A Study of Using Multimodal LLMs for Non-Crash Functional Bug Detection in Android Apps

Bangyan Ju*, Jin Yang*, Tingting Yu†, Tamerlan Abdullayev*, Yuanyuan Wu*, Dingbang Wang†, Yu Zhao*
*University of Cincinnati, USA;
†University of Connecticut, USA;
jubn@mail.uc.edu, yang3j7@mail.uc.edu, tingting.yu@uconn.edu, abdulltn@mail.uc.edu, wu3yy@mail.uc.edu
dingbang.wang@uconn.edu, zhao3y3@ucmail.uc.edu

***Abstract*—Numerous approaches employing various strategies have been developed to test the graphical user interfaces (GUIs) of mobile apps. However, traditional GUI testing techniques, such as random and model-based testing, primarily focus on generating test sequences that excel in achieving high code coverage but often fail to act as effective test oracles for non-crash functional (NCF) bug detection. To tackle these limitations, this study empirically investigates the capability of leveraging large language models (LLMs) to be test oracles to detect NCF bugs in Android apps. Our intuition is that the training corpora of LLMs, encompassing extensive mobile app usage and bug report descriptions, enable them with the domain knowledge relevant to NCF bug detection. We conducted a comprehensive empirical study to explore the effectiveness of LLMs as test oracles for detecting NCF bugs in Android apps on 71 well-documented NCF bugs. The results demonstrated that LLMs achieve a 49% bug detection rate, outperforming existing tools for detecting NCF bugs in Android apps. Additionally, by leveraging LLMs to be test oracles, we successfully detected 24 previously unknown NCF bugs in 64 Android apps, with four of these bugs being confirmed or fixed. However, we also identified limitations of LLMs, primarily related to performance degradation, inherent randomness, and false positives. Our study highlights the potential of leveraging LLMs as test oracles for Android NCF bug detection and suggests directions for future research.**

***Index Terms*—Mobile Testing, Large Language Model, Testing Oracle, Non-Crash Functional Bug**

I. Introduction

Mobile applications (apps) have witnessed a surge in popularity, with the Google Play app store hosting approximately three million applications [1]. A pivotal study revealed that a significant majority (88%) of app users are likely to abandon an application if they consistently encounter functional problems [2]. Indeed, apps that fail to function properly can have significant real-life consequences for users. This motivates developers to quickly identify and resolve issues, or risk losing users. Many automated GUI testing approaches for mobile apps have been proposed, such as random testing [3], [4], model-based testing [5]–[7], and learning based testing [8]–[11]. The goal of these automated test generators is to generate event sequences to achieve high code coverage and/or detect crashes. However, these tools mainly concentrate on designing test sequences instead of test oracles [12] to verify the presence of bugs. As a result, these works typically identify only those bugs that cause a system to crash, which are evident from crash logs, rather than more subtle non-crash functional (NCF) bugs. The absence of mobile-specific testing oracles presents a significant challenge in detecting NCF bugs [13].

Some specialized tools have been developed to detect specific types of NCF bugs in limited scopes. For example, DiffDroid [14] is a technique that automatically detects cross-platform inconsistencies in mobile apps. iFixDataloss [15] can detect data loss issues in Android apps. SetDroid [16] can detect setting-related issues. All current methods are derived from limited observations and based on pre-defined heuristic rules (e.g., differential analysis [17]) to detect a specific type of NCF bug with a dedicated test sequence that supports differential analysis. They can not generalize to a wide category of NCF bugs, accommodate general test sequences, and assure effectiveness. A recent study [18] revealed that of 399 crawled NCF bugs from Github [19], only 84 fall within the detection scope of seven of state-of-art existing tools [14]–[16], [20]–[23], which identified merely two of them in total. Consequently, it is necessary to generate test oracles to detect NCF bugs with high accuracy in diverse categories.

Our intuition in this study is that leveraging Multimodal Large Language Models (LLMs) as test oracles could extend the scope of detectable NCF bugs beyond the capabilities of existing tools and increase the successful detection rate. Multimodal LLMs [24], [25], such as GPT-4o [26] have significantly enhanced capabilities in natural language understanding, image processing, and question answering. By leveraging extensive, unlabeled text corpora and images for self-supervised learning, LLMs develop a deep reservoir of domain knowledge.

In this paper, we conduct an empirical study on using multimodal Large Language Models (LLMs) as test oracles for detecting NCF bugs. Different from a recent study [18] that focuses on the causes of NCF functional bugs and the performance of existing tools, our research is dedicated to assessing the performance of LLMs as test oracles. We also provide insights and suggestions on the advantages and limitations of LLMs for future research. We have formulated four research questions to guide our study:

RQ1: How effective and efficient are LLMs to be NCF bug test oracles compared to state-of-the-art bug detection tools?
RQ2: What roles do two-prompt strategy, in-context learning, and rules in prompt play in the effectiveness of LLMs?

Corresponding author: Yu Zhao (email: zhao3y3@ucmail.uc.edu).

2024 31st Asia-Pacific Software Engineering Conference (APSEC) | 979-8-3315-3401-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/APSEC65559.2024.00017

2640-0715/24/$31.00 ©2024 IEEE
DOI 10.1109/APSEC65559.2024.00017

RQ3: How do different models of LLMs, such as GPT and Gemini, perform in detecting NCF bugs?
RQ4: What is the usability of using LLMs for detecting real-world NCF bugs with random test sequence generation?

To comprehensively study the capability of leveraging LLMs as test oracles for detecting NCF bugs in Android, we propose OLLM, an attempt to leverage multimodal LLMs to be test oracles in NCF bug detection. OLLM incorporates fundamental LLM mechanisms, including prompts and in-context learning [27]. The input of OLLM is a given test sequence represented by a sequence of GUI events and an APK of the app under test. As a proof-of-concept, we assume the test sequences are already generated by test generation approaches [5]–[11], e.g, a random test sequence generation method used in section V-E. OLLM includes three phases. In the first phase, OLLM focuses on the collection of execution data during GUI testing. This involves monitoring and documenting essential information in the runtime testing, including the execution of events in the test sequence and the corresponding Execution Results (ERs). The ERs are represented as layout information that incorporates textual data and accompanying screenshot images supported by multimodal LLMs. The second phase involves designing two specific prompts as inputs for the LLM. One prompt is aimed at detecting UI logic errors using textual layout information, while the other prompt targets the identification of UI display issues by leveraging screenshot data. In the third phase, OLLM employs the LLM to execute two prompts sequentially and integrates in-context learning strategies to enhance detection accuracy. Then OLLM combines the insights from the two prompts to determine the presence of NCF bugs.

