure. This situation can happen for FBOs that check that a correct element should be on screen, but it is not (C03). These oracles estimate the element that should be on screen and that could be the one associated with the concurrent change.

For each test, the authors inspected the identified GUI elements and assertions' generated by the LLM and assessed whether the FBO was correct. An FBO is correct if it fails on the buggy version of the app, passes on the fixed version, and *semantically* matches the ground truth elements and assertions (*i.e.*, a syntactic match is not necessary). This was done because, for example, an assertion that reveals a failure can be created from different GUI element properties. The authors used the generated FBO code, ground truth FBO code, the bug report text, the XML hierarchy, and the failure screen to determine the correctness of FBOs' elements and assertions. In this task, we judged semantic oracle equivalence by looking at the elements and the assertions involved in the oracles. For two oracles to be judged semantically equivalent, they must operate on the same elements and must resolve to the same check logically, but do not need to have identical implementations. We considered FBOs that selected the same element through different properties to be equivalent. Generated FBOs that are more specific than the ground-truth FBOs were also considered equivalent. For example, generated FBOs that checked that an element existed and that they had a specific text were considered equivalent to ground-truth FBOs that only checked that the element existed if both functioned equivalently.

During the analysis, the authors labeled each FBO as *successfully-generated*, *incorrectly-generated*, or *not-usable*. A *successfully-generated* FBO semantically matches the ground truth FBO, fails on the buggy app (*i.e.*, it identifies the failure), and passes on the fixed app. An *incorrectly-generated* FBO does *not* match the ground truth but it fails on the buggy app (this can happen when a generated assertion fails but it is not related to the reported failure). A *not-usable* FBO does not contain any generated code by the LLM, has uncompilable code, or contains code passing on the buggy app.

Initially, two authors independently analyzed a shared set of FBOs comprising 10% of the generated set (6,936) to assess the potential subjectiveness of the inspections. During this initial analysis, the two authors had an agreement of 0.924, in terms of Cohen's Kappa [52], signaling very high inter-rater reliability. Disagreements were solved via discussion and consensus between the authors. Due to the high reliability, the two authors analyzed the remaining 90% by splitting and analyzing the FBOs independently.

If an FBO was *successfully-generated* in all of the three runs associated with a specific FBO generation task and it was consistent (*i.e.*, the generated code matched across the three runs), we consider the FBO generation task as a *true positive* (TP) because it is possible to reliably provide a correct FBO to a developer. If an FBO was *incorrectly-generated* in all of the three runs, we consider the FBO generation task as a *false positive* (FP) as the information provided to the developer is consistently incorrect. The remaining cases/combinations (which include not-usable FBOs) are considered *false negatives* (FN) as no consistent FBO can be provided to the

TABLE II: Preliminary study results.

ID	*configuration*		A	P	R	F_1
PC01	Full BR	Base	17.5	50.0	21.4	30.0
PC02		ZS-CoT	5.9	50.0	6.3	11.1
PC03		One-shot (BR15)	37.5	66.7	46.2	54.5
PC04		One-shot+CoT (BR16)	43.8	63.6	**58.3**	60.9
PC05		Chaining	23.5	36.4	40.0	38.1
PC06		**Chaining+ZS-CoT**	**47.1**	**72.7**	57.1	**64.0**
PC07		Chaining+One-shot (BR16)	37.5	54.5	54.5	54.5
PC08		Chaining+One-shot+CoT (BR14)	31.3	55.6	41.7	47.6
PC09	OB	Base	5.9	25.0	7.1	11.1
PC10		ZS-CoT	5.9	50.0	6.3	11.1
PC11		One-shot (BR10)	31.3	83.3	33.3	47.6
PC12		One-shot+CoT (BR17)	31.3	71.4	35.7	47.6
PC13		Chaining	11.8	15.4	33.3	21.1
PC14		Chaining+ZS-CoT	11.8	18.2	25.0	21.1
PC15		Chaining+One-shot (BR11)	31.3	50.0	45.5	47.6
PC16		Chaining+One-shot+CoT (BR01)	31.3	50.0	45.4	47.6

developer given the failure screen information. We decided to map the terms successfully-generated, incorrectly-generated, and not-usable to the traditional TP, FP, and FN metrics to best measure and represent the outcome of the FBO generation task with respect to the available ground truth.

We compute A, P, R, and F_1 using TPs, FPs, FNs, and related formulas [53]. In our context, given a certain number of TPs, having a higher P than R is important as FPs may prompt the developers to use FBOs that do not detect the failure, while FNs are not as problematic because developers would anyway need to manually create FBOs. We consider that using the A, P, R, and F_1 metrics are appropriate to measure the effectiveness of the configurations because, by definition, they capture configuration usefulness and usage overhead from a developer's viewpoint.

C. Results

*1) **RQ**$_1$: How effective are different LLM prompt configurations at automatically generating FBOs?* Table II reports the results for each prompt configuration. For one-shot configurations, we report the results for the best-performing configuration setting (which we systematically explored through our methodology). The setting is a specific bug report used as the example in the prompt. We report results in this way as we are interested in the best possible approach for generating FBOs. For instance, the best setting for PC04 uses BR16 [54] as the prompt example.

The configuration that performs best is PC06 (boldfaced in Table II), which achieves the highest accuracy (47.1%), precision (72.7%), and F_1 score (64.0%). PC06 combines chaining with ZS-CoT and has eight TPs, three FPs, and six FNs. The second best-performing configuration is PC04 with seven TPs, four FPs, and five FNs (the sum is equal to 16 as we excluded the one-shot example bug report from the evaluation of the configuration). Considering all TPs across all configurations, we also observed that PC06 includes all the TPs obtained by the other configurations.