Utilizing GPT-4o as the LLM, our study results demonstrate that OLLM has a broad NCF bug detection scope, covering all 71 well-documented Android NCF bugs in our dataset. In comparison, six state-of-the-art methods OwlEye [23], iFixDataloss [15], SetDroid [16], Genie [20], Odin [21], and ITDroid [22] cover only 17, 2, 1, 2, 11, and 3 bugs, respectively. OLLM effectively identified 35 (49%) of the NCF bugs and provided accurate descriptions of the detected bugs, significantly outperforming these existing tools. During applying LLMs, fundamental mechanisms, such as the two prompt strategy, in-context learning, and rule-based prompts, play significant roles in enhancing NCF bug detection. For instance, in-context learning alone increased the detection rate from 40% to 49%. By using five different models of LLMs, we observed significant performance differences among them. Our study was further extended to include a new dataset of 64 Android apps, in which OLLM successfully uncovered 24 previously unrecognized bugs, four of which have been confirmed or fixed. Despite these successes, we also identified limitations of LLMs, primarily related to performance degradation, inherent randomness, and false positives. Our study underscores the potential of LLMs as test oracles for identifying NCF bugs and highlights areas for future research.

The contributions of this paper are as follows:

- Our empirical study comprehensively explores the performance of leveraging LLMs to be test oracles. The evaluation results demonstrate that LLMs are effective in detecting Android NCF bugs while also revealing certain limitations, and suggesting directions for future research.
- Our research pioneers a novel attempt OLLM by leveraging LLMs as test oracles for NCF bug detection in Android apps.
- OLLM along with all experiment data are publicly available [28].

II. Preliminaries and Motivation

A. Preliminaries

Test sequences for mobile applications typically consist of sequences of Graphical User Interface (GUI) events[1], designed to simulate user interactions with the apps. After executing an event, the application exhibits certain behaviors as Execution Results (ERs), such as navigating to a different page or displaying a pop-up dialog. A proficient test sequence effectively engages the appropriate GUI widget on the app screen and detects bugs that include crash and NCF bugs. Recent study [18] shows 65.4% of bugs are categorized as NCF bugs. In this work, we define NCF bugs as software issues that deviate from the expected functionalities of an operational app, without leading to an observable crash of the app, aligning with existing research [18], [20], [21].

An illustration of a test sequence of Android bug Amaze-2113 [29] is depicted in Fig.1, which showcases a process of renaming a photo. In this scenario, a user attempts to rename a photo to "Messi 19 99.jpg". However, upon saving, the photo's name is incorrectly displayed as "messi%2019%2099.jpg", deviating from the expected "messi 19 99.jpg", thereby revealing an NCF bug. In the depicted test sequence, "Action" refers to the user interactions with the app, such as clicking and entering data. "Execution Results" (ERs) display the textual resulting layouts after an action is performed.

B. Observation

We studied hundreds of GitHub Android bug reports and drew on summarized categories based on empirical studies in existing works [15], [18], [22] to understand the characteristics of NCF bugs for guiding the design of OLLM to detect these bugs. The data was obtained from github and we summarized and categorized the NCF bug reports based on our own observation and conclusions from existing study. In general, NCF bugs can be grouped by UI logical bugs and UI display bugs that have been studied by existing work [18]. **UI display bugs** can be categorized as follows: (1) Content-related issues (C.): the UI displays correctly but the content text does not make sense, such as truncated sentences; (2) Missing UI elements (M.): a UI component unexpectedly disappears; (3) UI distortion (UD.): displaying issues such as overlapping elements or blurry images; (4) Redundant UI elements (R.): UI components are duplicated. Similarly, **UI logical bugs** include:

[1]In our setting, an event refers to an executable GUI widget associated with an action type (e.g., click, scroll, edit, swipe, etc). An action is defined as the execution of an event.

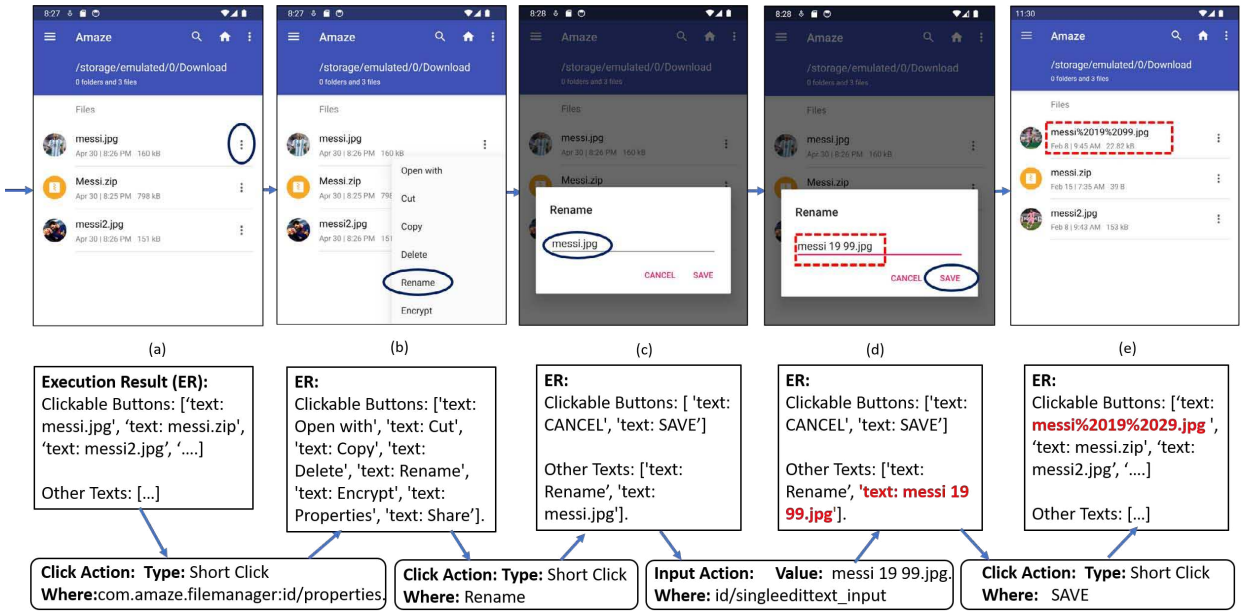

Fig. 1: Motivation Example

(1) Incorrect interaction (I.): the app's subsequent behavior deviates from expectations, such as clicking a button named "category" redirects to the home screen; (2) Functionality does not take effect (F.): the user successfully interacts, but the function doesn't work; (3) UI element does not react (UR.): components do not react to user inputs, such as no reaction after clicking a button; (4) Data loss (D.): the data entered by the user is unexpectedly cleared; (5) Language related (L.): language issues in the international version of the app, such as unexpected language changes.

C. Comparison with existing works

Some existing works focus on detecting NCF bugs, but all of them only focus on specific types of bugs rather than general NCF bugs. Several existing studies involve performing differential analyses of app statuses using dedicated test sequences. ITDroid [22] can automate the detection of internationalization (i18n) issues, by doing differential testing using the original app and the internationalized one. SetDroid [30] is designed to identify issues related to user-configurable settings (e.g., network, location, permissions) that apps may fail to adapt properly. iFixDataloss [15] executes specific events, such as screen rotation, and then analyzes the differences in data on the layout before and after the event to identify data loss. DiffDroid [14] can automatically identify cross-platform inconsistencies in mobile applications. GENIE [20] leverages a differential analysis that utilizes a differential analysis to identify NCF bugs, which are characterized by changes in one part of an app that adversely affect other parts. ODIN [21] utilizes differential analysis to identify the abnormal behavior of one pair of GUI layouts compared to other pairs. The tools discussed are specifically designed to address a narrowly defined range of bug categories, relying on pre-defined heuristic rules based on limited observations. Additionally, they depend on dedicated test sequences that facilitate differential analysis, such as running two test sequences on different devices for DiffDroid. These limitations restrict their generalizability and effectiveness. Owleye [23] employs deep learning to model the visual information from GUI screenshots, focusing on identifying UI display issues such as text overlap and blurred screens. All these tools also lack the natural language understanding capacity to understand the abnormal behavior of apps. As noted in the introduction, a recent study [18] reveals that only a small percentage of NCF bugs fall within their scope, with only about 0.5% of these bugs being detectable. None of the existing tools are capable of successfully detecting the NCF bug illustrated in Fig. 1, which necessitates understanding the behavior of the text "rename" both theoretically and practically. By incorporating extensive and in-depth domain knowledge, LLMs can detect a broader range of NCF bugs with heightened precision.

III. Proposed LLMs based Test Oracle

As an attempt to leverage LLMs to be test oracles for NCF bug detection in this study, OLLM should incorporate multiple essential components that have proven effective in several existing works [31]–[33]. These components include data collection, prompt design, and in-context learning. The overarching structure of OLLM encompasses three main phases: information extraction, prompt generation, and prompt execution.

A. Information Extraction from Test Sequence

OLLM extracts both action information and corresponding ERs from the captured screen data of the runtime Android app undergoing testing sequences. The ERs encompass textual information and screenshot images from the layout supported by multimodal LLMs.

The initial step of information extraction involves capturing runtime information during the execution of the test sequence. OLLM dumps GUI layout information from an app following a series of user actions. The current version of OLLM supports six types of UI actions: Click, Long Click, Drag, Swipe, and Input, as well as four system actions: Back, Home, Enter, and Rotate. During each testing iteration within a given test sequence, executing events such as "Click Menu", "Click Rename", "Input messi 19 99.jpg", and "Click Save" as illustrated in Fig.1 trigger updates in the GUI pages, reflecting execution results (ERs) of the event execution. OLLM employs UIAutomator [34] to perform these actions and then dump the textual screen layout and screenshot as the output of the corresponding action.

TABLE I: Textual Extraction Results from (d) to (e) in Fig.1

Steps	Text Extraction
(d)	**Action**: Input 'id/singleedittext_input' with 'messi 19 99.jpg' **Clickable Buttons**: ['content-desc: CANCEL', 'content-desc: SAVE'] **Other Texts**: ['content-desc: Rename', 'content-desc: messi 19 99.jpg']
(e)	**Action**: Click 'Save' **Clickable Buttons**: ['content-desc: messi%2019%2099.jpg', 'content-desc: messi.zip', 'content-desc: messi2.jpg', '...'] **Other Texts**: ['content-desc: Amaze','content-desc: /Storage/emulated/0/Download', ..]

To effectively represent the ERs gleaned from the dumped information, OLLM extracts essential data elements from the data dump of text layout including the text information on clickable buttons, long clickable buttons, checkable elements, and other texts. These other texts are defined as unexecutable text of the GUI, including labels, headings, and informational text. OLLM organizes the extracted textual information from the test sequence into a structured format that includes actions and corresponding textual ERs. Table I shows an example of the extracted textual information from steps (d) and (e) in the test sequence depicted in Fig. 1.

In addition to text-based information extraction, OLLM enhances its data extraction process by incorporating screenshot images corresponding to the action as an additional component of ERs. The screenshots are essential for enhancing the UI display bugs. Providing a visual clue is helpful for LLM to analyze certain kind of bugs. These screenshot images can enhance the detection of both UI display and UI logic bugs, which will be discussed in the next section.

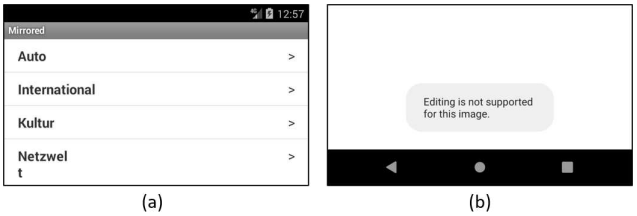

Fig. 2: Screenshots for Bug Detection

B. Two Prompt Generators

The second phase involves generating prompts that utilize information extracted from the execution of the test sequence. In the current version of OLLM, we have adopted the concept of Decomposed Prompting [35], which breaks down a complex prompt task into simpler sub-tasks. This methodology optimizes each prompt for its respective subtasks. We have designed and implemented two prompt generators, one for UI logic bugs and one for UI display bugs, as detailed in Table II. The first prompt (prompt 1) leverages only textual information from ERs and actions tailored for general NCF bugs. It can particularly detect logical bugs that can be identified through text, such as incorrect actions, calculations, decisions, or data processing. An example of an NCF bug targeted by prompt 1 is the text inconsistency bug shown in Fig. 1. Information from actions and ERs from steps (a) through (d) provides essential context for understanding the entire scenario of the test sequence. All this information has been extracted from texts of ERs and actions as detailed in section III-A.

The second prompt (prompt 2) incorporates both captured screenshot images and textual information from ERs and actions to target on UI display bug. For example, a UI display bug illustrated in Fig.2(a) from the app Mirrored [36] shows the text "Netzwelt" being cut off at the bottom, with only the "t" visible on the following line. Additionally, inconsistencies between dumped text information and UI display also need screenshot information to detect, as seen in the Android bug AnkiDroid-7232 [37] depicted in Fig. 2(b). The message "Editing is not supported for this image" can only be captured through screenshot images rather than text dumps.

Our prompt incorporates LLMs with 4 detailed instruction sections "Question", "Rules", "Test sequence", and "Output format" in prompts to set clear expectations for the LLM's responses when detecting potential NCF bugs within an app. The Question section defines the focus for each prompt and claims the expectations in the LLM's response as if there is an NCF bug. The Rule section includes some constraints and hints that the LLM needs to follow. These rules are designed to minimize the false positives as the fake alarm. These constraints and hints are generated based on our extensive analysis of hundreds of correct sequences and common fake alarms made by LLMs. Detailed documentation of the 10 rules along with 4 instruction sections is available in our GitHub repository [28].