By inspecting the FBOs generated by PC06 and PC04, we also identified that PC06 correctly generated 12 element components while PC04 only generated nine. This result further highlights the higher potential of PC06 in generating FBOs as it is able to more accurately identify the elements affected by failures. When inspecting the FBOs generated by PC06 we observed that three FBOs had assertions that did not compile due to hallucinations, that is, the LLM created FBOs that used methods that did not exist in the UIAutomator API. Notably, PC06 does not include any prompt examples and we observed cases where this strategy was effective. For example, in two of three cases where PC06 correctly identified the relevant GUI elements but PC04 did not, we observed that the elements selected by PC04 were not present in the XML hierarchy and were copies of elements of the prompt examples. This is likely due to the vague failure description in the bug reports. We believe that at least in these cases, PC06 correctly identifies the GUI elements because it forces the LLM to not simply take the example information but to process the XML hierarchy to select the most likely GUI element associated with the failure.

Overall, considering that the results show that PC06 has the highest F_1 score, a low number of FPs (which we believe to be preferable over a low number of FNs as FPs would demand developer's time to be filtered out), and the highest accuracy in finding the elements affected by failures, we identified that PC06 is the best configuration for generating FBOs.

RQ$_1$ answer: The best performing prompt configuration (PC06) is based on the full bug report and combines prompt chaining with zero-shot chain-of-thought prompting.

V. Approach

Informed by the results of our preliminary study, we define AndroB2O, an approach that automatically generates FBOs for validating reported non-crashing, functional failures in Android apps. Figure 2 provides an overview of the approach. AndroB2O takes as input the bug report text and the XML hierarchy of the failing screen, which can be provided manually by a developer or automatically by an automated bug reproduction tool (*e.g.*, [24]). Based on these inputs, AndroB2O is able to *automatically* produce an FBOs as output, which by definition fail on the provided buggy screen as they detect the failure. AndroB2O executes in four phases: ① *element prompt creation*, ② *element code generation*, ③ *assertion prompt creation*, and ④ *failure-based oracle generation*. AndroB2O is based on the PC06 configuration from our preliminary study, which was the best performing method.

The *element prompt creation* phase ① produces an *element prompt* to identify the element component of the FBO. The prompt asks the LLM to identify the XML node of the element(s) affected by the failure from the XML hierarchy and the bug report. The prompt also includes a description of the expected output format. Informed by the preliminary study results, AndroB2O runs the prompt three times and collects the answers (*element prompt results*). This parameter is customizable and we set three as its default value, as our preliminary study showed this to be a balanced trade-off between the stability of multiple predictions (given LLMs' proclivity for non-determinism) and computational cost. The

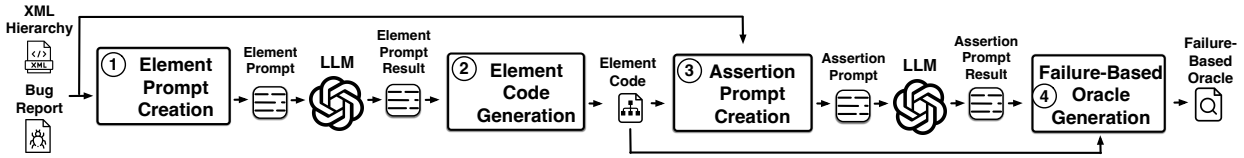

Fig. 2: Overview of the ANDROB2O approach.

approach proceeds to the next phase only when the LLM returns the same XML element across the runs.

The *element code generation* phase ② generates the code for selecting the element(s) from the GUI hierarchy using the `UIAutomator` API. This phase generates code for selecting the element(s) using the properties that uniquely identify the element(s) in the hierarchy. ANDROB2O prioritizes properties that provide textual information about an element (*e.g.*, an element's text or content description) over other properties. We made this choice as an element's text can provide semantic information about the element and the element code is used in the assertion prompt, which characterizes the next phase of the technique. If the element cannot be uniquely identified in the hierarchy, ANDROB2O stops and does not generate the FBO. If the XML returned by the element prompt contains a tree of elements (*e.g.*, in the case of a list element), this phase generates code that selects the root of the tree. If the LLM returns an XML element that is not present in the XML hierarchy, ANDROB2O generates code that identifies the element based on its text or content description (if available). We do not consider this result as erroneous as the LLM might be trying to create an XML element that should be on screen but actually is not (case C03 in Table I). We allow for disabling this aspect of the technique through a configuration parameter, which only affects the generation of oracles of type C03.

ANDROB2O does not use the LLM to generate the code of the element locators. It uses the LLM only to determine which GUI element(s) in the XML Hierarchy correspond to the element(s) associated with the reported failure. Establishing this correspondence requires advanced reasoning of different elements in the screen, mapping them to various information of the bug report (in particular, the OB). From the LLM-identified elements, ANDROB2O generates an element locator programmatically: in essence, the algorithm traverses the hierarchy and matches properties that uniquely identify an element.

The *assertion prompt creation* phase ③ produces the *assertion prompt*, which generates the assertion(s) that detect the failure under analysis. The prompt asks the LLM to generate the assertion(s) that checks the properties of the selected GUI element from the previous phase, according to the failure described in the bug report and the XML hierarchy. As in phase ①, ANDROB2O runs the assertion prompt three times. This is also a customizable parameter, using three as the default value, for similar reasons as stated earlier. We consider the generated code to be valid if it matches across runs. To avoid FBOs that do not compile due to hallucinations (which we observed in the manual inspections of our preliminary study), we include text in the assertion prompt that aims at limiting the presence of those issues: "*when operating on variables in the provided code your response needs to use* `UIAutomator` *methods from the following list:* `[UI Automator Method List]`".