Additionally, the extracted information in the test sequence, outlined in Section III-A, forms a crucial part of the prompt as the third section of the Test Sequence. Specifically, to detect bugs that require visual cues in prompt 2, OLLM integrates text extraction results with a screenshot of the final page. The decision to use only the last page's screenshot is due to the significant time and resource demands of image processing by these multimodal LLMs. Since OLLM already incorporates test sequence information in text, including all screenshots is unnecessary. Using the unique image significantly reduces processing time. Lastly, the Output Format section is established to clarify the expected response format. TABLE II shows the general structure of the 2 prompts. The prompt represents our input to LLM.

TABLE II: Prompt Structure

Prompt Structure		
Category	Prompt 1	Prompt 2
Question	You will read an event flow, containing page text and actions to reach other pages. Then I will ask you: Is there any logical error, or a bug in the output after the given test sequence?	You will read an Android app event sequence, containing actions to reach each page and the text on each page. Then I will send you a screenshot of the last page. Your task is to determine whether there is a UI error in the screenshot.
Rules	While evaluating logic errors, you should also consider the rules below: (1) You should not analyze battery ...	If you detect error messages, please also check the consistency or correctness of these elements about the provided sequence. ...
Test sequence	Text Extraction Result in Table I	Text Extraction Result in Table I + Screenshot of the Last Page
Output format	Provide your answer with yes or no. If your answer is yes, please also provide the reason. ...	Provide your answer with yes or no. If your answer is yes, please also provide the reason. ...

C. Prompt Execution

In the third phase, prompts are executed in the LLM to detect NCF bugs in responses. Since LLMs are typically not pre-trained for specific tasks like NCF bug detection, this may result in a low detection rate. To address this, recent research [27] highlights the LLMs' ability to acquire new skills by learning from a few examples provided in the prompt, a process known as in-context learning. OLLM employs in-context learning to enhance the effectiveness of NCF bug identification by providing question-answer pairs specifically curated from typical functional bug scenarios. In this study, these examples are randomly selected K-pairs from the corresponding group (logical or display bug) relevant to the prompt. We empirically set K to 3, consistent with the settings of recent work [38].

OLLM then aggregates responses from the two prompts to make a final decision regarding the presence of NCF bugs, a strategy we refer to as the integrated result of two prompts. To enhance the true positive of NCF bug detection, we have implemented a heuristic rule of the integrated result of two prompts: if either prompt suggests the presence of a bug, an NCF bug exists. Otherwise, there is no bug.

IV. Study Design

In this empirical study, we aim to study the comprehensive capability of LLMs to be NCF bug test oracles. Along with four research questions mentioned in section I, we also discuss the observed limitations of using LLMs, such as performance degradation, inherent randomness, and high false positive rates, and suggest directions for future research.

A. Datasets

To assess our methodology in RQ1, RQ2, and RQ3, it is crucial to establish datasets for evaluation. An existing study [18] has constructed a dataset of NCF bugs sourced from GitHub with well-documented bug reports, employing a standard selection procedure. This dataset includes 399 NCF bugs and is aimed at evaluating current methodologies. Out of these, we retained 71 bugs identified through manual examination by our two graduate students as genuine NCF issues satisfying the definition in section II-A. Bugs excluded from our evaluation include those with unavailable APKs, unclear reports, irreproducibility, or outdated apps. Additionally, requests for functional design improvements, like the date format request in WordPress-15026 [39], are not considered bugs.

To further validate the usability of OLLM to detect unaware NCF bugs in RQ4, we applied it to a widely used app dataset that serves as the benchmark for various Android testing tools [40], [41]. This dataset comprises 68 apps spanning 18 domains [42]. We excluded four apps that consistently crashed immediately upon launch in our Intel Atom emulator. OLLM's objective differs from the existing tools that use this dataset primarily to identify crash bugs and enhance code coverage.

B. Evaluation Setup

We conducted our experiment on a physical x86 machine equipped with an Intel i5 CPU and running Ubuntu 20.04. We primarily focused our investigation on GPT-4o [43], the most recent cutting-edge large language model from OpenAI to facilitate bug detection responses.

1) RQ1: Effectiveness and Efficiency of OLLM: To address Research Question 1, we assess OLLM and six other Android functional detection tools (OwlEye [23], iFixDataloss [15], SetDroid [16], Genie [20], Odin [21], and ITDroid [22]) in detecting NCF bugs and testing scope. We exclude DiffDroid [14] from this comparison as it focuses on compatibility bugs, which are outside OLLM's scope. Since OLLM does not generate test sequences, our analysis focuses solely on the effectiveness of test oracles using pre-defined test sequences. These test sequences used in this study are based on bug reports analyzed manually generated by two graduate students, ensuring they reach the bugs. We directly compare OwlEye and iFixDataloss with OLLM using the same test sequences. However, the test oracles in SetDroid, Genie, Odin, and ITDroid require two specifically designed sequences to expose differences in application status and cannot operate on our manually generated sequences. To compare with these tools, we extract the testing scope of these tools by reviewing relevant papers and checking if their test oracles could detect the NCF bugs in our dataset, assuming these tools generated the correct inputs to trigger these bugs. Then we use the number of NCF bugs within the test scope of SetDroid, Genie, Odin, and ITDroid to represent the optimistic upper bound of their bug detection capability. An existing study [18] has already measured the true positive rate of these tools on the same dataset, finding only 0.5% of NCF bugs were detected, suggesting no need to reevaluate their overall performance.

False positives (FP) represent false alarms that waste developers' time during verification. To fairly assess false positives, we generate bug-free test sequences that closely resemble those used to evaluate true positives capable of reaching the bugs by removing the final step that triggered bugs. OLLM was then tested on these modified sequences to determine its accuracy in confirming the absence of bugs. Furthermore, to enhance our evaluation beyond merely confirming bug presence, we introduced a metric called the True Positive Correct (TPC) rate that captures both the successful identification of bugs and the correctness of their descriptions as specified in the bug documentation. In the following text, "detected bug by OLLM" indicates that the bug meets the TPC metric criteria. In this study, we calculate the TPC rate by dividing the number of true positive correct results by the total number of actual positives. Conversely, the FP rate is calculated as the ratio of the number of negative events wrongly categorized as positive to the total number of actual negative cases.

We evaluate OLLM across configurations to assess its overall performance including the default setting of querying once with the integrated result of two prompts (detailed in section III-C). We evaluate OLLM in two modes: querying once (q1) and querying three times (q3) using the same prompt. The q3 mode is employed to thoroughly examine the LLM's performance by minimizing its randomness, as it offers OLLM three opportunities to provide the correct response. If at least one TPC appears in the results of the three queries, we regard it as a successful TPC detection under q3 mode. Conversely, if at least one FP occurs in a bug-free sequence, we categorize it as an FP alarm. For each mode, we separately repeat the settings using only prompt 1, only prompt 2, and the integrated result of two prompts, totaling six settings.

2) RQ2: Roles of two-prompt strategy, in-context Learning, and rules in prompt within OLLM, we investigate how different components influence its effectiveness and efficiency for RQ2. To do this, we repeated the experiment with the two-prompt strategy, in-context learning, prompt rules components disabled individually, then reevaluated OLLM's performance under q1 mode. The performance of prompt 1 and prompt 2 are analyzed exclusively. OLLM-PA merges and consolidates all unduplicated contents from prompts 1 and 2 by disabling the prompt decomposition strategy outlined in section III-B. OLLM-PA is then used to analyze the case presented by a single prompt in a complex task. OLLM-NoCon is utilized to assess OLLM by disabling in-context learning, as described in Section III-C. OLLM-NoRule is used to evaluate the performance impact of disabling the "Rules" section, which serves as constraints and guidance in the prompts, as outlined in Table II.

3) RQ3: Performance of Different Models of LLMs: We assess OLLM across two leading multimodal large language models that facilitate image processing, including Gemini [44], and ChatGLM-4 [45], along with other GPT versions (GPT-3.5, 4 [43]), to assess their capabilities under q1 mode.

4) RQ4: Usability on real-world previously-unaware apps: To rigorously assess OLLM's effectiveness in identifying new bugs in real-world applications, we developed an automated test sequence generator utilizing a semi-random strategy. Unlike Monkey [3], which employs a fully random strategy to select events, our method randomly selects one of the least frequently visited events to execute on the current page of the app. This strategy, inspired by Stoat [41], aims to increase code coverage. Each selected event in the test sequence is executed as an action and dumped layout information is recorded, which OLLM then uses to detect NCF bugs. The testing duration is set to one hour per app, aligning with the benchmarks set by other Android app testing tools [9], [10], [40], [46].

V. Results and Analysis

A. RQ1: Effectiveness and Efficiency of detecting NCF bugs

Table III presents the results of NCF bug detection by OLLM across 71 bugs in 6 different settings. In the 'Total' and 'Percentage' rows, values before the parentheses represent results under the single-query mode (q1), and those within parentheses correspond to the three-query mode (q3). By using the default setting of the integrated result of two prompts under q1, OLLM achieves a TPC rate of 49% reflecting the accurate identifying of actual NCF bugs, and a false positive (FP) rate of 59% indicating the rate of false alarm. The setting with the lowest FP rate is under mode q1 with unique prompt 1: TPC rate=42% and FP rate= 29%. This setting is ideal for test oracles prioritizing bug-free test results while maintaining a reasonable NCF bug detection rate. Conversely, the setting with the highest TPC rate involves three queries with the integrated result from two prompts: TPC rate =63% and FP rate=93%. This setting is suitable for test oracles that aim to reveal as many NCF bugs as possible but may suffer from a high FP rate. These results underscore OLLM's capability to successfully detect NCF bugs with accurate descriptions of the detected NCF bugs.

OLLM and the six existing works are designed to detect NCF bugs. However, due to the limitations of heuristic-based techniques of existing works, OLLM achieves much broader bug detection scopes as discussed in section II-C. With pre-trained in-depth domain knowledge of LLM and generated prompts, OLLM targets a wider array of NCF bug categories. Currently, we are not aware of any of the 71 NCF bugs in our dataset that fall outside OLLM's detection scope. In contrast, most bugs fall outside the designated scopes of existing tools. OWLEYE, IFixDataloss, ITDroid, SetDroid, Genie, and Odin cover only 17, 2, 3, 1, 2, and 11 bugs respectively. This limitation is a primary reason for their inability to detect these bugs. For example, in our experiments with the same dataset, OLLM correctly identified and described 38 NCF bugs, whereas OwlEye and iFixDataloss detected only 4 and none, respectively. ITDroid, SetDroid, Genie, and Odin were able to detect up to 3, 1, 2, and 11 bugs, respectively.

The exceptional TPC rate of OLLM is attributed to its proficiency in semantic understanding of text via LLMs. Analysis of 71 bugs in the dataset revealed that 66 bugs heavily depend on comprehending the actions and layout texts within the test sequence to detect the bug. The average