Finally, the *failure-based oracle generation* phase ④ processes the assertion prompt results and provides the FBO as output. The approach extracts the code from the prompt, checks whether the code compiles, and if that is the case, ANDROB2O combines the element code and the assertion code into the FBO, which is the final output of the approach. ANDROB2O focuses only on generating the oracle component of a test, as the app interactions in the test can be either directly or indirectly (with small modifications) generated by reproduction tools (*e.g.*, [15]) or manually by developers.

VI. ANDROB2O'S EVALUATION

We evaluate the effectiveness of ANDROB2O by answering the following research questions (RQs):

RQ$_2$: *How effective is* ANDROB2O *in creating FBOs?*
RQ$_3$: *How effective is* ANDROB2O *on unseen data?*
RQ$_4$: *How effective is* ANDROB2O *when integrated with an automated bug reproduction tool?*
RQ$_5$: *How robust is* ANDROB2O *under different LLMs?*

A. Methodology

RQ$_2$ and **RQ**$_3$ assess ANDROB2O 's effectiveness in generating FBOs from bug reports and manually provided failure screens, simulating scenarios where developers supply screens after reproducing failures. We also compare ANDROB2O to two baselines derived from our preliminary study: PC01 (using the full bug report) and PC04 (using BR16 as the prompt example). PC01 represents the simplest strategy, while PC04 was the second-best configuration in preliminary results.

Experiments use GPT-4 Turbo `gpt-4-0125-preview` with temperature zero, as in Section IV, running each prompt three times (as ANDROB2O does internally). We evaluate ANDROB2O and baselines on FBO4A$_{AE}$ (152 failures) and FBO4A$_{UD}$ (16 failures), using FBO4A$_{AE}$ for main evaluation and FBO4A$_{UD}$ for generalization.

RQ$_4$ evaluates ANDROB2O combined with automated failure reproduction. We integrate ANDROB2O with REBL [24], which uses GPT-based reasoning to reproduce failures from bug reports by iteratively predicting and executing GUI actions, checking for reproduction, and generating reproduction traces, but does not produce assertions.

Before integration, we replicated REBL's original results using GPT-4 Turbo `gpt-4-0125-preview` to match versions. We extended REBL to save XML hierarchies and screenshots of failure screens, which serve as input to ANDROB2O.

Screenshots were manually validated to confirm reproduction correctness. After integration, we re-executed REBL to verify consistency and correctness of the collected data.

We ran both REBL+ANDROB2O and S+REBL+ANDROB2O, the latter adding manually supplied setup actions—GUI interactions required to configure the app prior to the first S2R, but not described in the bug report. This evaluates ANDROB2O in semi-automated, human-assisted scenarios. Two authors independently identified and reviewed setup actions following [2], identifying 3.6 and 3 setup actions per bug report in FBO4A$_{AE}$ and FBO4A$_{UD}$, respectively (FBO4A$_{AE}$ and FBO4A$_{UD}$ require 9.7 and 9.9 total setup actions on average).

RQ$_5$ evaluates the robustness of ANDROB2O when used with different LLMs. In this RQ, we evaluated ANDROB2O using GPT-4 Turbo and two additional LLMs: GPT-4o [55] (version `gpt-4o-2024-11-20`) and LLaMA 4 Maverick [56]. We selected GPT-4o as it is a newer model from OpenAI as compared to GPT-4 Turbo and it is optimized for code generation and reasoning [55]. We use LLaMA 4 Maverick as it is open-source and can perform complex reasoning tasks [57]. We provide bug reports and XML hierarchies to ANDROB2O with the PC01 and PC04 baselines (as done in **RQ**$_2$ and **RQ**$_3$) to assess whether their relative performance is consistent across the different models.

Due to the cost of manually validating every FBO generated by the LLMs (as done in Sections IV and VI), we performed this experiment on a sample of 35 bug reports from the 169 reports in FBO4A$_{PS}$ and FBO4A$_{AE}$ combined. We first partitioned the 169 reports as successfully (98) and unsuccessfully (71) handled by ANDROB2O. We then performed stratified sampling to include at least 20% of the reports in each partition and one bug report from each FBO category (see Table I).

B. Metrics

To answer the RQs, we use the same metrics used in our preliminary study: accuracy (A), precision (P), recall (R), and F_1 score (F_1). For **RQ**$_2$, **RQ**$_3$, and **RQ**$_5$, we use the same definitions for true positives (TPs), false positives (FPs), and false negatives (FNs) as in our preliminary study (see Section IV-B)–the three RQs and the preliminary study are based on the same context: manually provided failing screens.

For **RQ**$_4$, we refine the definitions of TPs, FPs, and FNs to account for the integration with REBL (*i.e.*, when failure screens are provided automatically). The refinement is needed as REBL can produce false positives by itself, that is, REBL can report that it reproduced a failure but actually it did not. When REBL identifies that it reproduced a failure, the tool marks the failure as being reproduced, and we use the acronym MR (**M**arked as **R**eproduced) to refer to this case. Otherwise, we label the failure as NMR (**N**ot **M**arked as **R**eproduced)—for this case, ANDROB2O will not receive any input screens to work with so it cannot generate FBOs. If an MR failure was actually reproduced (*i.e.*, the resulting screen shows the failure), we consider this case as a *ReBL true positive* (RTP); otherwise, we label the failure as a *ReBL false positive* (RFP).

TABLE III: Evaluation results for ANDROB2O (with manually provided failing screens) and the baselines on FBO4A$_{AE}$ and FBO4A$_{UD}$ across failure check categories.