TABLE III: Test Results of OLLM

Bugs	TC	Cat.	Prompt 1		Prompt 2		Integrated result	
			TPC	FP	TPC	FP	TPC	FP
AnkiDroid-7465	2	I.		✓1		1		✓2
AnkiDroid-7232	2	F.	✓2		✓3	1	✓5	1
AnkiDroid-7836	4	UD.		✓1		✓3		✓4
AnkiDroid-7801	4	F.	✓1				✓1	
AnkiDroid-7793	2	F.		1		1		2
AnkiDroid-7768	6	F.	1		2	✓1	3	✓1
AnkiDroid-7730	7	I.				✓2		✓2
AnkiDroid-7674	2	F.	✓2	1		✓2	✓2	✓3
AnkiDroid-6288	2	F.	✓3	1	✓3		✓6	1
AnkiDroid-7377	4	UD.		✓2		✓1		✓3
AnkiDroid-7070	4	F.	2			✓2	2	✓2
AnkiDroid-7027	4	R.		1	✓3	2	✓3	3
AnkiDroid-6887	8	I.		2		✓2		✓4
AnkiDroid-6894	3	I.	1			1	1	1
AnkiDroid-5688	2	I.	1	1		✓1	1	✓2
AnkiDroid-5091	4	I.	✓3	1		✓1	✓3	✓2
AnkiDroid-5167	4	F.		1		1		2
AnkiDroid-4935	0	I.	✓1			1	✓1	1
AnkiDroid-8975	2	F.				1		1
AnkiDroid-8379	3	I.	✓2	✓1		✓1	✓2	✓2
AnkiDroid-7023	1	F.		✓1		1		✓2
AntennaPod-4776	2	I.	✓3	1	✓2	✓3	✓5	✓4
AntennaPod-3786	2	I.	✓1	1			✓1	1
AntennaPod-2992	0	M.		✓1	1	✓3	1	✓4
AntennaPod-4548	2	I.	✓3	1		2	✓3	3
Firefox-3617	2	F.	1	1		1	1	2
Firefox-3152	0	D.	✓1				✓1	
Firefox-3291	3	I.		1		1		2
Firefox-4068	3	C.						
Firefox-3146	4	L.	✓2	✓1	✓2	2	✓4	✓3
Firefox-3254	2	F.	✓2	✓2	✓3		✓5	✓2
Firefox-3121	1	F.		✓2		✓1		✓3
Firefox-3304	1	C.		1				1
Firefox-3297	2	D.				1		1
Simplenote-1294	3	F.	2	1		✓2	2	✓3
Simplenote-1190	1	F.	2		✓2	✓1	✓4	✓1
Simplenote-1111	3	I.	✓3	✓3	✓3		✓6	✓3
Simplenote-1046	2	UD.						
Simplenote-984	1	UD.	✓3			✓2	✓3	✓2
Simplenote-952	1	UD.		✓1	2	✓2	2	✓3
Simplenote-623	0	I.		1		2		3
AnkiDroid-7758	5	L.		1	1	✓3	1	✓4
AnkiDroid-6857	0	I.				2		2
AnkiDroid-7366	1	UD.			✓3		✓3	
AnkiDroid-6587	9	I.	✓3	✓2	✓3	✓1	✓6	✓3
AnkiDroid-6119	2	I.				1		1
AnkiDroid-5334	5	I.	✓3	✓2	✓3	✓2	✓6	✓4
AnkiDroid-5156	2	M.				✓2		✓2
AnkiDroid-5105	1	UD.				2		2
AnkiDroid-4999	3	I.	✓1			✓1	✓1	✓1
AnkiDroid-8072	5	F.	✓3	✓1	✓3	✓1	✓6	✓2
AnkiDroid-8466	2	F.		✓1		✓2		✓3
AnkiDroid-8547	3	I.	✓1			✓2	✓1	✓2
AnkiDroid-7896	3	C.		✓1	✓3	2	✓3	✓3
Amaze-2113	2	I.	✓3	✓1	✓3	✓1	✓6	✓2
Amaze-1919	1	I.	✓3		✓3	✓2	✓6	✓2
Amaze-1916	3	I.	2	1	✓3	1	✓5	2
Amaze-1872	3	UR.	✓3	✓1		1	✓3	✓2
Amaze-1834	3	I.	✓3			1	✓3	1
Amaze-1797	2	I.	✓3	2	2	2	✓5	4
Amaze-1712	7	F.	✓3			1	✓3	1
Amaze-1628	2	F.	✓2			2	✓2	2
NewPipe-5363	4	I.		1	2	1	2	2
NewPipe-6409	1	C.				✓2		✓2
NewPipe-4113	3	M.				✓3		✓3
NewPipe-6397	2	I.	✓2	✓1	✓3	2	✓5	✓3
WordPress-14234	4	I.		1		✓3		✓4
WordPress-13671	2	I.		✓1		✓3		✓4
WordPress-13121	3	UD.				✓3		✓3
WordPress-9966	3	I.	✓3	✓1		✓2	✓3	✓3
WordPress-8755	4	M.	✓1	1	✓3	✓3	✓4	✓4
Total: 71	2.7		30(38)	21(43)	19(24)	34(61)	35(45)	42(66)
Percentage:			42% (53%)	29% (61%)	26% (34%)	48% (86%)	49% (63%)	59% (93%)

Integrated result= The integrated result of two prompts TC= number of texts required for comprehension to detect the bug. Cat.= category, TPC= true positive correct, FP= false positive, ✓= TPC/FP in first time query (q1), Number n in TPC/FP column = number of TPC/FP in 3 queries (q3), M(N)= number of first time query(number of at least one occurrence in 3 queries (q3))

number of instances requiring text semantic understanding for bug detection in these test sequences is 2.7 as shown in the TC column of Table III. As illustrated by the example in Fig. 1, the essential text comprehension of "Files", "Rename" and "Save" provide OLLM with sufficient information for logical reasoning and comparing two text strings: "messi 19 99.jpg" and "messi%2019%2029.jpg," thereby enabling OLLM to detect NCF bugs. In contrast, previous approaches [14]–[16], [20]–[22] relying solely on pre-defined heuristic rules and differential comparison. These methods depended entirely on specific differential analysis-based test sequences that could reveal differences in the status of Android apps for comparison. Without leveraging NLP techniques these methods fail to identify general functional issues.

TABLE IV: Multiple LLMS

	Prompt1		Prompt2		Integrated result	
	TPC	FP	TPC	FP	TPC	FP
GPT-4o	30 (42%)	21 (29%)	19 (26%)	34 (48%)	35 (49%)	42 (59%)
GPT-4	27 (38%)	56 (79%)	23 (32%)	48 (68%)	37 (52%)	63 (89%)
GPT-3.5	0 (0%)	45 (63%)	-	-	0 (0%)	45 (63%)
Gemini	21 (30%)	12 (17%)	6 (8%)	44 (62%)	22 (31%)	48 (68%)
ChatGLM-4	9 (13%)	42 (59%)	10 (14%)	14 (20%)	16 (23%)	48 (68%)

B. RQ2: Effectiveness and Efficiency of Two-prompt strategy, In-context Learning, and Rules in Prompt

Within OLLM, we assess the performance of two prompts separately. The results indicate that prompt 1 achieved a 42% TPC rate and a 29% FP rate, whereas prompt 2 achieved a 26% TPC rate and a 48% FP rate. Prompt 1 successfully detects 62% more NCF bugs and 40% fewer mistakes in incorrectly identifying bug-free sequences as buggy compared to prompt 2, indicating prompt 2 is distracted by additional screenshot image information with text and overlooks some detailed text. Additionally, we find the average processing time of prompt 2 is 7.5 seconds which is much higher than prompt 1 which is 3 seconds. However, prompt 2 is able to detect 5 bugs that prompt 1 cannot, with three of these bugs falling into the UI display bug group as in section II-B, including one instance of redundant UI elements, one instance of UI distortion, and one instance of content related issue. These findings confirm our expectation that Prompt 2, with its visual cues, is particularly effective at detecting specific NCF bugs, especially those related to UI display. OLLM-PA with a 35% TPC rate performs worse than OLLM, which has a 49% TPC rate under the same setting, though both share an FP rate of 59%. This highlights that managing complex tasks with a single prompt is less effective than employing a two-prompt decomposed approach in detecting NCF bugs.

By removing the in-context learning, the results reveal that OLLM-NoCon exhibited a lower TPC rate of 40%, compared to the 49% TPC rate observed with OLLM equipped with in-context learning. These findings suggest that the inclusion of in-context learning in OLLM plays a critical role in increasing the TPC rate. The results reveal that OLLM-NoRule on prompt 1 exhibited a significantly high FP rate of 93%, substantially greater than the 29% observed with the original OLLM on prompt 1 under mode q1. These findings suggest that the inclusion of rules in OLLM plays a critical role in reducing the FP rates.

C. RQ3: Performance of Multiple LLMs

Overall, with reference to Table IV, we find GPT-4 achieves the highest TPC rate at 52% but it suffers the highest FP rate

at 89%. GPT-4o achieves a slightly lower TPC rate at 49% but has the lowest FP rate at 42%. However, GPT 3.5 and ChatGLM-4 exhibit significantly lower TPC rates of 0% and 16%. GPT-3.5's TPC rate is 0% because all detected bugs have incorrect bug descriptions, violating the definition of TPC. Additionally, GPT-3.5 cannot query on prompt 2 as it does not support image input. This study highlights the substantial performance differences among various LLMs when used as test oracles for NCF bug detection. Based on our findings, we recommend utilizing GPT-4o due to its strong TPC rate and lower FP rate.

D. RQ4: Real World NCF bug Detection with Test Sequence Generation

OLLM effectively detected 24 NCF bugs in five bug categories across 64 Android applications included in the dataset. These bugs are unaware by the authors and not reported by literature [40], [41], [47] using the same dataset. One graduate student manually verifies the bug reported by OLLM to ensure these bugs satisfy the NCF bug definition in section II-A. Two issues have been confirmed and three of them have been resolved. In this context, "confirmed" means that the bug has been reported and acknowledged by the developer, while "fixed" refers to bugs that have been resolved by developers in subsequent versions. Table V illustrates a selection of the NCF bugs detected by OLLM, including the names of the affected apps, the categories of the bugs, and brief descriptions of each issue. OLLM demonstrates the ability to detect NCF bugs across various categories by simply generating test sequences randomly. In addition to the successful detection rate, we also calculate the FP rate with randomly picked 25 randomly generated bug-free sequences in the dataset. The FP rates for prompt 1 and prompt 2 are 32% and 52%. Detailed reports of these findings are accessible in our experimental dataset [28].

E. Limitations in LLMs and Future Research Direction

The most significant limitation observed in this study is the performance degradation of online business LLMs over time. Our study conducted in May 2024, using GPT-4 achieved a TPC rate of 43.7% with old version of prompt 1 of OLLM. We repeat the study with the same prompt and setting in July 2024, the result shows the TPC rate plummeted to 1%. Due to this serious performance degradation, we were compelled to modify our prompt design and incorporate an in-context learning strategy to achieve an acceptable detection rate, as demonstrated in this study. This severe instability in LLM performance could significantly impact the practical application of LLMs as test oracles. A potential solution is to design an adaptive learning system for prompt generation and in-context learning that can utilize machine learning to adjust its settings corresponding the current performance of the LLM. In future work, we will use Fine-tuning or RAG to improve the performance of LLM.

We also observed significant randomness in the performance of LLMs. As indicated in Table III, the TPCs of prompt 1 and prompt 2 under mode q3 are 26% and 31% higher than the TPCs detected in the first query. The increase in TPCs with repeated queries highlights the randomness in LLM responses. In 38 TPC cases under q3 in prompt 1, 22 cases missed at least one successful detection across the three queries. This level of randomness results in a significant time cost in manually verifying results, as the initial query may not yield an acceptable outcome due to this variability. A potential solution involves leveraging the BLEU metric [48] to identify common elements in the responses across repeated queries. The most frequent of these common elements in multiple queries can then represent the prime decision of LLMs and be used as the bug detection result.

TABLE V: Selected Real World NCF Bug Detection

App Name	App Category	Bug Cat.	Bug Description	Status
Adsdroid	Reference	F.	Unable to fetch any results.	**Confirmed**+Fixed
Mileage	Utility	C.	Displays unrealistic values.	**Confirmed**
Wikipedia	Reference	F.	Pages cannot be saved.	Fixed
Anymemo	Educational	F.	Link directs to invalid site.	Fixed
Anymemo	Educational	I.	Look up in dictionary fail.	**Reported**
Manpages	Educational	I.	All manual pages not found.	Manually verified
Mirrored	News	F.	Articles cannot load.	Manually verified
Tippytipper	Utility Tool	F.	Tip counting incorrectly.	Manually verified
Filexplorer	File Manager	UR.	Buttons are non-responsive.	Manually verified
Weight-Chart	Health	UD.	Text overlays on grid lines.	Manually verified
yahtzee	Game	C.	Message contradicts text.	Manually verified

The FP rate of LLMs remains high, as evidenced in Table III. Even with the setting that yields the lowest FP rate, only prompt 1 under q1 mode, the rate is still 29% in this study. Pursuing a higher TPC rate leads to an FP rate of 93%, which is unacceptable. This high rate of FP substantially increases the time users spend on verifying results from LLMs. A potential solution could be to develop a machine learning model specifically designed to filter out some of these FP cases. Additionally, training a dedicated LLM on NCF bug related documentation, such as bug reports, could optimize the model to address these limitations more effectively.

Finally, OLLM lacks a dedicated mechanism for event selection or the design of test sequences. To improve the effectiveness of NCF bug testing in Android apps, a specialized test sequence generation method is needed.

VI. Related work

Automated Android GUI testing has developed significantly, utilizing a variety of methods. Simple approaches like random testing [3] often lead to the generation of redundant and ineffective events. Tools such as DynoDroid [4] aim to enhance efficiency by minimizing this redundancy, yet they still fall short in rigorously testing comprehensive app functionalities. Meanwhile, model-based strategies [5]–[7], [49]–[54] rely on constructing a GUI model of the app to inform test generation, often employing finite state machines to delineate app states and transitions. For instance, Stoat [41] uses a stochastic Finite State Machine model to simulate the app's behavior under test. Machine learning-based testing [8], [10], [11], [55], [56] utilizes machine learning and deep learning techniques to produce test sequences that probe for Android crashes. Both QBE and DinoDroid [8], [47] apply reinforcement learning to derive testing methods for Android apps from a training set. However, these methodologies predominantly concentrate

on generating test sequences to detect crash bugs and do not address the detection of NCF bugs.

As outlined in Section II-C, some research focuses on identifying NCF bugs in Android applications. However, these studies predominantly target specific bug types within a narrow scope, rather than addressing general NCF bugs. These methods have demonstrated a modest success rate, detecting only about 2 in 399 NCF bugs when the dataset consists of bugs crawled from open-source apps on GitHub, according to a recent empirical study [18]. Beyond the NCF bug detection tool discussed in Section II-C, other approaches also work on NCF bugs. For instance, AppFlow [57] and AppTestMigrator [58] reuse test sequences in similar apps to validate functional correctness. Augusto [59] and FARLEAD-Android [60] utilize manually defined app functionalities in specification languages to detect NCF bugs. Tools like KREfinder [61] and LiveDroid [62] employ static analysis to scan the source code for specific issues related to inconsistent states of Android apps. Compared to OLLM, these tools require human input or source code access to identify NCF bugs. OLLM only needs Android Apks and test sequences generated by any of the automated Android GUI testing tools as mentioned in the last paragraph.

Research in software engineering, beyond NCF bug detection, is increasingly leveraging LLMs to enhance performance. For instance, adbGPT [63] utilizes an LLM to replicate Android bugs. Similarly, PG-TD [64] employs LLMs to generate code. GPTDroid [33] is designed to generate Android test sequences. In contrast to these works, OLLM addresses a critical bottleneck and devises a solution that transforms the challenge of general NCF bug detection from nearly impossible to feasible.