		FBO4A$_{AE}$										FBO4A$_{UD}$
		C01	C02	C03	C04	C05	C06	C07	C08	C09	Tot	Tot
# of failures		55	50	30	5	5	3	2	1	1	152	16
PC01 (Baseline)	*A*	30.9	32.0	20.0	40.0	20.0	0	0	**100**	0	28.3	26.7
	P	37.8	42.1	21.4	50.0	25.0	0	0	**100**	0	34.7	28.6
	R	63.0	57.1	75.0	66.7	50.0	0	N/A	**100**	N/A	60.6	66.7
	F_1	47.2	48.5	33.3	57.1	33.3	0	0	**100**	0	44.1	40.0
PC04 (Baseline)	*A*	45.5	38.0	13.3	20.0	20.0	0	0	**100**	**100**	34.2	25.0
	P	61.0	46.3	15.4	33.3	25.0	0	0	**100**	**100**	42.6	40.0
	R	64.1	67.9	50.0	33.3	50.0	N/A	N/A	**100**	**100**	63.4	40.0
	F_1	62.5	55.1	23.5	33.3	33.3	0	0	**100**	**100**	51.0	40.0
AndroFC	*A*	**72.7**	**66.0**	**43.3**	**60.0**	**40.0**	**0**	0	**100**	**100**	**61.2**	**62.5**
	P	**83.3**	**78.6**	**54.2**	**60.0**	**50.0**	0	0	**100**	**100**	**72.7**	**71.4**
	R	**85.1**	**80.5**	**68.4**	**100**	**66.7**	0	N/A	**100**	**100**	**79.5**	**83.3**
	F_1	**84.2**	**79.5**	**60.5**	**75.0**	**57.1**	0	0	**100**	**100**	**75.9**	**76.9**

In the case of ReBL true positives (RTPs), when the resulting screen shows the failure, ANDROB2O can successfully generate an FBO (TP), incorrectly generate an FBO (FP_1), or not generate an FBO (FN). In the case of a REBL false positive (RFP), when the resulting screen does not show the failure, ANDROB2O can incorrectly generate an FBO (FP_2) or not generate an FBO, *i.e.*, a *true negative* (TN). Table IV illustrates how the refined definitions relate to REBL results. Two authors analyzed and labeled the REBL+ANDROB2O results using the same methodology used in the preliminary study.

C. Results

*1) **RQ**$_2$: How effective is* ANDROB2O *in creating FBOs?* Table III (under FBO4A$_{AE}$) reports A, P, R, and F_1 for ANDROB2O and the baselines, itemized by FBO category. Categories C10 and C11 are excluded as they were used in the preliminary study. Overall, ANDROB2O achieves 61.2% A, 72.7% P, 79.5% R, and 75.9% F_1, successfully generating FBOs for over half (93/152) of the bug reports, with 35 false positives. Compared to the baselines, ANDROB2O generates over twice as many FBOs while producing fewer FPs (70 for PC04, 81 for PC01). In the element selection step, ANDROB2O correctly identifies elements in 65.8% of cases, compared to 38.2% (PC01) and 43.4% (PC04). By category, ANDROB2O performs best on C01, with 72.7% A, 83.3% P, 85.1% R, and 84.2% F_1—the largest category—highlighting its practical potential. ANDROB2O also successfully handles C03, despite missing elements in the corresponding XML hierarchies for 30 reports. This can be attributed to the LLM's ability to infer missing elements directly from the bug report description. The only two categories that no methods were able to successfully create FBOs for were C06 and C07. For C07 (keyboard failures), creation of FBOs involved the use of methods outside of the UIAutomator API to retrieve keyboard properties, which is out of scope for ANDROB2O. For C06 (failures related to the location of elements on the screen through coordinate vlaues), ANDROB2O was able to select the correct elements but it was never able to identify the precise location values to create a valid FBO. Finally, looking at the reasons why our approach failed to create the assertions, we identified that there were no cases of hallucinations,

TABLE IV: Evaluation results for ANDROB2O on FBO4A$_{AE}$ and FBO4A$_{UD}$ when used in combination with REBL.

Dataset	REBL+ANDROB2O					S+REBL+ANDROB2O				
	REBL			ANDROB2O		S+REBL			ANDROB2O	
FBO4A$_{AE}$	MR	RTP	32	TP	19	MR	RTP	136	TP	75
				FP$_1$	2				FP$_1$	4
				FN	11				FN	57
		RFP	17	FP$_2$	5		RFP	6	FP$_2$	1
				TN	12				TN	5
	NMR		120	-		NMR		27	-	
FBO4A$_{UD}$	MR	RTP	5	TP	5	MR	RTP	14	TP	11
				FP$_1$	0				FP$_1$	0
				FN	0				FN	3
		RFP	0	FP$_2$	0		RFP	0	FP$_2$	0
				TN	0				TN	0
	NMR		12	-		NMR		3	-	

highlighting the usefulness of the improvements included in ANDROB2O based on the preliminary study results.

RQ$_2$ answer: ANDROB2O generates FBOs in 61.2% of the cases and outperforms baselines in accuracy, precision, recall, and F_1 score.

*2) **RQ**$_3$: How effective is* ANDROB2O *on unseen data?* The section of Table III under FBO4A$_{UD}$ reports the results for this RQ. Our approach was able to create FBOs for 10 out of 16 bug reports (A of 62.5%) and achieves a P, R, and F_1 of 71.4%, 83.3%, and 76.9%, respectively. ANDROB2O significantly outperforms the baselines by having the highest number of successfully generated FBOs (10 for ANDROB2O *vs.* four for PC01 and four for PC04) and the lowest number of FPs (four for ANDROB2O *vs.* 10 for PC01, and six for PC04). Overall, ANDROB2O is the best approach and performs well on unseen data.

RQ$_3$ answer: ANDROB2O correctly generated 10 out of 16 FBOs for likely unseen reports by the used LLM, outperforming the baselines, thus providing positive preliminary evidence of ANDROB2O's generalizability.

*3) **RQ**$_4$: How effective is* ANDROB2O *when integrated with an automated bug reproduction tool?* Table IV reports the results associated with the tools' integration under the REBL+ANDROB2O header and their integration supplemented with setup steps under the S+REBL+ANDROB2O header, for the FBO4A$_{AE}$ and FBO4A$_{UD}$ datasets.

The results for REBL+ANDROB2O on FBO4A$_{AE}$ show that REBL marked 49 bug reports as reproduced (MR), with 32 true positives (RTPs) and 17 false positives (RFPs). Among the 49 XML hierarchies provided to ANDROB2O, it generated 19 FBOs (TPs), seven FPs (two FP$_1$s and five FP$_2$s), and failed in 11 cases (FNs). For the 17 non-reproduced failures, ANDROB2O correctly refrained from generating FBOs in 12 cases (TNs). The resulting A, P, R, and F_1 were 63.3%, 73.1%, 63.3%, and 67.9%, respectively. On FBO4A$_{UD}$, REBL reproduced five failures (RTPs), with ANDROB2O generating FBOs for all.

With setup steps (S+REBL+ANDROB2O), REBL marked 136 bug reports as reproduced (MR), with 136 RTPs and 6 RFPs. Among these, ANDROB2O produced 75 TPs, five FPs (four FP$_1$s, one FP$_2$), 57 FNs, and five TNs, yielding A, P, R, and F_1 of 56.3%, 93.8%, 56.8%, and 70.7%, respectively. On FBO4A$_{UD}$, REBL reproduced 14 failures, with ANDROB2O generating 11 TPs and 4 FNs.

Manual analysis of all FPs revealed that the LLM asserted the presence of elements mentioned in the bug reports but unrelated to the failures. This may be mitigated by filtering content (e.g., S2Rs) from bug reports during reproduction. Notably, ANDROB2O declined to generate FBOs in 17 of 23 cases (sum of TNs across RFPs) where REBL reproduced failures that were not actually present. This suggests potential for ANDROB2O to provide feedback to reproduction tools like REBL by recognizing when a failure is absent on a screen.

Overall, the results provide preliminary evidence that ANDROB2O is also effective when used with a reproduction tool, *i.e.*, when a (presumably) failing screen is provided automatically as input to ANDROB2O.

RQ$_4$ answer: ANDROB2O generated 19 FBOs out of 32 correctly reported failure screens provided by an automated failure reproduction tool. When the reproduction tool is supplied with setup steps, ANDROB2O generates 75 FBOs out of 136 correctly reported failure screens. This result together with the low number of incorrectly generated FBOs and ANDROB2O's ability to not generate FBOs on invalid screens show preliminary evidence of the ANDROB2O's effectiveness when combined with a reproduction tool.

*4) **RQ**$_5$: How robust is* ANDROB2O *under different LLMs?* The results of the experiment are reported in Table V. For ANDROB2O and the baselines, the three models performed similarly and the best approach was still ANDROB2O. GPT-4o was the best performing model and performed at or slightly above the other models (with modest absolute improvements). Notably, ANDROB2O achieved the highest accuracy when paired with GPT-4o (A=61.8%, P=87.5%, R=67.7%, F_1=76.4%), followed by GPT-4 Turbo and LLaMA 4 Maverick with same scores (A=52.9%, P=78.3%, R=62.1%, F_1=69.3%). For the baselines, GPT-4o and LLaMA 4 Maverick either matched or slightly improved upon GPT-4 Turbo. However, ANDROB2O, with any of the LLMs, outperforms the baselines by a significant margin. These findings suggest that ANDROB2O's accuracy is consistent across models.

RQ$_5$ answer: ANDROB2O's performance across LLMs is consistent and substantially higher than the baselines'.

VII. DISCUSSION

Qualitative analysis. Several patterns emerged in both the successes and failures of ANDROB2O. One of the most persistent challenges was C03, where missing elements in the buggy XML hierarchy made it essentially impossible for the model to select them during the FBO construction step. Occasionally, however, ANDROB2O successfully handled C03 cases when the element selection step proposed non-existent elements

TABLE V: Performance across LLMs.

Method	Metric	GPT-4 Turbo	GPT-4o	LLaMA 4
PC01	Accuracy	27.3	35.3	35.3
	Precision	71.4	85.7	85.7
	Recall	33.3	37.5	37.5
	F1	45.5	52.2	52.2
PC04	Accuracy	39.4	47.1	47.1
	Precision	73.7	84.2	84.2
	Recall	48.3	51.6	51.6
	F1	58.3	64.0	64.0
ANDROB2O	Accuracy	52.9	**61.8**	52.9
	Precision	78.3	**87.5**	78.3
	Recall	62.1	**67.7**	62.1
	F1	69.3	**76.4**	69.3

that correctly reflected what should have been present in a non-buggy app. These successes typically occurred when the bug report explicitly described the missing element, often specifying its expected text.

In addition to C03, ANDROB2O struggled in cases where detection depended on variable attributes such as screen position, particularly for scroll positioning failures. Another recurring issue was the models' inability to handle non-English text; in such cases, the LLMs frequently substituted English translations rather than using the correct localized strings. This problem was consistent across all three models.

While the poor performance of C03 was slightly exacerbated by the two-step process, the performance gains of ANDROB2O outweighed these limitations. ANDROB2O achieved nearly double the element selection performance of PC01 and over a 50% improvement compared to PC04.