VII. Conclusion

Our study provides empirical evidence on the effectiveness of using Large Language Models (LLMs) as test oracles for detecting non-crash functional (NCF) bugs in Android applications. We propose OLLM, which employs an LLM to determine whether a given test sequence reveals an NCF bug, using two tailored prompts. Our results indicate that OLLM outperforms state-of-the-art tools in detecting Android NCF bugs. Additionally, we found that the two-prompt strategy, in-context learning, and prompt rules significantly enhance OLLM's performance. Among the five different LLM models tested, GPT-4o demonstrated outstanding performance compared with others. Furthermore, we show that OLLM can uncover previously unknown functionalities in real-world applications. By highlighting some limitations of OLLM, our study provides a foundation for future research and development, aiming to refine and enhance the effectiveness of LLM-based test oracles in Android apps.

VIII. Acknowledgements

This work was partially supported by the Ohio Department of Higher Education (ODHE) through the Strategic Ohio Council for Higher Education (SOCHE) and Ohio Cyber Range Institute (OCRI) sub-awards and by the National Science Foundation (NSF) award CCF-2342355 and CCF-2403747.

References

[1] "Google Play Data," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store, 2020.

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

[3] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html, 2022.

[4] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proc. of the Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proc. of the International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2015.

[7] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, 2013, pp. 641–660.

[8] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: Qlearning-based exploration of android applications," in *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 105–115.

[9] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proc. of the International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[10] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: a deep learning-based approach to automated black-box android app testing," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1070–1073.

[11] C. Degott, N. P. Borges Jr, and A. Zeller, "Learning user interface element interactions," in *Proc. of the International Symposium on Software Testing and Analysis*, 2019, pp. 296–306.

[12] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, 2014.

[13] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *Proc. of 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 399–410.

[14] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proc. of 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 308–318.

[15] W. Guo, Z. Dong, L. Shen, W. Tian, T. Su, and X. Peng, "ifixdataloss: a tool for detecting and fixing data loss issues in android apps," in *Proc. of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 785–788.

[16] J. Sun, "Setdroid: detecting user-configurable setting issues of android apps via metamorphic fuzzing," in *Proc. of the 43rd International Conference on Software Engineering: Companion Proceedings*. IEEE, 2021, pp. 108–110.

[17] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *Proc. of the 2017 IEEE Symposium on security and privacy (SP)*. IEEE, 2017, pp. 615–632.

[18] 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 *Proc. of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1319–1331.

[19] "Github," https://github.com, 2012.

[20] 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," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–31, 2021.

[21] J. Wang, Y. Jiang, T. Su, S. Li, C. Xu, J. Lu, and Z. Su, "Detecting non-crashing functional bugs in android apps via deep-state differential analysis," in *Proc. of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 434–446.

[22] C. Escobar-Velásquez, M. Osorio-Riaño, J. Dominguez-Osorio, M. Arevalo, and M. Linares-Vásquez, "An empirical study of i18n collateral changes and bugs in guis of android apps," in *Proc. of 2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2020, pp. 581–592.

[23] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 398–409.

[24] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[25] S. Ozdemir, *Quick Start Guide to Large Language Models: Strategies and Best Practices for Using ChatGPT and Other LLMs*. Addison-Wesley Professional, 2023.

[26] "Gpt-4o," https://openai.com/index/hello-gpt-4o/, 2024.

[27] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[28] "Ollm," https://github.com/randomNameless/OLLM, 2024.

[29] "Amaze-2113," https://github.com/TeamAmaze/AmazeFileManager/issues/2113, 2024.

[30] 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 *Proc. of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 204–215.

[31] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 47–51.

[32] D. Wang, Y. Zhao, S. Feng, Z. Zhang, W. G. Halfond, C. Chen, X. Sun, J. Shi, and T. Yu, "Feedback-driven automated whole bug report reproduction for android apps," *arXiv preprint arXiv:2407.05165*, 2024.

[33] 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.

[34] "Uiautomator," https://developer.android.com/training/testing/other-components/ui-automator, 2024.

[35] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, "Decomposed prompting: A modular approach for solving complex tasks," *International Conference on Learning Representations*, 2023.

[36] "Mirrored," https://github.com/homac/Mirrored/, 2024.

[37] "Ankidroid-7232," https://github.com/ankidroid/Anki-Android/issues/7232, 2024.

[38] S. Deng, W. Xu, H. Sun, W. Liu, T. Tan, J. Liu, A. Li, J. Luan, B. Wang, R. Yan *et al.*, "Mobile-bench: An evaluation benchmark for llm-based mobile agents," *arXiv preprint arXiv:2407.00993*, 2024.

[39] "Wordpress-15026," https://github.com/wordpress-mobile/WordPress-Android/issues/15026, 2024.

[40] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proc. of the International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[41] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proc. of the Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[42] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proc. of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[43] "Gpt-4," https://openai.com/index/gpt-4/, 2024.

[44] "Gemini," https://gemini.google.com/, 2024.

[45] "ChatGLM-4," https://chatglm.cn/, 2024.

[46] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proc. of the International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.

[47] Y. Zhao, B. Harrison, and T. Yu, "Dinodroid: Testing android apps using deep q-networks," *ACM Transactions on Software Engineering and Methodology*, 2022.

[48] C. Callison-Burch, M. Osborne, and P. Koehn, "Re-evaluating the role of bleu in machine translation research," in *11th conference of the european chapter of the association for computational linguistics*, 2006, pp. 249–256.

[49] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *Proc. of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[50] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 238–249.

[51] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 103–114.

[52] J. Yan, L. Pan, Y. Li, J. Yan, and J. Zhang, "Land: a user-friendly and customizable test generation tool for android apps," in *Proc. of the International Symposium on Software Testing and Analysis*, 2018, pp. 360–363.

[53] J. Yan, T. Wu, J. Yan, and J. Zhang, "Widget-sensitive and back-stack-aware gui exploration for testing android apps," in *Proc. of the International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 42–53.

[54] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proc. of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013, pp. 250–265.

[55] N. P. Borges, M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proc. of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2018, pp. 133–143.

[56] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2019, pp. 42–53.

[57] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 269–282.

[58] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[59] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles," in *Proc. of the 40th International Conference on Software Engineering*, 2018, pp. 280–290.

[60] Y. Koroglu and A. Sen, "Functional test generation from ui test scenarios using reinforcement learning for android applications," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1752, 2021.

[61] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in android applications," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 864–880, 2016.

[62] U. Farooq, Z. Zhao, M. Sridharan, and I. Neamtiu, "Livedroid: Identifying and preserving mobile app state in volatile runtime environments," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

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

[64] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," *arXiv preprint arXiv:2303.05510*, 2023.