There were two large contributors to this improvement. First, ANDROB2O handled cases requiring reasoning over multiple elements more effectively—whether counting elements, comparing multiple selections, or checking for the presence of several items. Second, the two-step design stopped the LLMs' tendency to incorrectly generate GUI interactions together with assertions as part of FBOs, which prevented failure detection by changing the app state during test case execution. By providing a more structured framework, the two-step approach only generated assertions as expected.

Looking forward, several extensions may address remaining failures. For C03, providing historical XML hierarchies may help infer missing elements. Specifying the target language during prompting may improve non-English handling. For variable attributes like scroll position, supplying user input traces and their intended effects may improve reasoning about dynamic GUI changes.

Category-specific FBO generation. In this work, we identified 11 categories of FBOs. Although we studied a general approach for generating FBOs applicable to any failure, we observed that there is the potential for an FBO-specific approach for failures of some categories. For example, for the category of FBOs that check for the position of an element on the screen (C06), we observed that prompts with an example from this category performed better than using examples from other categories. Another interesting situation happens for FBOs that need to assert that right element (which is not on screen) should be on screen (C03). For this category, we observed cases in which the LLM is able to generate proper assertions but the element used for the assertions is not the correct one. In these cases, our approach could be supplemented with code from previous versions of the app to account for cases in which the failure was introduced through regression. Future work could investigate automated methods that jointly categorize, then generate FBOs using category-specific approaches.

Using more artifacts. In our work, we did not consider reports whose failures require asserting on properties not available in the XML hierarchy (*e.g.*, the color of a GUI element). We believe that our approach could be enhanced by using more detailed information about the GUI and its elements provided in an augmented version of the XML hierarchy or other artifacts, to be able to identify and create more types of FBOs. For example, the color of a GUI element could be detected and provided in the hierarchy to help determine if a failure has occurred. Future work in this direction could investigate the trade-off between having more detailed artifacts and a limited context window in the LLM. Additionally, using GUI information from the current and previous versions of the app could also help generate FBOs by providing the LLM with more contextual information about differences/regressions.

Integration with automated failure reproduction tools. Our integration with REBL is a promising first step of potentially many. In particular, we observed that REBL produces false positives (*i.e.*, failures marked as reproduced that actually are not) that could be filtered out by ANDROB2O as our approach does not generate an FBO in those cases. We believe that ANDROB2O could be adapted or extended to work with automated failure reproduction tools to guide reproduction as a feedback mechanism for tools to decide whether to continue or stop failure reproduction.

VIII. LIMITATIONS & THREATS TO VALIDITY

Limitations. For oracles of type C02, which represents oracles that check that the wrong element is not on screen, the FBO generated by ANDROB2O attempts to search for the GUI element in the GUI hierarchy by checking whether or not there is one that matches the property value(s) suggested by the LLM. This type of oracle could miss failures if it is improperly applied to a screen other than the failing screen (which also does not contain the component) during test execution. However, this scenario is unlikely. While app changes or non-determinism could change the test behavior such that it does not arrive at the proper failure screen, the test is likely to fail before triggering the oracle due to the non-execution of GUI actions leading to the failure screen. This type of oracle could be strengthened in future work by also including the name of the screen on which it needs to be applied as part of the oracle.

External validity. Our results might not generalize to other bug reports. To mitigate this, we built our dataset using the two largest collections of reproducible bug reports. Our evaluation includes 152 bug reports covering various failures across 27 apps from different domains. A limitation of our

preliminary study is that it relies on only 17 bug reports from failure screens, and we ran prompts three times. This was necessary due to the expensive manual analysis of over six thousand configuration results. Although we performed an evaluation on data that we believe was not seen by the model considered, the model might have seen similar bug reports and apps, affecting the results associated with this part of the evaluation. However, the 16 cases follow a similar distribution of oracle categories to the 152 reports used in the approach evaluation and are diverse in terms of bug types and apps. This diversity reduces threats to the external validity of this set of unseen cases. It should be noted that developing this dataset required substantial manual effort to (i) reliably reproduce bugs, (ii) construct the ground truth, and (iii) validate the oracles produced by the approach and the baselines. We plan to consider a large dataset in future work.

Internal and construct validity. Our results may be affected by errors in our analysis scripts or approach implementation. To reduce this risk, we thoroughly tested our code and manually reviewed the results. Additionally, since some findings rely on qualitative analysis, differences in rater interpretation could be a factor. However, when applicable, we measured inter-rater reliability, which was consistently high.

IX. Related Work

Automated Bug Reproduction. Existing approaches focus on reproducing crashing failures by extracting S2Rs from bug reports, parsing steps, matching them to GUI elements, and executing corresponding GUI-level events [14–23]. These techniques rely on heuristics [14–16], neural models [17, 18], and LLMs [19, 20]. Additional strategies leverage stack traces [14, 58], search-based app exploration [58], reinforcement learning [21, 23], and LLM-based reasoning [19, 20, 22, 23].

REBL [24] addresses the problem of failure reproduction by using an LLM paired with reinforcement learning to explore the app and identify sequences of user actions that trigger failures. Its goal is to reach the failure state based on crash reports or stack traces; however, it does not generate test oracles to validate the failure, instead it uses a stopping mechanism based on LLM feedback. In contrast, ANDROB2O focuses on oracle generation, synthesizing test assertions from bug reports and UI hierarchies for failure validation. While the two approaches address different stages of the debugging process, they are complementary and, as we show in our results, could be integrated together for improved performance.

Most LLM-based methods target mobile app crashes, with some adapting to libraries [19, 20, 58, 59]. LIBRO [19, 20] uses LLMs to generate scripts for library failure reproduction, but these include assertions only for libraries, not failures in GUIs. Our work extends prior work by automatically generating oracles for non-crashing functional failures. To our knowledge, ANDROB2O is the first to generate test oracles for mobile app failures reported in bug reports.

Test Oracle Automation. Test oracle generation has been studied across domains [60–63], leveraging diverse techniques such as formal specification [64], static analysis [65], fuzzing [25], and computer vision [66, 67]. Mobile app-specific oracle generation has focused on GUI display issues [66–69], code-based failure detection [70], and setting-related defects [71]. However, these approaches do not generate oracles from bug reports.

Zaeem *et al.* [72] and Baral *et al.* [29] classified mobile app failure oracles and proposed techniques for detecting invariant violations. Our failure categorization builds on Baral *et al.*'s taxonomy. Liu *et al.* [73] introduced GPTDroid, using iterative LLM-based Q&A for app interaction, while Yoon *et al.* [74] developed LLM agents to generate test scenarios. As opposed to these works, we are the first to explore LLM-driven assertion-based test oracle generation for verifying functional mobile app failures reported in bug reports.

X. Conclusion

We explored the feasibility of automatically generating test oracles for validating non-crashing, functional failures described in Android app bug reports. Due to the variability of the manifestation of such failures and their descriptions, we first empirically study the capabilities of an LLM for the task, and then develop a technique, ANDROB2O, for the fully automated generation of failure-based oracles from bug reports. An empirical evaluation of ANDROB2O on our newly derived benchmark (FBO4A) illustrates that the approach can effectively generate correct failure-revealing assertions both when failure screens are manually and automatically provided.

Acknowledgments

This research has been supported in part by NSF grants CCF-1955853 and CCF-2441355. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Y. Song, J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Toward interactive bug reporting for (android app) end-users," in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 344–356.

[2] J. Johonson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, "An empirical investigation into the reproduction of bug reports for Android apps," in *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'22)*, Honolulu, Hawaii, March 2022.

[3] M. Fazzini, K. Moran, C. Bernal-Cardenas, T. Wendland, A. Orso, and D. Poshyvanyk, "Enhancing mobile app bug reporting via real-time understanding of reproduction steps," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1246–1272, 2022.

[4] J. Mahmud, A. Saha, O. Chaparro, K. Moran, and A. Marcus, "Combining language and app ui analysis for the automated assessment of bug reproduction steps," in *Proceedings of the IEEE/ACM 33rd International Conference on Program Comprehension (ICPC'25)*, 2025, pp. 1–12.

[5] A. Saha and O. Chaparro, "Decoding the issue resolution process in practice via issue report analysis: a case study of firefox," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE'25)*, 2025, pp. 2316–2328.

[6] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," in *Proceedings of the 2019 27th ACM joint meeting on the foundations of software engineering (ESEC/FSE'19)*, 2019, pp. 86–96.

[7] Y. Song, J. Mahmud, N. De Silva, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Burt: A chatbot for interactive bug reporting," in *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering(ICSE'23)*, 2023, pp. 170–174.

[8] A. Adnan, A. Saha, and O. Chaparro, "Sprint: An assistant for issue report management," in *Proceeding of IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR'25)*, 2025, pp. 86–90.

[9] J. Mahmud, N. De Silva, S. A. Khan, S. H. Mostafavi, S. M. H. Mansur, O. Chaparro, A. A. Marcus, and K. Moran, "On using gui interaction data to improve text retrieval-based bug localization," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608139

[10] J. Mahmud, J. Chen, T. Achille, C. Alvarez-Velez, D. D. Bansil, P. Ijieh, S. Karanch, N. De Silva, O. Chaparro, A. Marcus *et al.*, "Ladybug: A github bot for ui-enhanced bug localization in mobile apps," in *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME'25)*, 2025, p. (to appear).

[11] A. Saha, Y. Song, J. Mahmud, Y. Zhou, K. Moran, and O. Chaparro, "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)*, 2024, pp. 1249–1261.

[12] O. Chaparro, J. M. Florez, and A. Marcus, "Using bug descriptions to reformulate queries during text-retrieval-based bug localization," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2947–3007, 2019.

[13] ——, "Using observed behavior to reformulate queries during text retrieval-based bug localization," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 376–387.

[14] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk, "Generating reproducible and replayable bug reports from android application crashes," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 48–59.

[15] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018, pp. 141–152.

[16] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "Recdroid: Automatically reproducing Android application crashes from bug reports," in *IEEE/ACM International Conference on Software Engineering (ICSE'19)*, 2019, pp. 128–139.

[17] H. Liu, M. Shen, J. Jin, and Y. Jiang, "Automated classification of actions in bug reports of mobile apps," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 128–140.

[18] Y. Zhao, T. Su, Y. Liu, W. Zheng, X. Wu, R. Kavuluru, W. G. Halfond, and T. Yu, "Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–33, 2022.

[19] S. Kang, J. Yoon, N. Askarbekkyzy, and S. Yoo, "Evaluating diverse large language models for automatic and general bug reproduction," *arXiv preprint arXiv:2311.04532*, 2023.

[20] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[21] Z. Zhang, R. Winn, Y. Zhao, T. Yu, and W. G. Halfond, "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*, 2023, pp. 411–422.

[22] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[23] Y. Huang, J. Wang, Z. Liu, Y. Wang, S. Wang, C. Chen, Y. Hu, and Q. Wang, "Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[24] D. Wang, Y. Zhao, S. Feng, Z. Zhang, W. G. J. Halfond, C. Chen, X. Sun, J. Shi, and T. Yu, "Feedback-driven automated whole bug report reproduction for android apps," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '24. ACM, Sep. 2024, p. 1048–1060. [Online]. Available: http://dx.doi.org/10.1145/3650212.3680341

[25] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Association for Computing Machinery, October 2021.

[26] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[27] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," 2023. [Online]. Available: https://arxiv.org/abs/2303.18184

[28] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 396–407. [Online]. Available: https://doi.org/10.1145/3106237.3106285

[29] K. Baral, J. Johnson, J. Mahmud, S. Salma, M. Fazzini, J. Rubin, J. Offutt, and K. Moran, "Automating gui-based test oracles for mobile apps," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024.

[30] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[31] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[32] C. Parnin, G. Soares, R. Pandita, S. Gulwani, J. Rich, and A. Z. Henley, "Building your own product copilot: Challenges, opportunities, and needs," *arXiv preprint arXiv:2312.14231*, 2023.

[33] "Android uiautomator http://developer.android.com/tools/help/uiautomator/index.html."

[34] J. Johnson, J. Mahmud, O. Chaparro, K. Moran, and M. Fazzini, "Generating failure-based oracles to support testing of reported failures in android apps," Online, https://zenodo.org/records/15556567, last access May 2025.

[35] Google, "Layouts in views," 2024. [Online]. Available: https://developer.android.com/develop/ui/views/layout/declaring-layout

[36] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su, "An empirical study of functional bugs in android apps," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1319–1331. [Online]. Available: https://doi.org/10.1145/3597926.3598138

[37] Google, "Build and run your app," Online, https://developer.android.com/studio/run, last access March 2025.

[38] K. Krippendorff, "Reliability in content analysis: Some common misconceptions and recommendations," *Human communication research*, vol. 30, no. 3, pp. 411–433, 2004.

[39] ——, *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[40] Campbell, J. L., C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological methods & research*, vol. 42, no. 3, pp. 294–320, 2013.

[41] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[42] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative data analysis: A methods sourcebook*. Sage publications, 2018.

[43] OpenAI, "Models," Online, https://platform.openai.com/docs/models/gpt-4-turbo, last access March 2025.

[44] ——, "New embedding models and api updates," 2024. [Online]. Available: https://openai.com/index/new-embedding-models-and-api-updates

[45] Y. Song and O. Chaparro, "Bee: a tool for structuring and analyzing bug reports," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA:

Association for Computing Machinery, 2020, p. 1551–1555. [Online]. Available: https://doi.org/10.1145/3368089.3417928
[46] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, "Decomposed prompting: A modular approach for solving complex tasks," 2023.
[47] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" 2022.
[48] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.
[49] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
[50] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proceedings of the 2022 CHI conference on human factors in computing systems*, 2022, pp. 1–22.
[51] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
[52] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.
[53] D. L. Olson and D. Delen, *Advanced data mining techniques*. Springer Science & Business Media, 2008.
[54] Hustensirup, "Answer to e-mail without scrolling," 2019. [Online]. Available: https://github.com/thunderbird/thunderbird-android/issues/4342
[55] OpenAI, "Hello gpt-4o," May 2024, accessed: 2025-05-29. [Online]. Available: https://openai.com/index/hello-gpt-4o/
[56] Meta, "Llama 4: Benchmarks, api pricing, open source," April 2025, accessed: 2025-05-29. [Online]. Available: https://ai.meta.com/blog/llama-4-multimodal-intelligence
[57] Apidog. (2025) Llama 4: Benchmarks, api pricing, open source. Accessed: 2025-05-30. [Online]. Available: https://apidog.com/blog/llama-4-api/
[58] M. Soltani, A. Panichella, and A. Van Deursen, "A guided genetic algorithm for automated crash reproduction," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 209–220.
[59] L. Plein, W. C. Ouédraogo, J. Klein, and T. F. Bissyandé, "Automatic generation of test cases based on bug reports: a feasibility study with large language models," in *ICSE'23*, 2024.
[60] M. Taromirad and P. Runeson, "A literature survey of assertions in software testing," in *International Conference on Engineering of Computer-Based Systems*. Springer, 2024, pp. 75–96.
[61] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, 2016.
[62] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for guis," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6, pp. 30–39, 2000.
[63] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with gui test automation (will) remain," *Information and Software Technology*, vol. 138, p. 106625, 2021.
[64] Y. Koroglu and A. Sen, "Reinforcement learning-driven test generation for Android GUI applications using formal specifications," *arXiv preprint arXiv:1911.05403*, 2019.
[65] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, and H. Mei, "Supporting oracle construction via static analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 178–189.
[66] Y. Su, Z. Liu, C. Chen, J. Wang, and Q. Wang, "Owleyes-online: A fully automated platform for detecting and localizing UI display issues," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Athens, Greece: Association for Computing Machinery, 2021, p. 1500–1504.
[67] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Nighthawk: Fully automated localizing ui display issues via visual understanding," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.
[68] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, "Don't do that! hunting down visual design smells in complex uis against design guidelines," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 761–772. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00075
[69] D. Zhao, Z. Xing, C. Chen, X. Xu, L. Zhu, G. Li, and J. Wang, "Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1286–1297. [Online]. Available: https://doi.org/10.1145/3377811.3380411
[70] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, "GLIB: Towards automated test oracle for graphically-rich applications," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. Athens, Greece: Association for Computing Machinery, 2021, pp. 1093–1104.
[71] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "Understanding and finding system setting-related defects in Android apps," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 204–215.
[72] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "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, 2014, pp. 183–192.
[73] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "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*, 2024, pp. 1–13.
[74] J. Yoon, R. Feldt, and S. Yoo, "Intent-driven mobile gui testing with autonomous large language model agents," in *ICST'24*, 2024.